An Axiomatic Theory of Software Complexity Measure

Ronald E. Prather
Department of Mathematics and Computer Science, University of Denver, Denver, Colorado, USA

In software engineering, various 'metrics' have been introduced in an attempt to measure the complexity of programs. We show how the whole idea of a 'software complexity measure' can be axiomatized in such a way as to include the more familiar concrete examples and to allow for new measures that might offer advantages not captured by those previously introduced. In particular, a new testing measure is introduced, based on the 'multiple-condition' test strategy. Comparisons are made between this new measure and the more traditional metrics. In addition, a more general theoretical study is initiated, showing the effect of the axiomatic development in relation to the treatment of program structuredness.

1. INTRODUCTION

The new and rapidly developing field of 'software metrics' seeks to measure the inherent complexity of large software systems with a view toward predicting the overall project cost and evaluating the quality and effectiveness of the design.

It is fairly certain that no one 'magic number' can serve as a measurement for all of the characteristics of software that might be considered important in this respect. Instead, one expects that different metrics will be needed in estimating a program's inherent psychological complexity, its readability, testability, maintainability, flexibility, etc. On the other hand, it would be nice to think that these separate measures might share some common characteristics that could serve to unite the field and point the way to future research.

With this in mind, we propose a new axiomatic theory of software complexity measure over the class of structured programs. We show that the traditional metrics of Halstead and McCabe indeed satisfy this system of axioms when they are suitably restricted to the structured programming constructs. We then describe a general method for extending certain measures satisfying our axiom scheme to the class of unstructured programs, i.e. we provide a uniform treatment of the 'psychological complexity' of the goto statement, yielding a universal abstract complexity theory grounded in the original axiomatic system.

In the course of this development, we also present a new testing complexity measure, based on the so-called 'multiple-condition' test strategy. It is shown that this new measure also satisfies our axiom scheme, and yet, offers several advantages over the traditional measures of Halstead, McCabe, etc., now widely in use. In turn, this shows that our scheme is sufficiently flexible as to allow for improvements and comparisons from one measure to another.

2. TWO STANDARD COMPLEXITY MEASURES

It has been said[1] that there must be, 'as many complexity measures as there are computer scientists'. Not all of these have caught on with the software engineering community, however. Nevertheless, their number serves to explain our attempt to axiomatize the whole notion of a complexity measure, hoping to provide for a more unified and systematic approach to the field. For our purposes, and without regard to their validity, it will suffice to briefly describe the two most often cited metrics—those of McCabe[2] and Halstead.[3] Although they do provide some basis for the scheme to be introduced shortly, we feel obliged to comment on certain inadequacies in these metrics in order that comparisons and recommendations for improvement can be made at a later point.

We view a program F as being represented by a 'skeleton' of its underlying flowchart, a directed graph symbolizing the flow of control. Accordingly, a *program* or *flowchart graph* $F = (V, E)$ is a directed graph with vertex set V and edge set E. It is further supposed that the graph has a single entry point, the *start* vertex ($\bigtriangledown$), and a single exit point, the *stop* vertex ($\bigtriangleup$), and that every vertex lies on a path from 'start' to 'stop'. The other vertices are further partitioned into *statement nodes* and *decision nodes*, the former having one exiting edge, the latter two (or more). The statement nodes are identified with (sequences of) simple statements (of assignment, input, and output) and the decision nodes are characterized by predicates, representing Boolean conditions causing a branch in the flow of control (with the two exiting edges labelled 'T' or 'F' accordingly).

Given a program graph $F = (V, E)$, McCabe[2] suggests the use of the graph's *circuit rank*:

$$\rho(F) = |E| - |V| + 1$$

as a measure of the program's complexity. It is well known that this number is closely related (and in fact, differs by one from) the number of linearly independent paths through the graph, and it has been suggested that the testing of a program over a corresponding 'basis' of program paths represents an adequate test of the program's performance. Furthermore, McCabe has observed that this same circuit rank ρ can be computed as the 'number ($\rho(F) = d$) of decision nodes' in the graph.

In spite of the fair degree of acceptance of the McCabe measure in software engineering circles, several criticisms can be made. Of course, it will be true that one can

CCC-0010-4620/84/0027-0340 $04.00

always expect to find a set of examples for which a given measure performs poorly, i.e. contrary to our intuition. More than this, however, we believe that there are three fundamental objections to McCabe's measure, incongruities that have not been given much attention elsewhere. We list these as follows:

1. ρ is relatively insensitive to program restructuring.
2. ρ correlates too closely with 'number of lines of code'.
3. ρ takes no account of program nesting levels.

The net effect, we feel, is that McCabe's metric is too coarse, too elementary to reflect the intricacies of program complexity, the many features standing in the way of our ability to understand the program or to test its correctness.

Regarding (1), it has been shown[4] that McCabe's metric may actually increase for our having restructured the code according to accepted structured programming precepts. Considering (2), we of course realize that the 'number of lines of (uncommented) code' is often used as a crude 'yardstick' of programming effort. But the close correlation reported[5] between this figure and McCabe's is hardly a validation of the McCabe theory. Intuitively, we feel that there must be 'more to it' than this. Particularly in reference to (3), it can be argued that the levels of nesting of a program segment should contribute to its complexity. But the two flowchart segments of Fig. 1 are equally complex from the standpoint of the McCabe measure. Surely it is easier to analyse the sequence of loops in (a). If I understand each of them in isolation, then I understand their sequential effect—in an *additive* fashion. On the other hand, the nesting of constructs (repetitions, selections, etc.) as in (b) may be expected to have more of a *multiplying* effect on the complexity, or at least, so it can be argued. The point is, we are looking for a broad-based axiomatic framework where such points of view can be explored and compared to the traditional approaches to complexity.

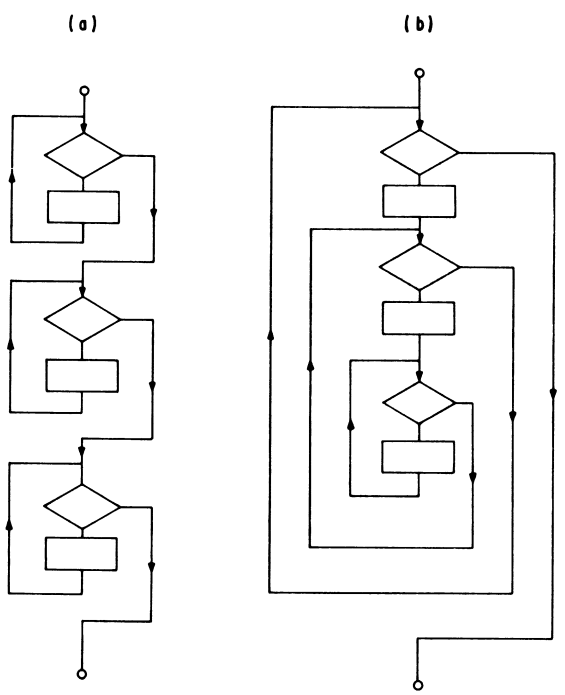

Figure 1

The second standard measure of program complexity is the 'software science' metric of Halstead.[3] If we count:

n = number of distinct operators and operands
N = number of operator and operand occurrences

in a program F, then the *program volume*

$$v(F) = N \log n$$

is taken as a measure of the complexity. It is ordinarily suggested that this expression represents the number of 'mental comparisons' necessary to implement the program, and that a comparable effort is necessary in understanding the program. At least, this is the justification usually given.

In fact, a number of empirical studies[6,7] have been offered in support of an apparent correlation of Halstead's measure with various counts of programming errors. On the other hand, it is easily seen that Halstead's metric is subject to the same criticisms previously cited for the case of McCabe's measure. The study previously noted[4] shows the ineffectiveness of Halstead's metric in relation to program structure, and clearly, the volume measure does not take a program's nesting levels into account. In fact, we could hardly expect it to be otherwise, since Halstead's analysis is 'linguistic' rather than 'structural' in nature.

But we remind the reader that our point in all of this, as mentioned above, is to seek a unified axiomatic framework for countering such objections as listed here, and for developing new metrics along more universal guidelines. This goal then becomes our central point of focus in the discussions to follow.

3. AXIOMATIC SOFTWARE COMPLEXITY MEASURE

As we have indicated, we first intend to present an axiomatic framework applicable to software complexity measure as it relates to the structured programming methodology. In Section 5, we then show how certain measures satisfying our axioms in this respect can be extended so as to be applicable to unstructured programs as well, thus transcending the initial limitations and allowing for a broader scope to the theory.

The tenets of the *structured programming* methodology are now well understood[8,9] and they need not be repeated here. For our purposes, it will suffice to say that a *structured program* is one that has been built up inductively from certain *simple statements* acting as a base set of *processes*, using only the three familiar constructs:

(a) *sequence*: begin S_1; S_2; . . .; S_n end
(b) *selection*: if P then S_1 else S_2
(c) *repetition*: while P do S

where the S_i (S, T respectively) are themselves structured (but possibly compound) processes and P is an arbitrary predicate in the programming language under consideration. Nore that a *process* is a program segment (subflowchart) having a single entry and a single exit edge. Among these are the assignment statements and input/output statements ordinarily considered to be 'simple'. Depending on the programming language under consideration, other statements might also be regarded as simple, but such details need not concern us here.

Let us now suppose that m is a function from the class of structured programs (in a given programming language) to the non-negative real numbers. Such a function will be called a *proper measure* (*of program complexity*) if it satisfies the three axiom schemes:

(a) $m(\text{begin } S_1; S_2; \ldots; S_n \text{ end}) \geq \sum m(S_i)$

(b) $2(m(S_1) + m(S_2)) \geq m(\text{if } P \text{ then } S_1 \text{ else } S_2)$
$> m(S_1) + m(S_2)$

(c) $2\,m(S) \geq m(\text{while } P \text{ do } S) > m(S)$

where we require the two left-hand inequalities (denoted (b′) and (c′) in the sequel) only for sufficiently large $m(S)$. Note the strict inequalities at the right, hereafter referred to as conditions (b) and (c) to distinguish them from the left-hand inequalities.

The reasons for our requiring the first inequality and the right-hand inequalities of (b) and (c) are not very profound; we ask little more than that 'the complexity of the whole be greater than (or equal to, in the case of (a)) the complexity of its parts'. But how much greater? That is the question we address in (b′) and (c′). Here, one may think of the predicate P as having an implicit complexity of its own. The two inequalities then insist that the complexity of the predicate be regarded as not having exceeded that of the body of the construct in question. If these are arguable positions to have taken, then the burden of proof must await the consequences of these assumptions (e.g. see Theorem 3 in Section 5). Beyond this, we must first test the adequacy of these requirements in relation to the concrete measures currently in use.

Verification that ρ is a proper measure

Using McCabe's simplified count, i.e. the number d of decision nodes, we have:

(a) $\rho(\text{begin } S_1; S_2; \ldots; S_n \text{ end})$
$= \sum d_i = \sum \rho(S_i)$

(b) $\rho(\text{if } P \text{ then } S_1 \text{ else } S_2)$
$= 1 + d_1 + d_2 > d_1 + d_2 = \rho(S_1) + \rho(S_2)$

(c) $\rho(\text{while } P \text{ do } S) = 1 + d > d = \rho(S)$

considering only the right-hand inequalities of (b) and (c). The inequalities (b′) and (c′) are seen to be satisfied whenever $\rho(S) \geq 1$, say, and that is sufficient for our purposes.

Verification that v is a proper measure

Considering the Halstead metric, $N \log n$, and using an obvious notation, we have:

(a) $v(\text{begin } S_1; S_2; \ldots; S_n \text{ end}) = N \log n$
$= \sum N_i \log n \geq \sum N_i \log n_i = \sum v(S_i)$

(b) $v(\text{if } P \text{ then } S_1 \text{ else } S_2)$
$= N \log n = (N_P + N_1 + N_2) \log n$
$\geq N_P \log n_P + N_1 \log n_1 + N_2 \log n_2$
$> N_1 \log n_1 + N_2 \log n_2$
$= v(S_1) + v(S_2)$

(c) $v(\text{while } P \text{ do } S) = N \log n = (N_P + N_S) \log n$
$\geq N_P \log n_P + N_S \log n_S$
$> N_S \log n_S = v(S)$

again concentrating on the right-hand inequalities of (b) and (c). In this case, the inequalities (b′) and (c′) may present something of a problem. It is, of course, possible that the Halstead complexity of the predicate P may exceed that of the body of the construct in question, whatever its measure. However, this is indeed an unusual occurrence, and we are entirely justified in thinking that this happens so rarely in practice as to regard the Halstead metric as a 'proper measure' in good standing.

As we have said, the main test of the soundness of our axioms will rest with the conclusions we are able to draw from them. For the time being, we offer the following result, stated without proof.

Theorem 1

Let $m_1, m_2, \ldots, m_n$ be proper measures of program complexity. Then any weighted linear combination of these measures is again a proper measure.

It is clear that this result is of definite interest in any attempt to obtain a composite complexity measure built up out of existing measures as components. On the other hand, the more substantive questions must be deferred until we provide the extended capabilities of Section 5.

4. A NEW TESTING COMPLEXITY MEASURE

The general *goal of software testing* is to affirm the quality of a program through systematic exercising of the code in a carefully controlled environment. It is hoped that the successive execution of a program according to a well designed test scheme will continually exhibit the proper behaviour, thus indicating the unlikelihood of programming errors. In fact, considering the difficulty in obtaining actual proofs of correctness, program testing has become possibly the only truly effective means for ensuring the quality of software systems of non-trivial complexity.

There are a number of testing strategies that can be used to ensure a reasonable level of confidence in a program's correctness. Primarily, these criteria are based on the objective of providing adequate 'coverage' of the flowchart graph by the totality of program paths that have been traversed by the testing scheme. Among the more commonly used criteria are the following:

1. *Statement coverage*. Execute all statements in the graph.
2. *Node coverage*. Encounter all decision node entry points in the graph.
3. *Branch coverage*. Encounter all exit branches of each decision node in the graph.
4. *Multiple condition coverage*. Achieve all possible combinations of condition outcomes at each decision node of the graph.
5. *Path coverage*. Traverse all paths in the graph.

It is clear that 'statement coverage' and 'node coverage' are in themselves rather weak testing strategies. The 'branch coverage' criterion, however, implies these two, and has come to be regarded as a minimal standard of testing achievement.

The stronger criterion of 'path coverage' is difficult to achieve in a program of reasonable complexity. The 'multiple condition coverage' criterion again ensures branch coverage, and moreover, provides for an extensive checking of a program's condition logic. Moreover, the

mechanics of the test case generation problem are somewhat easier to handle than are those for the path coverage criterion. Furthermore, the multiple condition coverage strategy leads to an effective measure of a program's testing complexity, as we now show.

As we would expect, this *new testing measure* μ will first be defined inductively over the simple statements of a language so as to constitute a proper measure on the class of structured programs. We must therefore first decide on a measure of complexity for the simple statements of a (structured) program. Rather than attempting to assess the relative complexity of one simple statement as compared with another, however, we agree instead to take the simple statement as our 'unit' of complexity, writing in effect:

$$\mu(\text{simple statement}) = 1$$

Of course, it is only a relative complexity that matters in comparing one program with another, so we are free to choose any 'scaling factor' we wish in developing our standard, and one may thus interpret the above convention as our having normalized all complexity measurements accordingly. It is our feeling, however, that this would be an appropriate place to superimpose a 'Halstead-like' metric, so as to differentiate the complexity of one simple statement from another, providing a fine-tuned accounting for the operators and operands involved. We mention this as a possibility for further study, but in the present context we will continue to operate with the unit measure given above.

In a *bottom-up* testing methodology, based on any testing strategy whatsoever, one assumes that certain processes have already been adequately tested and that their correctness is reasonably assured before they have been integrated into a larger process whose testing is under study. Our inductive definition of the testing measure μ is consistent with this philosophy. Thus, in considering (a) the sequence construct, we suppose that the measures $\mu(S_i)$ have already been assigned to processes $S_1, S_2, \ldots, S_n$, and we then seek to assign a measure to their sequential 'sum'. We take the view that μ should be additive over such sums, writing:

(a) $\mu(\text{begin } S_1, S_2, \ldots, S_n \text{ end}) = \sum \mu(S_i)$

the intuitive assumption being that 'If I understand each process separately, then I understand their sequence taken as a whole'. Moreover, we view this additivity formula as a recognition that *in some respects* a complexity measure should correlate with the crude measure of 'program length'. But we only expect a high degree of correlation in the case of programs that are, to a large extent, built up out of straight line code. As we will see, our measure performs quite differently with respect to a program's nested constructs.

Arguing by induction once more in treating (b) the selection construct, we must again assume that appropriate measures of complexity $\mu(S_1)$, $\mu(S_2)$ have already been assigned to the processes S_1 and S_2. In consideration of the multiple condition testing strategy, we then agree to assign:

(b) $\mu(\text{if } P \text{ then } S_1 \text{ else } S_2) = 2 \max(\mu(S_1), \mu(S_2))$

by taking a 'worst case' view of the situation, as though the two test cases needed to exercise the (supposedly simple) Boolean condition P will cause a traversal of the process (S_1 or S_2) of greatest complexity. (Note: we substitute the figure $2^{|P|}$ for 2 in the case where P is a complex Boolean condition composed of $|P|$ simple Boolean relations.) And we remark that an alternative approach would be to substitute the 'ave' for the 'max' function, in assuming that roughly half of the paths through P will traverse S_1, the other half S_2. In any case, we will have worked toward the objective of compounding the complexity of deeply nested processes.

Finally, in (c) the repetition construct, we again impose a multiplicative effect in writing:

(c) $\mu(\text{while P do S}) = 2\mu(\text{S})$

(Note: once more, we replace 2 by $2^{|P|}$ in the case where P is a compound Boolean condition.) In keeping with the multiple condition testing strategy, we wish to recognize the existence of all possible combinations of the simple conditions appearing in P, treating each combination as having specified a path through S. Since each such path entails the derivation of appropriate input test data and the test is then administered successively through P to the module S (though admittedly, some inputs may bypass S via the 'false' alternative), our multiplicative formulation would seem to be well founded.

We note the compounding effect of successive nesting levels with μ as opposed to McCabe's measure ρ. For the two loop structures of Fig. 1, we obtain by way of comparison:

(a)	(b)
$\rho = 3$	$\rho = 3$
$\mu = 2 + 2 + 2 = 6$	$\mu = 2(1 + 2(1 + 2(1))) = 14$

assuming simple statements and simple conditions throughout. For the measure ρ, the two programs are considered equally complex, whereas for μ, the statements at a deeper level of nesting weigh more heavily in the ultimate determination of program complexity. Moreover, in the case of McCabe's metric, there is no provision for weighing the complexity of the straight line code that might be represented by the three repeated process of Fig. 1. Sixteen assignment statements would count the same as one. With the measure μ, however, their effect would at least be additive, and moreover, we have the additional opportunity of imposing a Halstead-like fine tuning of the measure of the individual simple statements, as mentioned earlier.

Before proceeding further, we should check to see that our new testing measure satisfies the axiom scheme of Section 3, as this will be important in the sequel.

Verification that μ is a proper measure

Reviewing the axioms, and comparing with (a), (b), (c) above, we may compute:

(a) $\mu(\text{begin } S_1; S_2; \ldots; S_n \text{ end}) = \sum \mu(S_i)$

(b) $\mu(\text{if } P \text{ then } S_1 \text{ else } S_2)$
$= 2 \max(\mu(S_1), \mu(S_2)) > 2 \text{ave}(\mu(S_1), \mu(S_2))$
$= \mu(S_1) + \mu(S_2)$

(c) $\mu(\text{while } P \text{ do } S) = 2\mu(\text{S}) > \mu(\text{S})$

as required. In this case, we see that the inequalities (b'), (c') are trivially satisfied.

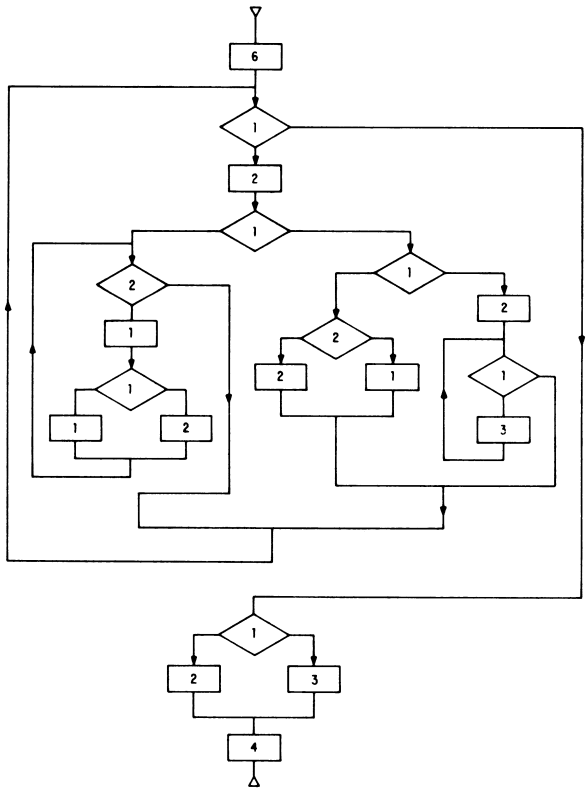

Figure 2

Example

To illustrate the more intricate computations with our new testing complexity measure, consider the (structured) flowchart of Fig. 2. Here, the notation n written in a square denotes n sequential simple statements, and similarly, the notation m written in a diamond indicates a Boolean condition composed of m simple Boolean relations. The inner selection statement has two branches, of complexities:

$4(1 + 2 \max(1, 2)) = 20$ (the left branch)

$2 \max(4 \max(2, 1), 2 + 2(3)) = 16$ (the right branch)

respectively. Finally, we compute the complexity of the flowchart as a whole:

$$\mu = 6 + 2(2 + 40) + 6 + 4 = 100$$

thus completing the example.

5. THE PSYCHOLOGICAL COMPLEXITY OF THE GOTO STATEMENT

Given the emphasis of software development management on quality, testability, maintainability, etc. and considering all of the literature over the last decade extolling the virtues of structured programming in this regard, one would think that only structured programs would be written nowadays. Sadly, this is not yet the case, and we must still acknowledge the fact that some programs are ill conceived, poorly structured and therefore difficult to test or maintain. Considering this state of affairs, a viable program complexity measure must allow for unstructured programs, and ideally, should treat 'structuredness' as one of the features that it attempts to measure, so that a well structured rewriting of a program might be reflected in a comparison of measures.

In a modern programming language (e.g., Pascal, C, etc.) a *goto* statement may be an available though unnecessary complement to the usual range of programming constructs considered earlier. In such a setting, the status of the goto statement is on the same level as a simple statement. That is, syntactically at least, one is allowed to write 'goto label' wherever a simple statement could have appeared. Thus (as if in Pascal), we may write for example:

```
      begin
      goto L;
M:    S;
L:    if P then
        goto M
      end
```

in order to effect an unnatural implementation of a 'while loop', as shown in Fig. 3(a). Redrawing the flowchart to reflect the illusion of structuredness exhibited in the code, we have Fig. 3(b), where a flowchart circular symbol g has been introduced to simulate the goto statement. More precisely, circular symbols g, h are paired as a means of reflecting the discontinuity in program flow—from g to h.

Arguing in the abstract, we think of g (or better $m(g)$) as an 'indeterminate' to be used in algebraic expressions for program complexity, using a measure m. From this point of view, we can compute expressions (involving g) for the complexity of unstructured flowcharts just as before—thinking of each g as a simple statement and disregarding the accompanying hs. But the question then naturally arises: What complexity figure $m(g)$ should be associated with the gs? We offer several suggestions in this regard.

As a first approach, one might try to 'calibrate' g by comparing a number of standard poorly structured

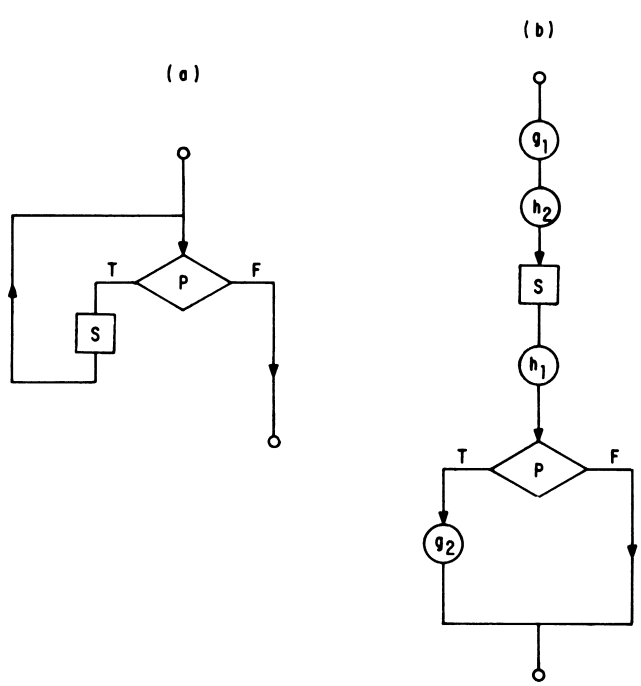

Figure 3

programs to their (standard) restructured equivalents, computing their measures for both versions in each case. The results could be used (equating and solving for g in each case) to arrive at a *nominal complexity figure* g', to be used in measuring each goto found in practice. In fact, some program managers could assign a figure somewhat in excess of this nominal value with a view toward discouraging any use of gotos whatsoever. As an additional refinement to this technique, 'upstream' gotos could be given an additional penalty (say twice the usual value), since it is well known that these cause the most difficulty in practice.

Notwithstanding this simple refinement, it must still be argued that 'all gotos are not created equally'—some are worse than others. Ideally, we would like to measure the complexity of one goto as opposed to another in some meaningful way. In our attempt to do this (and throughout this section generally), we are of course thinking of an arbitrary proper measure m, and thus, we are departing from the testing philosophy that governed our thinking in the preceding phase of our study.

We suppose then that m is a proper measure in the sense of Section 3. More specifically, such a measure will be said to be *inductive* if it can be defined inductively over the class of simple statements (to the structured programs) of the language. In this connection, the following result is of interest:

Theorem 2

The measures ρ and μ are inductive (but ν is not).

Proof. For McCabe's measure ρ, we have

$$\rho(\text{simple statement}) = 0$$

and the inductive definitions:

(a) $\rho(\text{begin } S_1; S_2; \ldots; S_n \text{ end}) = \sum \rho(S_i)$
(b) $\rho(\text{if } P \text{ then } S_1 \text{ else } S_2) = 1 + \rho(S_1) + \rho(S_2)$
(c) $\rho(\text{while } P \text{ do } S) = 1 + \rho(S)$

whereas for our new testing measure μ, the definitions of Section 4 are already inductive, thus completing the proof.

As is generally agreed, the difficulty with the goto statement is that its effects cannot be localized. We cannot understand its contribution to a program's complexity without surveying the broader segment of the program in which the goto g (and its accompanying *target* h) are embedded. It is as if the level of complexity of the code surrounding (g, h) acts as a barrier to our understanding the effect of g itself. And it is in this spirit that we offer a more comprehensive treatment of the complexity of a goto g, viewed as an extended axiom in our writing:

(d) $$m(g) = \max_{\substack{A \subseteq X = [g,h] \\ \{g,h\} \cap A = \theta}} m(A)$$

where X is the subflowchart 'spanned' by g and h, and we take the maximum complexity of all maximal subflowcharts[10] A, disjoint from g, h but contained within X. We feel that this approach serves to distinguish one goto from another, generally giving more weight to those with remote targets.

We will see that the consequence of axiom (d) is to enforce a penalty for the use of the goto statement, one

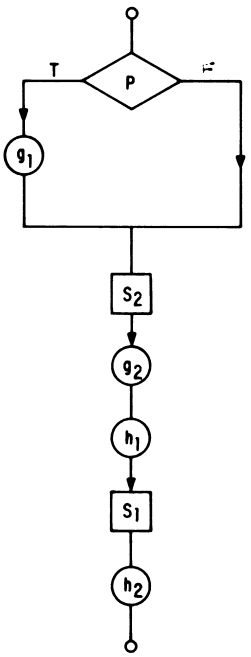

Figure 4

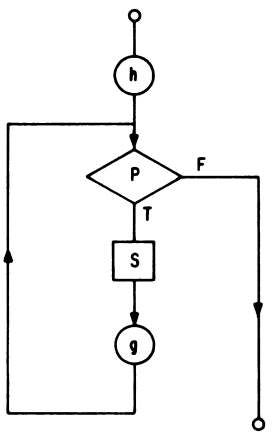

Figure 5

that is sufficient to render a restructured version of poorly conceived code to be of lesser complexity, at least in the typical situations that one is likely to encounter in practice.

Suppose we are reviewing programs written in a modern programming language (e.g. Pascal, C, etc.), but we are dealing with programmers who are inexperienced in such languages, having previously coded only in FORTRAN or BASIC, say. Such programmers are likely to follow their old habits, 'wiring up' if–then–else and while–do constructs using goto statements, as indicated in Figs 4 and 5, respectively. Our fundamental result shows that such a practice is uniformly discouraged, as long as we are measuring complexity as outlined above.

Theorem 3

Let m be an inductive measure of program complexity, extended (to treat the goto statement) as in axiom (d) above. Then if $F(S_i)$ is any structured programming construct (i.e. if–then–else, while–do, etc.) and if F' is its unstructured goto implementation, then we have:

$$m(F) < m(F')$$

for all S_i of sufficiently large measure.

Proof. It will suffice to indicate the technique of proof in the typical situations exhibited in Figs 4 and 5. For Fig. 4 (an unstructured if–then–else), we have:

$$\begin{aligned} &m(\text{if } P \text{ then } S_1 \text{ else } S_2)' \\ &\quad = m(\text{begin if } P \text{ then } g_1; S_2; g_2; S_1 \text{ end}) \\ &\quad \geq m(\text{if } P \text{ then } g_1) + m(S_2) + m(g_2) + m(S_1) \\ &\quad > m(g_1) + m(S_2) + m(g_2) + m(S_1) \\ &\quad = m(S_2) + m(S_2) + m(S_1) + m(S_1) \\ &\quad = 2(m(S_1) + m(S_2)) \geq m(\text{if } P \text{ then } S_1 \text{ else } S_2) \end{aligned}$$

using axioms (a), (b), (d) and (b′).

Similarly, for Fig. 5 (the while–do, where the programmer has not understood that the 'while' statement automatically loops back to the test P), we have:

$$\begin{aligned} &m(\text{while } P \text{ do } S)' \\ &\quad = m(\text{while } P \text{ do begin } S; g \text{ end}) \\ &\quad > m(\text{begin } S; g \text{ end}) \\ &\quad \geq m(S) + m(g) \\ &\quad = m(S) + m(S) = 2m(S) \geq m(\text{while } P \text{ do } S) \end{aligned}$$

using axioms (a), (c), (d) and (c').

Note that it was necessary to use all of our axioms in order to verify only these two instances of restructuring. On the other hand, a more extensive survey of the common programming mistakes of this kind shows that the penalty remains in force. For example, a more obtuse rendering of the while–do construct is that shown earlier in Fig. 3. Nevertheless, in this case, we again have:

$$m(\text{while } P \text{ do } S)' > m(\text{while } P \text{ do } S)$$

as the reader may easily check. It is hoped that this survey, although falling short of a complete proof, will be sufficient to convince the reader of the validity of our claim.

Having considered these few examples, we have good reason to expect that a general extended measure m (in the sense of axiom (d) above), will adequately reflect an improvement in program structure through elimination of gotos. On the other hand, our formulation of the weighting of goto statements presents a few computational difficulties. First of all, in reference to axiom (d), it must be mentioned that the effective determination of the spanning subflowchart X and its maximal subflowcharts A, though certainly feasible,[10] represents a rather elaborate algorithm in and of itself. More serious perhaps is the observation that the figures $m(A)$ may not be readily available, owing to the fact that the subflowcharts A may themselves involve goto statements! It may even happen that a number of gotos g are so 'tightly coupled' that, in reference to axiom (d), it seems that we cannot compute any one value $m(g)$ without knowing all of the others. It is this last point that we now wish to discuss.

In returning to the idea that each goto statement g appearing in a program is an indeterminate (and writing g rather than $m(g)$ in axiom (d)), a closer look at our earlier formulation leads to a system of linear equations with the gs as unknowns:

$$\begin{aligned} g_1 - a[1,2]g_2 - \cdots - a[1,n]g_n &= b_1 \\ -a[2,1]g_1 + g_2 - \cdots - a[2,n]g_n &= b_2 \\ \cdots & \\ -a[n,1]g_1 - a[n,2]g_2 - \cdots + g_n &= b_n \end{aligned}$$

if we simply make the replacement $g = \text{ave } A$ in (d). On the other hand, we would prefer to retain the 'max' rather than the 'ave', if only to maximize the penalty for using gotos. We still obtain a system of equations (non-linear, however) for the gs as unknowns. (Note: the use of the average rather than the maximum perhaps gives a better statistical accounting for the psychological barrier represented by the goto, and in addition, provides for an easier algebraic analysis of the complexity equations. One then has the option of solving the system of equations by ordinary means.) But instead, we take a pragmatic, if less conventional approach.

We suggest that the 'max' be retained in axiom (d) and that the nominal complexity figure g' be substituted for all gs appearing on the right-hand side of our system of equations. One may then solve for the gs immediately, and we propose that these values be taken as representative of the complexity of the various goto statements. In effect, we have suggested the computation of one Gauss–Seidel-like iteration,[11] using the initial guess $g = g'$.

Example

Consider the (unstructured) flowchart of Fig. 6, and suppose we are using the new (but extended) testing complexity measure μ, as introduced in Section 4. (In the absence of a known empirical value for g' relative to the testing measure μ, we suggest that a unit nominal value $g' = 1$ be used, though we have reason to expect that an experimental determination would yield a $g' \gg 1$.) The reader should recall the meanings of n in a square and m in a diamond, as in our earlier example. One then computes:

$$\begin{aligned} g_1 &= \max(1, 1, 1, 2\max(3, 4), 2 + 2(1 + g_2)) \\ &= \max(1, 1, 1, 8, 6) = 8 \\ g_2 &= \max(1, 2, 2\max(3, 4), 2(1 + 2\max(1, 1 + g_1))) \\ &= \max(1, 2, 8, 10) = 10 \end{aligned}$$

substituting $g = g' = 1$ on the right, as explained above. With these results for the goto complexities, we then compute:

$$\begin{aligned} \mu &= 2 + 2\max(2(1 + 2\max(1, 1 + 8)), \\ &\qquad 2\max(2\max(3, 4), 2 + 2(1 + 10))) + 2 \\ &= 2 + 2\max(38, 48) + 2 = 100 \end{aligned}$$

for the flowchart as a whole. Of course, it must be remembered that our hand computation obscures the difficulty in identifying the required subflowcharts X, A appearing in axiom (d). Nevertheless, it is clear that our treatment of the resulting system of equations does provide a convenient, practical, and comprehensive solution to the goto complexity issue.

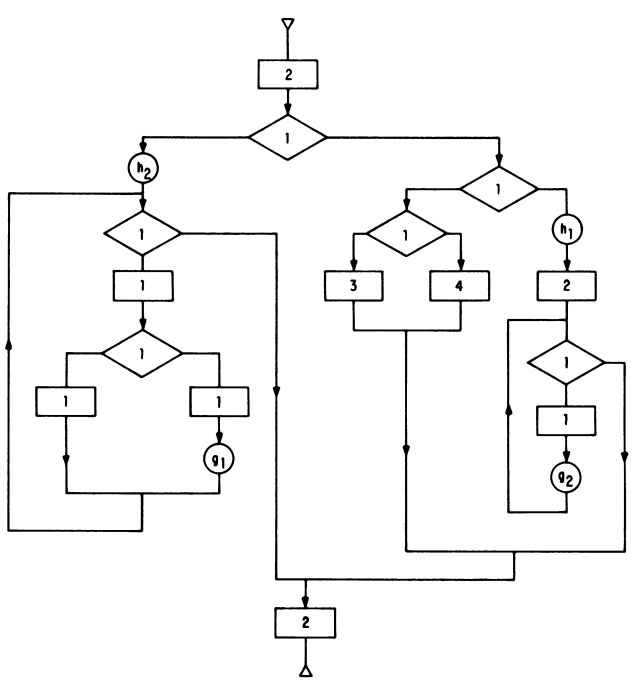

Figure 6

6. CONCLUDING DISCUSSION

With program complexity measures, it is traditional that a *maximum module complexity* figure be suggested for use as a 'yardstick' for deciding that a program is 'too big' and should be split up into separate procedures. Thus the figures:

$$\rho = 10$$
$$\eta = 50$$

are generally accepted[12] as appropriate for use with McCabe's measure ρ and with η, the 'number of lines of code' measure. (Note that η also satisfies our axioms.) Our experience suggests that a corresponding figure:

$$\mu = 100$$

be used with the new testing complexity measure developed here. Note, however, that a module of McCabe complexity $\rho = 10$ say, can vary in μ measure by several orders of magnitude, depending on whether we are basically dealing with straight line code or with deeply nested code. Again, this demonstrates the failure of McCabe's measure to distinguish between these two very different situations. On the other hand, our new testing measure, when restricted to programs of size 100 (the suggested maximum module complexity figure) might be said, generally, to have established a natural threshold at 'flowcharts that fit on a page', so to speak, if we can judge by Figs 2 and 6. And yet, depth of nesting, use of well-structuredness, etc., will significantly influence the measure in specific instances.

One of the main features of the particular measure we have introduced is its relevance to an established testing methodology. Hopefully, further experimental verification will confirm the apparent intuitive advantages of this particular measure. On the other hand, it must be remembered that this measure is only one of many that might fit into the axiomatic framework we have provided here. And it is further hoped that this framework might serve as a basis for other investigations into software complexity measure, both in the abstract and in the concrete realm of actual software metrics.

REFERENCES

1. B. Curtis, In search of software complexity, *Workshop on Quantitative Software Models*, IEEE, New York (1979).
2. T. J. McCabe, A complexity measure, *IEEE Trans. Software Engineering* **SE-2**(4), 308–320 (1976).
3. M. H. Halstead, *Elements of Software Science*, Elsevier, North Holland (1977).
4. A. I. Baker and S. H. Zweben, A comparison of measures of control flow complexity, *IEEE Trans. Software Engineering* **SE-6**(6), 506–512 (1980).
5. M. R. Paige, A metric for software test planning, *Proc. Software and Applications Conf.*, IEEE, New York (1980).
6. T. Sunohara, A. Takano, K. Vehara and T. Ohkawa, Program complexity measure for software development management, *5th Intl. Conf. on Software Engineeing*, IEEE, New York (1981).
7. B. Curtis, S. B. Sheppard, P. Millman, M. A. Borst and T. Love, Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics, *IEEE Trans. Software Engineering* **SE-5**(2), 96–104 (1979).
8. R. C. Linger, H. O. Mills and B. I. Witt, *Structured Programming*, Addison-Wesley, Reading, Mass. (1979).
9. O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, New York (1972).
10. R. E. Prather and S. G. Giulieri, Decomposition of flowchart schemata, *The Computer Journal* **24**(3), 258–262 (1981).
11. V. N. Faddeeva, *Computational Methods for Linear Algebra*, Dover Publishing Co., London (1959).
12. C. L. McClure and J. Martin, *Maintenance of Computer Programs*, Prentice Hall, Englewood Cliffs, N.J. (1982).

Received December 1982