A Critique of Cyclomatic Complexity as a Software Metric

Article *in* Software Engineering Journal · April 1988

DOI: 10.1049/sej.1988.0003 · Source: IEEE Xplore

1 author:

Martin John Shepperd
Brunel University London
185 PUBLICATIONS **9,161** CITATIONS

SEE PROFILE

A critique of cyclomatic complexity as a software metric

by Martin Shepperd

McCabe's cyclomatic complexity metric is widely cited as a useful predictor of various software attributes such as reliability and development effort. This critique demonstrates that it is based upon poor theoretical foundations and an inadequate model of software development. The argument that the metric provides the developer with a useful engineering approximation is not borne out by the empirical evidence. Furthermore, it would appear that for a large class of software it is no more than a proxy for, and in many cases is outperformed by, lines of code.

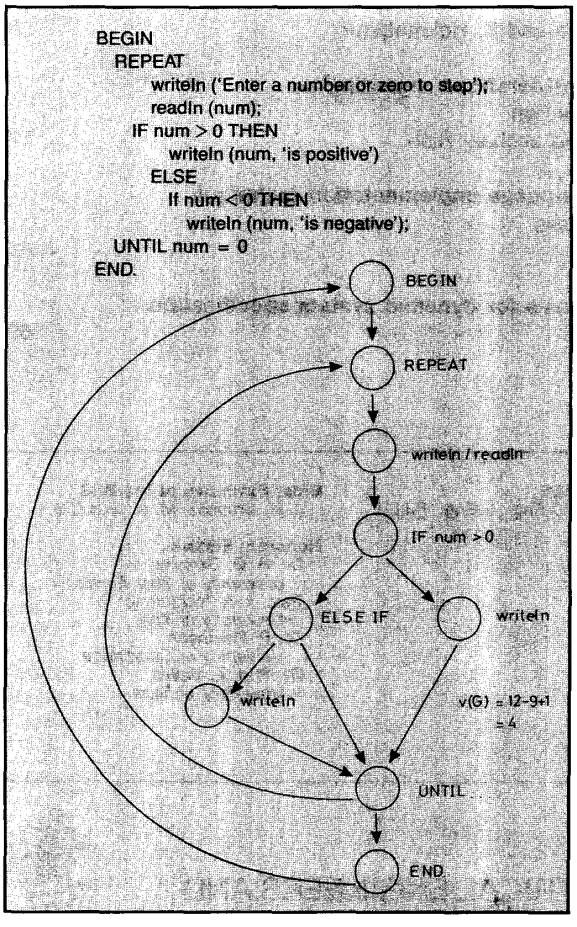

Fig. 1 Derivation of $v(G)$ for an example program

1 Introduction

The need for some objective measurement of software complexity has been long acknowledged. Two early contributions to this field are Halstead's 'software science' (Ref. 1) and the cyclomatic complexity approach of McCabe (Ref. 2). Both metrics are based upon the premise that *software complexity is strongly related to various measurable* properties of program code.

Although initially well received by the software engineering community, software science based metrics have been increasingly subject to criticism. Attacks have been made upon the underlying pyschological model (Refs. 3 and 4). The soundness of many empirical 'validations' has been questioned (Ref. 5) and difficulties noted with counting rules (Ref. 6). The ability of software science metrics to capture program complexity in general would thus appear to be in great doubt.

It is thus rather surprising that the cyclomatic complexity metric has not been subjected to a similar degree of scrutiny to that given to software science. This is particularly the case given the high degree of acceptance of the metric within the software engineering community. It is widely cited (Refs. 7 – 13), subjected to a 'blizzard of refinements' (Refs. 14 – 22), applied as a design metric (Ref. 23) and described in best-selling textbooks on software engineering (Refs. 24 and 25). Yet there have been comparatively few empirical studies; indeed, as a basic approach, the metric has been allowed to pass relatively unquestioned.

The hypothesis of a simple deterministic relationship between the number of decisions within a piece of software and its complexity is potentially of profound importance to the whole field of software engineering. This requires very careful evaluation.

The rest of the paper reviews the theories put forward by McCabe. Theoretical criticisms of the metric are outlined and the various empirical validations for the metric are reviewed, together with aspects of experimental design. It is concluded that cyclomatic complexity is questionable on both theoretical and empirical grounds. Therefore cyclomatic complexity is of very limited utility.

2 The cyclomatic complexity metric

Given the increasing costs of software development, McCabe considered that a 'mathematical technique that will provide a quantitative basis for modularisation and allow us to identify software modules that will be difficult to test or maintain' was required. Use of a lines of code (LOC) metric was rejected since McCabe could see no obvious relationship between length and module complexity. Instead, he suggested that the number of control paths through a module would be a better indicator, particularly as this appeared to be strongly related to testing effort. Furthermore, much of

the work on 'structured programming' in the early 1970s concentrated on program control flow structures (Refs. 26 and 27).

Unfortunately, the number of paths through any software with a backward branch is potentially infinite. Fortunately, the problem can be resolved by the application of graph theory. The control flow of any procedural piece of software can be depicted as a directed graph, by representing each executable statement (or group of statements where the flow of control is sequential) as a node, and the flow of control as the edges between them. The cyclomatic complexity of a graph is useful because, providing the graph is strongly connected, it indicates the number of basic paths (i.e. linearly independent circuits) contained within a graph, which, when used in combination, can generate all possible paths through the graph or program.

The cyclomatic complexity v of a program graph G is

$$v(G) = e - n + 1 \quad (1)$$

where e is the number of edges, and n is the number of nodes.

A strongly connected graph is one for which given any two nodes r and s there exist paths from r to s and s to r. Fig. 1 shows an example derivation of cyclomatic complexity from a simple program and its related control graph. Note that the program graph is made strongly connected by the addition of an edge connecting the END node to the BEGIN node.

The process of adding an extra edge to the program graph can be bypassed by adding one to the cyclomatic complexity calculation. The calculation can be generalised for program graphs that contain one or more components, subject to the restriction that each component contains a single entry and a single exit node. For a graph S with a set of connected components the cyclomatic complexity is

$$v(S) = e - n + 2p \quad (2)$$

where p is the number of connected components.

A multi-component program graph is derived if the software contains separate subroutines. This is illustrated in Fig. 2.

As McCabe observed, the calculation reduces to a simple count of conditions plus one. He argued that since a compound condition, for example

```
IF X < 1 AND Y <2 THEN
```

was a thinly disguised nested IF, then each condition should contribute to module complexity, rather than merely counting predicates (see Figs. 3*a* and *b*). Likewise a case statement is viewed as a multiple IF statement (i.e. it contributes $n - 1$ to $v(G)$, where n is the number of cases).

McCabe saw a practical application of the metric in using it to provide an upper limit to module complexity, beyond which a module should be subdivided into simpler components. A value of $v(G) = 10$ was suggested, although he accepted that in certain situations, notably large case structures, the limit might be relaxed.

3 Theoretical considerations

The counting rules for different control statements have been the subject of some controversy. Myers (Ref. 19) has argued that a complexity interval is a more effective measure of complexity than a simple cyclomatic number. The interval has a lower bound of decision statement count (i.e. predicate count) plus one and an upper bound of individual condition count plus one.

Myers used the following three examples to support his

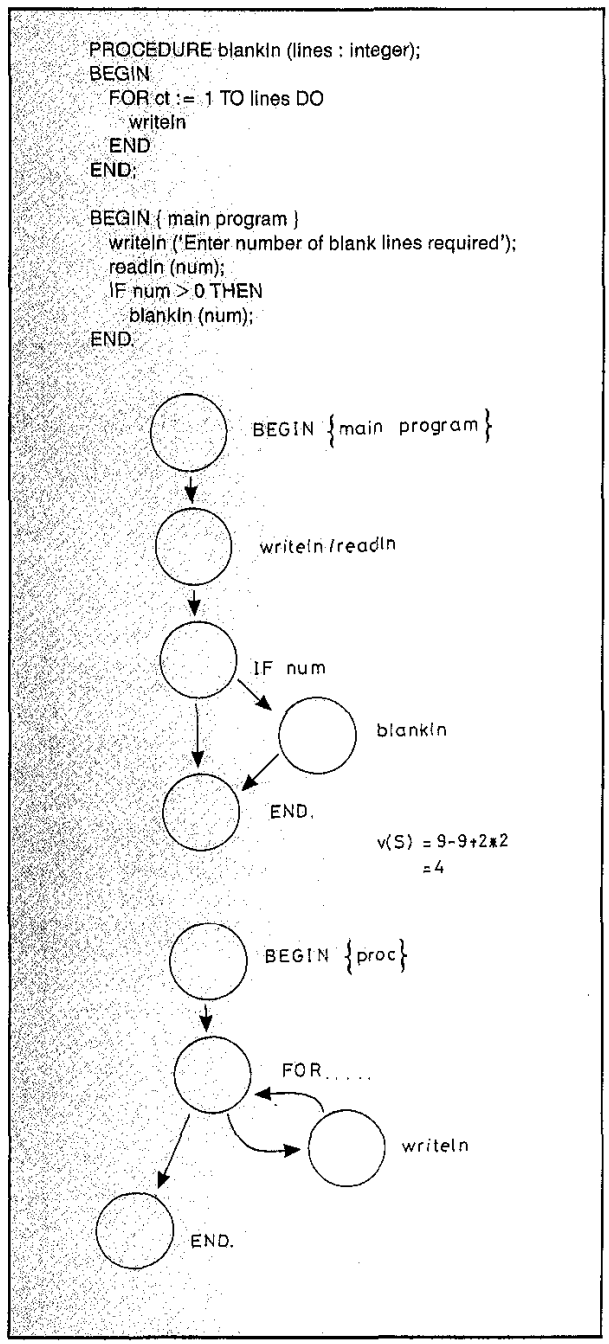

Fig. 2 Derivation of $v(S)$ for a program with a subroutine

modified form of the cyclomatic complexity metric:

```
IF X = 0 THEN ...                    v(G) = 2
       ELSE ...;                     Myers = (2:2)

IF X = 0 AND Y > 1 THEN ...          v(G) = 3
                ELSE ...;            Myers = (2:3)

IF X = 0 THEN
       IF Y > 1 THEN ...
                ELSE ...             v(G) = 3
       ELSE . . .;                   Myers = (3:3)
```

His argument is that it is intuitively obvious that the third example is more complex than the second, a distinction not made by the cyclomatic number. The idea underlying his

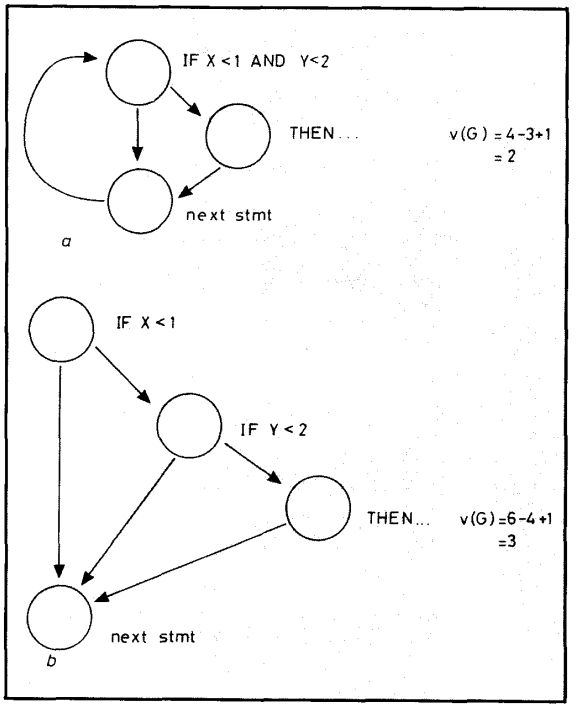

Fig. 3 Compound condition treated as a single decision and as separate decisions

a Treated as a single decision

b Treated as separate decisions

modification appears to be that there is more potential for inserting additional ELSE clauses into a program with a larger number of IF statements. They are not counted by the McCabe metric, as is demonstrated by the following two program fragments, both of which have cyclomatic complexities of 2:

```
IF X<1 THEN
    ...;                                  v(G) = 2

IF X<1 THEN
    ...
ELSE
    ...;                                  v(G) = 2
```

Since Myers' complexity interval does not directly count ELSE statements it is arguable whether it represents much of an improvement over that of McCabe's metric. However, the criticism of cyclomatic complexity remains, in that it fails to distinguish between selections with and without ELSE branches. From the standpoint of pyschological complexity this is significant; however, since the number of basic paths remains unaltered testing difficulty may not increase. Thus the failure of cyclomatic complexity to count ELSE branches is only a serious deficiency if the metric is intended to capture complexity of comprehension.

The treatment of case statements has also been subject to disagreement. Hansen (Ref. 15) has suggested that since they were easier to understand than the equivalent nested IFs they should only contribute one to the module complexity. Other researchers (Ref. 28) have suggested a $\log_2(n)$ relationship, where n is the number of cases. Evangelist (Ref. 29) also encountered anomalies in the application of counting rules. Much of the difficulty stems from the fact that McCabe was originally thinking in terms of Fortran, whereas most of these difficulties arise from other languages, some of them more recent, such as Ada.† Here one has to contend with problems such as distinguishing between 'IF y = 1 OR y = 3' and 'IF y = 0 OR ELSE x/y>1'. The mapping from code to a program graph is ambiguous.

Another area of controversy is that $v = 1$ will remain true for a linear sequence of any length. Since the metric is insensitive to complexity contributed from linear sequences of statements, several researchers have suggested modifications to the simple use of cyclomatic complexity. Hansen has proposed a 2-tuple of cyclomatic complexity and operand count (defined to be arithmetical operators, function and subroutine calls, assignments, input and output statements and array subscription). Unfortunately, as Baker and Zweben (Ref. 30) point out, this approach does suffer from the problem of 'comparing apples and oranges'. It is not clear how to rank in order of complexity the 2-tuples (i,j) and (l,k) where $i>l$ and $k>j$.

Stetter (Ref. 21) suggests an alternative approach to this particular problem in the form of a cyclomatic flow complexity metric. Flow of data is considered in addition to flow of control. Complexity will generally increase with an increase in length of a linear sequence of statements since more data references will almost invariably be made.

A further objection to the cyclomatic complexity metric is its behaviour towards the structuring of software. A number of researchers (Refs. 30 – 33) argue that the cyclomatic complexity can increase when applying generally accepted techniques to improve program structure. Certainly the metric is insensitive to the use of unstructured techniques such as jumping in and out of loops, since all that is captured is the number of decisions plus one. Evangelist (Ref. 34) reports that the application of only 2 out of 26 of Kernighan and Plauger's rules of good programming style (Ref. 35) invariably results in a decrease in cyclomatic complexity.

A development of the unstructured/structured argument is the objection that the metric ignores the context or environment of a decision. All decisions have a uniform weight, regardless of depth of nesting or relationship with other decisions. The complexity of a decision cannot be considered in isolation, but must take into account other decisions within its scope. This has resulted in variants of cyclomatic complexity which allow for nesting depth (Refs. 18, 32 and 36).

It is worth noting that all counting rule variants to the metric are based upon arguments along the lines that it is intuitively obvious that one example is more complex than another and therefore an adjustment must be made to the counting rules. Such arguments are based upon issues of cognitive complexity or 'perplexity' (Ref. 37), which is only one view of software complexity. Difficulty of testing is another aspect of software complexity and one with which McCabe was primarily concerned. These different interpretations of cyclomatic complexity have significant implications upon the validation and application of the metric.

A more fundamental objection to cyclomatic complexity is the inconsistent behaviour when measuring modularised software. As Eqn. 2 indicates, $v(G)$ is sensitive to the number of subroutines within a program, because McCabe suggests that these should be treated as unconnected components within the control graph. This has the bizarre result of increasing overall complexity as a program is divided into more, presumably simpler, modules. In general, the complexity v of a program P' will be:

$$v(P') = v(P) + i \qquad (3)$$

where P is equivalent program to P' but with a single

†Ada is a trademark of the US Government Ada Joint Program Office.

component, and i is the number of modules or subroutines used by P' (i.e. the number of graph components − 1).

However, the relationship is further complicated by the observation that graph complexity may be reduced in a situation where modularisation eliminates code duplication. Thus

$$v(P') = v(P) + i - \sum_{j=1}^{j=i} ((v_j - 1) * (u_j - 1)) \quad (4)$$

where v_j is the complexity of the jth module or subroutine, and u_j is the number of times the jth module is called.

To summarise, general program complexity increases with the addition of extra modules but decreases with the factoring out of duplicate code. All other aspects of modularity are disregarded. If one were to be prescriptive on the basis of Eqn. 4, it would be to only modularise when a fragment of non-linear code (i.e. containing decisions) could be factored out. As a model with which to view general software complexity, this appears unacceptable.

Three classes of theoretical objection have been presented. First, there is the issue of the very simplistic approach to decision counting. Ease of program comprehension is unlikely to be completely orthogonal to software complexity. The ease of comprehending a decision is not invariant, and thus a constant weighting of one seems inappropriate. Secondly, the metric appears to be independent of generally accepted program structuring techniques. Since these are intended to reduce complexity this does not exactly inspire confidence. Thirdly, and most importantly, is the arbitrary impact of modularisation upon total program complexity. As a measure of inter-modular complexity, in other words for all non-trivial software, cyclomatic complexity would seem unsatisfactory on theoretical grounds.

4 Empirical validation of the metric

Many early validations of the metric were merely based upon intuitive notions of complexity. For example, McCabe states that 'the complexity measure v is designed to conform to our intuitive notion of complexity' (Ref. 2). Hansen (Ref. 15) argues that a good measure of program complexity should satisfy several criteria, including that of relating 'intuitively to the psychological complexity of programs'. He does not suggest that there is a need for any objective validation. Likewise, Myers (Ref. 19) treats intuition as sufficient grounds for employing the metric.

This seems a rather curious approach: if intuition is a reliable arbiter of complexity this eliminates the need for a quantitative measure. On the other hand, if intuition cannot be relied upon, it hardly provides a reasonable basis for validation. Clearly a more objective approach to validation is required.

The theoretical objections to the metric, that it ignores other aspects of software such as data and functional complexity, are not necessarily fatal. It is easy to construct certain pathological examples, but this need not invalidate the metric if it is possible to demonstrate that in practice it provides a useful engineering predictor of factors that are associated with complexity. Researchers have usually taken these to include effort involved in testing and maintenance, error incidence and ability to recall code.

A number of empirical studies have been carried out. These are summarised in Table 1. A difficulty that arises in interpretation of many of these studies is that there is no explicit hypothesis being evaluated. Two possible *a posteriori* hypotheses with which to examine the empirical work are as follows:

- *Hypothesis 1:* Total program cyclomatic complexity can be used to predict various useful software characteristics (for example development time, incidence of errors and program comprehension).
- *Hypothesis 2:* Programs comprising modules with low $v(G)$ (<10) are easier to test and maintain than those for which this is not the case (McCabe's original hypothesis).

As Table 1 indicates, the results of various empirical validation studies do not give a great deal of support to either hypothesis. In general the results are not very compelling, either at the program level (hypothesis 1) or for the studies

Table 1 Empirical validations of cyclomatic complexity

Researchers	LOC	Errors		Programming effort	Bug location	Program recall	Design effort
		density	*absolute*				
Basili (Ref. 38)	r^2=0.94	r is –ve					
Basili (Ref. 39)				R=0.48	R=0.21		
Bowen (Ref. 40)			r^2=0.47		r=–0.09		
Curtis (Ref. 41)	r^2=0.41,0.81,0.79					r=–0.35*	
Curtis (Ref. 42)	r^2=0.81,0.66				r^2=0.4,0.42		
Davis (Ref. 43)						r is –ve, +ve	
Feuer (Ref. 44)	r^2=0.90***						
Gaffney (Ref. 45)				r^2=0.60			
Henry (Ref. 46)	r^2=0.84***		r^2=0.92****				
Kitchenham (Ref. 47)	r^2=0.86,0.88		r^2=0.46,0.49,0.21****				
Paige (Ref. 48)	r^2=0.90						
Schneiderman (Ref. 49)	r^2=0.61*****		r^2=0.32*****				
Shen (Ref. 50)			r^2=0.78***				
Sheppard (Ref. 51)	r^2=0.79			r^2=0.38		r=0.35	
Sunohara (Ref. 52)				r^2=0.4,0.38			r^2=0.72,0.7
Wang (Ref. 53)	r^2=0.62			r^2=0.59			
Woodfield (Ref. 54)				r^2=0.26,R=0.39			
Woodward (Ref. 22)	r^2=0.90						

r^2 = Pearson moment R = Rank Spearman

* r was 'improved' when modified for potentially 'aberrant' results

** correlated with N (i.e. Halstead's token count)

*** a simple decision count (i.e. $v(G)$–1)

**** indirect error count (i.e. version count), or program change count

***** using log-log transformations

that deal with individual modules (hypothesis 2), such as Basili and Perricone (Ref. 38). The major exception is the Henry *et al.* (Ref. 46) study of 165 procedures from the UNIX§ operating system, where the results show a strong correlation between $v(G)$ and module error rates. This result may be slightly artificial since they appear to have filtered out all error-free modules.

Based upon the observation that large modules tend to contain more errors than small modules, the Basili and Perricone (Ref. 38) study uses error density (i.e. errors per thousand LOC) as a size-normalised metric of software error-proneness. Their rather surprising finding was that error density diminishes with increasing cyclomatic complexity. Work by Shen *et al.* (Ref. 50) gives support to this result, although there is disagreement as to whether error density is an appropriate means of size normalisation since module size and error density do not appear to be independent. Nevertheless, this strongly underlines the deficiency of a simple intra-modular complexity metric.

The clearest result from the empirical studies is the strong relationship between cyclomatic complexity and LOC. Even in the study of Henry *et al.* there appears to be a fairly strong association. Ironically it was the 'inadequacy' of LOC as a module complexity metric that led to McCabe proposing cyclomatic complexity as an alternative. A considerable number of studies (Refs. 41, 47, 48, 53 and 55) indicate that LOC actually outperforms cyclomatic complexity.

The most reasonable inference that can be drawn from the above studies is that there exists a significant class of software for which $v(G)$ is no more than a proxy for LOC. A suggestion of Henry *et al.* (Ref. 46) that software can be characterised as either decision or computation bound could have a considerable bearing upon interpretation of empirical studies. In cases of decision-bound software such as UNIX, $v(G)$ will closely correspond to LOC. In computation-bound software, with sizeable portions of linear code this correspondence will be very marginal, and possibly accounts substantially for the erratic results of Table 1.

An interesting development of this point has been made by Humphreys (Ref. 56), who argues that there exists a trade-off between decision or control flow complexity and data structure complexity. One such example is the use of decision tables to replace multiple IF or CASE statements (a common technique in systems programming). The consequence of this is that the cyclomatic complexity for the decision table solution will be substantially lower than for the alternative solution. Yet, he argues, the two pieces of software appear to have similar complexities. More significantly, they will require a similar amount of testing effort since they have the same number of boundary conditions to contend with. Thus the claimed association between testing difficulty and $v(G)$ in many cases is distinctly tenuous. The suggestion has been made (Ref. 57) that this is due to McCabe's ambiguous mapping function of program control flow to a program graph. Either way it does not bode well for cyclomatic complexity as a predictor of testing effort.

Most of the studies reported above place reliance upon obtaining high correlation coefficients. Use of Pearson's product moment, which is the most widely used correlation coefficient in the studies above, requires the assumption that the data is drawn from a population with a roughly normal distribution. This creates a particular problem when examining module error rates. The impossibility of a negative error count results in a pronounced skew in the error distribution. This skew can be reduced by various transformation techniques, for instance by using the square root or logarithm. Studies such as Refs. 40 and 46 would be more meaningful if one of these techniques were applied to obtain a more normal distribution so we could place a higher degree of confidence in the correlation coefficients produced.

There are two alternative empirical approaches; both have considerable difficulties associated with them. The first is large-scale passive observation, where the researcher has little if any influence. The second is more carefully controlled experimentation, which out of practical necessity tends to be very small scale; see for example Refs. 22, 41, 42 and 53.

Large-scale passive observation is based upon the notion that the variance introduced into the study from uncontrolled factors such as differences in individual ability, task difficulty and differing environments is compensated by the large sample size involved. Problems include the difficulty of obtaining accurate measurements (Ref. 39). Their results showed some improvement when restricted to results validated by various cross-checks. More significant is the problem of variation in individual ability (Ref. 58). Brooks (Ref. 59) suggests that differences in ability for individuals from similar backgrounds of up to 25 to 1 are such as to make it very difficult to obtain statistically significant results.

The second approach, as typified by Ref. 54, is more carefully controlled since the timescales and number of subjects are relatively small. Here measurements are potentially more accurate; however variance from external factors is still a major difficulty. Use of within-subject experimental design is a partial solution, although it does not address a number of factors, such as the subject's familiarity with the problem and the comparability of tasks. The small size of tasks being undertaken is another problem area; frequently programs of less than 300 LOC (Refs. 43, 44, 53 and 54) are used. These programs are, by software engineering standards, trivial. In such situations the onus is upon the researcher to demonstrate that results at a small scale are equally applicable for large systems. Such a finding would be counter to current directions in software engineering.

To summarise, many of the empirical validations of McCabe's metric need to be interpreted with caution. First the use of correlation coefficients on skewed data causes artificially high correlations. Secondly, the assumption of causality would seem doubtful given the consistently high association between cyclomatic complexity and LOC. Thirdly, the high variation in programmer ability reduces the statistical significance of correlation coefficients.

However, despite the above reservations, some trends in the results are apparent. The strong association between LOC and cyclomatic complexity gives the impression that the latter may well be no more than a proxy for the former. The ability of $v(G)$ to predict error rates, development time and program recall is quite erratic. Most damning is the outperforming of $v(G)$ by a straightforward LOC metric in over a third of the studies considered.

5 Conclusions

A severe difficulty in evaluating McCabe's metric and associated empirical work is the lack of explicit model upon which cyclomatic complexity is based. The implicit model appears to be that the decomposition of a system into suitable components (or modules) is the key issue. The decomposition should be based upon ease of testing individual components. Testing difficulty is entirely determined by the number of basic paths through a program's flowgraph.

Unfortunately, and perhaps not surprisingly, different investigators have interpreted cyclomatic complexity in a variety of ways. For example, some studies treat cyclomatic complexity at a program level by summing individual module complexities (Ref. 54), while others consider com-

§UNIX is a trademark of AT & T Bell Laboratories.

plexity purely at a module level (Ref. 46). Naturally this state of affairs does not facilitate the comparison of results.

An important distinction is made between intra- and inter-modular complexity. Eqns. 3 and 4 suggest that cyclomatic complexity is rather suspect in the latter area. Thus the only possible role for cyclomatic complexity is as an *intra*-modular complexity metric. Even this is made to look doubtful in the light of the work of Basili and Perricone. In any case, many researchers (Ref. 60) would argue that the problem of how to modularise a program is better resolved by considerations of 'coupling' and 'cohesion' (i.e. *inter*-modular complexity), which are not adequately captured by the metric.

As noted earlier, most of the empirical work has relied upon obtaining high correlation coefficients to substantiate McCabe's metric. However, a high correlation coefficient between two variables does not necessarily imply causality, as illustrated by the well known, if slightly apochryphal, example of the spatial distribution of ministers of religion and prostitutes! Setting aside quibbles of experimental methodology (Refs. 59 and 61), the fundamental problem remains that without an explicit underlying model the empirical 'validation' is meaningless and there is no hypothesis to be refuted.

Even if we disregard all the above problems and accept the correlation coefficients at face value, the results are distinctly erratic. Cyclomatic complexity fails to convince as a general software complexity metric. This impression is strengthened by the close association between $v(G)$ and LOC and the fact that for a significant number of studies LOC outperforms $v(G)$.

The majority of modifications to McCabe's original metric remain untested. To what extent do validations of cyclomatic complexity impinge upon these modified metrics, many of which appear to be very minor variants? Prather (Ref. 32), in an attempt to provide some unifying framework, suggests a set of axioms which a 'proper' complexity metric must satisfy:

- *Axiom 1:* The complexity of the whole must not be less than the sum of the complexities of the parts.
- *Axiom 2:* The complexity of a selection must be greater than the sum of all the branches (i.e. the predicate must contribute complexity).
- *Axiom 3:* The complexity of an iteration must be greater than the iterated part (for the same reason as axiom 2).

Although an interesting idea, a number of problems remain. First, the axioms are limited to structured programs. Secondly, the axioms provide very little constraint upon the imaginations of software complexity metrics designers. Thirdly, the axioms, however reasonable, are based purely upon arguments of intuition. This is particularly the case for Prather's suggestion of an upper bound of twice the lower bound for axioms 2 and 3. Finally, the underlying model is incomplete, in as much as there are no connections with observable events in the software development process.

This axiomatic approach has been further developed (Refs. 30, 62 and 63) such that any program may be reduced into a hierarchy of irreducibles (prime trees). The benefits are the removal of subjectivity over the issue of counting rules and the ability to draw comparisons between different metrics. Still unresolved are the problems of using intuition when deriving actual complexity values from different irreducibles and the construction of a complete model of the relevant world for a complexity metric. The difficulty is, of course, that the 'real world' is not entirely formal, in the sense that we cannot model it with precise mathematical relationships. The best we can hope for is engineering approximations.

It is arguable that the search for a general complexity metric based upon program properties is a futile task. Given the vast range of programmers, programming environments, programming languages and programming tasks, to unify them into the scope of a single complexity metric is an awesome task. A more fruitful approach might be to derive metrics from the more abstract notations and concepts of software designs. This would have the additional advantage that design metrics are available earlier on in the software development process.

For a software complexity metric to be treated seriously by the software engineering community, considerably more emphasis must be placed on the validation process. It may well be 'intellectually very appealing' (Ref. 22) but this is insufficient. Following from the suggestion (Ref. 55) that the LOC metric be regarded as a 'baseline' for the evaluation of metrics, there must exist considerable doubts about the utility of McCabe's cyclomatic complexity metric.

6 Acknowledgments

The author would like to thank Prof. Darrel Ince of the Open University, Milton Keynes, England, for the many useful suggestions and kind help he has given during the preparation of this paper. He would also like to record his thanks towards the referee who provided constructive criticism and a number of additional insights.

7 References

1 HALSTEAD, M.H.: 'Elements of software science' (North-Holland, 1977)
2 McCABE, T.J.: 'A complexity measure', *IEEE Transactions on Software Engineering*, 1976, **2**, (4), pp. 308 – 320
3 COULTER, N.S.: 'Software science and cognitive psychology', *IEEE Transactions on Software Engineering*, 1983, **9**, (2), 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 Transactions on Software Engineering*, 1983, **9**, (2), pp. 155 – 165
5 HAMER, P.G., and FREWIN, G.D.: 'M.H. Halstead's software science – a critical examination'. Proceedings of Sixth International Conference on Software Engineering, Tokyo, Japan, 1982
6 LASSEZ, J.L., VAN DER KNIJFF, D., SHEPHERD, J., and LASSEZ, C.: 'A critical examination of software science', *Journal of Systems & Software*, 1981, **2**, pp. 105 – 112
7 ARTHUR, L.J.: 'Measuring programmer productivity and software quality' (Wiley-Interscience, 1985)
8 COBB, G.W.: 'A measurement of structure for unstructured languages'. Proceedings of ACM SIGMETRICS/SIGSOFT Software Quality Assurance Workshop, 1978
9 DE MARCO, T.: 'Controlling software projects: management, measurement and estimation' (Yourdon Press, 1982)
10 DUNSMORE, H.E.: 'Software metrics: an overview of an evolving methodology', *Information Processing & Management*, 1984, **20**, (1 – 2), pp. 183 – 192
11 HARRISON, W., MAGEL, K., KLUCZNY, R., and DE KOCK, A.: 'Applying software complexity metrics to program maintenance', *Computer*, 1982, **15**, Sept., pp. 65 – 79
12 SCHNEIDEWIND, N.F.: 'Software metrics for aiding program development and debugging'. National Computer Conference, New York, NY, USA, Jun. 1979, AFIPS Conference Proceedings Vol. 48, pp. 989 – 994
13 TANIK, M.M.: 'A comparison of program complexity prediction models', *SIGSOFT Software Engineering Notes*, 1980, **5**, (4), pp. 10 – 16
14 CURTIS, B.: 'Software metrics: guest editor's introduction', *IEEE Transactions on Software Engineering*, 1983, **9**, (6), pp. 637 – 638
15 HANSEN, W.J.: 'Measurement of program complexity by the pair (cyclomatic number, operator count)', *SIGPLAN Notices*, 1978, **13**, (3), pp. 29 – 33

16 HARRISON, W., and MAGEL, K.: 'A complexity measure based on nesting level', *SIGPLAN Notices*, 1981, **16**, (3), pp. 63 – 74
17 IYENGAR, S.S., PARAMESWARAN, N., and FULLER, J.: 'A measure of logical complexity of programs', *Computer Languages*, 1982, **7**, pp. 147 – 160
18 MAGEL, K.: 'Regular expressions in a program complexity metric', *SIGPLAN Notices*, 1981, **16**, (7), pp. 61 – 65
19 MYERS, G.J.: 'An extension to the cyclomatic measure of program complexity', *SIGPLAN Notices*, 1977, **12**, (10), pp. 61 – 64
20 OVIEDO, E.: 'Control flow, data flow and program complexity' Proceedings of COMPSAC 80 Conference, Buffalo, NY, USA, Oct. 1980, pp. 146 – 152
21 STETTER, F.: 'A measure of program complexity', *Computer Languages*, 1984, **9**, (3), pp. 203 – 210
22 WOODWARD, M.R., HENNELL, M.A., and HEDLEY, D.A.: 'A measure of control flow complexity in program text', *IEEE Transactions on Software Engineering*, 1979, **5**, (1), pp. 45 – 50
23 HALL, N.R., and PREISER, S.: 'Combined network complexity measures', *IBM Journal of Research & Development*, 1984, **28**, (1), pp. 15 – 27
24 PRESSMAN, R.S.: 'Software engineering. A practitioner's approach' (McGraw-Hill, 1987), Second Edition
25 WIENER, R., and SINCOVEC, R.: 'Software engineering with Modula-2 and Ada' (Wiley, 1984)
26 DAHL, O.J., DIJKSTRA, E.W., and HOARE, C.A.R.: 'Structured programming' (Academic Press, 1972)
27 DIJKSTRA, E.W.: 'Goto statement considered harmful', *Communications of ACM*, 1968, **18**, (8), pp. 453 – 457
28 BASILI, V.R., and REITER, R.W.: 'Evaluating automatable measures of s/w development'. Proceedings of IEEE Workshop on Quantitative Software Models, 1979, pp. 107 – 116
29 EVANGELIST, W.M.: 'Relationships among computational, software and intuitional complexity', *SIGPLAN Notices*, 1983, **18**, (12), pp. 57 – 59
30 BAKER, A.L., and ZWEBEN, S.: 'A comparison of measures of control flow complexity', *IEEE Transactions on Software Engineering*, 1980, **SE-6**, (6), pp. 506 – 511
31 OULSNAM, G.: 'Cyclomatic numbers do not measure complexity of unstructured programs', *Information Processing Letters*, 1979, **8**, pp. 207 – 211
32 PRATHER, R.E.: 'An axiomatic theory of software complexity metrics', *Computer Journal*, 1984, **27**, (4), pp. 340 – 347
33 SINHA, P.K., JAYAPRAKASH, S., and LAKSHMANAN, K.B.: 'A new look at the control flow complexity of computer programs', *in* BARNES, D., and BROWN, P.: 'Software engineering 86'. Proceedings of BCS-IEE Software Engineering 86 Conference, Southampton, England, Sept. 1986 (Peter Peregrinus, 1986), pp. 88 – 102
34 EVANGELIST, W.M.: 'Software complexity metric sensitivity to program structuring rules', *Journal of Systems & Software*, 1982, **3**, pp. 231 – 243
35 KERNIGHAN, B.W., and PLAUGER, P.: 'The elements of programming style' (McGraw-Hill, 1978)
36 PIOWARSKI, P.: 'A nesting level complexity measure', *SIGPLAN Notices*, 1982, **17**, (9), pp. 40 – 50
37 WHITTY, R.W., and FENTON, N.E.: 'The axiomatic approach to systems complexity', *in* 'Designing for system maturity'. Pergamon Infotech State of the Art Report (Pergamon Press, 1985)
38 BASILI, V.R., and PERRICONE, B.T.: 'Software errors and complexity: an empirical investigation', *Communications of ACM*, 1983, **27**, (1), pp. 42 – 52
39 BASILI, V.R., SELBY, R.W., and PHILLIPS, T.Y.: 'Metric analysis and data validation across Fortran projects', *IEEE Transactions on Software Engineering*, 1983, **SE-9**, (6), pp. 652 – 663
40 BOWEN, J.: 'Are current approaches sufficient for measuring software quality?'. Proceedings of Software Quality Assurance Workshop, 1978, pp. 148 – 155
41 CURTIS, B. *et al.*: 'Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics', *IEEE Transactions on Software Engineering*, 1979, **SE-5**, (2), pp. 96 – 104
42 CURTIS, B., SHEPPARD, S.B., and MILLIMAN, P.: 'Third time charm: stronger prediction of programmer performance by software complexity metrics'. Proceedings of Fourth IEEE International Conference on Software Engineering, New York, NY, USA, 1979
43 DAVIS, J.S.: 'Chunks: a basis for complexity measurement', *Information Processing & Management*, 1984, **20**, (1 – 2), pp. 119 – 127
44 FEUER, A.R., and FOWLKES, E.B.: 'Some results from an empirical study of computer software'. Proceedings of Fourth IEEE International Conference on Software Engineering, Munich, West Germany, pp. 351 – 355
45 GAFFNEY, J.E.: 'Program control, complexity and productivity'. Proceedings of IEEE Workshop on Quantitative Software Models for Reliability, 1979, pp. 140 – 142
46 HENRY, S., KAFURA, D., and HARRIS, K.: 'On the relationship among three software metrics'. ACM SIGMETRICS Performance Evaluation Review, Spring 1981, pp. 81 – 88
47 KITCHENHAM, B.A.: 'Measures of programming complexity', *ICL Technical Journal*, 1981, **2**, (3), pp. 298-316
48 PAIGE, M.: 'A metric for software test planning'. Proceedings of COMPSAC 80 Conference, Buffalo, NY, USA, Oct. 1980, pp. 499 – 504
49 SCHNEIDERMAN, N.F.: 'An experiment in software error data collection and analysis', *IEEE Transactions on Software Engineering*, 1979, **SE-5**, (3), pp. 276 – 286
50 SHEN, V.Y., YU, T.-J., THEBAUT, S.M., and PAULSEN, L.R.: 'Identifying error-prone software — an empirical study', *IEEE Transactions on Software Engineering*, 1985, **SE-11**, (4), pp. 317 – 323
51 SHEPPARD, S.B., CURTIS, B., MILLIMAN, P., BORST, M.A., and LOVE, T.: 'First-year results from a research program on human factors in software engineering'. National Computer Conference, New York, NY, USA, Jun. 1979, AFIPS Conference Proceedings Vol. 48, pp. 1021–1027
52 SUNOHARA, T., TAKANO, A., UEHARA, K., and OHKAWA, T.: 'Program complexity measure for software development management'. Proceedings of Fifth IEEE International Conference on Software Engineering, San Diego, CA, USA, March 1981, pp. 100–106
53 WANG, A.S., and DUNSMORE, H.E.: 'Back-to-front programming effort prediction', *Information Processing & Management*, 1984, **29**, (1–2), pp. 139–149
54 WOODFIELD, S.N., SHEN, V.Y., and DUNSMORE, H.E.: 'A study of several metrics for programming effort', *Journal of Systems & Software*, 1981, **2**, pp. 97–103
55 BASILI, V.R., and HUTCHENS, D.H.: 'An empirical study of a syntactic complexity family', *IEEE Transactions on Software Engineering*, 1983, **SE-9**, (6), pp. 664–672
56 HUMPHREYS, R.A.: 'Control flow as a measure of program complexity'. UK Alvey Programme Software Reliability and Metrics Club Newsletter 4, 1986, pp. 3–7
57 WHITTY, R.: 'Comments on "Control flow as a measure of program complexity"'. UK Alvey Programme Software Reliability and Metrics Club Newsletter 5, 1987, pp. 1–2
58 SCHNEIDER, G.M., SEDLMEYER, R.L., and KEARNEY, J.: 'On the complexity of measuring software complexity'. National Computer Conference, Chicago, IL, USA, May 1981, AFIPS Conference Proceedings Vol. 50, pp. 317–322
59 BROOKS, R.E.: 'Studying programmer behaviour experimentally: the problems of proper methodology', *Communications of ACM*, 1980, **23**, (4), pp. 207–213
60 STEVENS, W.P., MYERS, G.J., and CONSTANTINE, L.L.: 'Structured design', *IBM Systems Journal*, 1974, **13**, (2), pp. 115–129.
61 SAYWARD, F.G.: 'Experimental design methodologies in software science', *Information Processing & Management*, 1984, **20**, (1–2), pp. 223–227
62 FENTON, N.E., and WHITTY, R.W.: 'Axiomatic approach to software metrification through program decomposition', *Computer Journal*, 1986, **29**, (4), pp. 330–340
63 PRATHER, R.E.: 'On hierarchical software metrics', *Software Engineering Journal*, 1987, **2**, (2), pp. 62–65

M. Shepperd is with the School of Computing & Information Technology, Wolverhampton Polytechnic, Wolverhampton WV1 1LY, England, and is also with the Computing Discipline, Faculty of Mathematics, The Open University, Walton Hall, Milton Keynes MK7