A measure of program nesting complexity

by ELDON Y. LI
California Polytechnic State University
San Luis Obispo, California

ABSTRACT

For more than a decade, metrics of software complexity has been an intriguing topic for discussion. Many metrics have been proposed. Among them, the cyclomatic complexity metric is the easiest to understand and compute. In this paper, the cyclomatic complexity metric and its extensions are reviewed. The strengths and weaknesses of the cyclomatic metric are identified. One of the major weaknesses of the cyclomatic metric as well as its extensions is that they are insensitive to the level of nesting within various constructs. To remove this shortcoming, a "nesting" complexity metric is proposed. The process of deriving this new metric is described in this paper. This new metric is proved to be superior to the cyclomatic metric in reflecting program complexity.

INTRODUCTION

Since the emergence of structured programming concepts, program complexity has received tremendous attention from researchers in software engineering. "Program complexity" may be classified into two categories: computational complexity and psychological complexity.[1] Computational complexity refers to the difficulty of deriving expected output and of verifying an algorithm's correctness, and psychological complexity refers to the characteristics of software which make it difficult to understand and work with. Both types of complexity are not easily measured or described, and are often ignored during the system planning process. "But when this complexity exceeds certain unknown limits, frustration ensues. Computer programs capsize under their own logical weight, or become so crippled that maintenance is precarious and modification is impossible."[2] Based on Mills's observation, it seems wise to apply the "divide-and-conquer" principle to program design by decomposing the entire program into modules and submodules. Each module and submodule will have much less complexity and will, in turn, be much easier for programmers and users to comprehend and maintain.

Numerous metrics have been proposed to measure program complexity. Excellent reviews of these measures are provided by Fitzsimmons and Love,[3] Mohanty,[4] and Berlinger.[5] Several empirical studies have applied some selected metrics to measure program complexity and correlate such complexity with the number of errors occurring in the measured modules. It was found that the occurrence of program errors correlates significantly with the complexity of the target program.[3,6,7,8,9,10,11,12,13,14] This finding supports the popular hypothesis that program complexity is a major factor influencing the quality of computer programming.

Among various current complexity measures, the cyclomatic metric[15] is the easiest to understand and calculate. It is also the only one that lends itself to determining a minimum test set for program testing. In this paper, we review the cyclomatic metric and its extensions. The strengths and weaknesses of cyclomatic metric is identified as well. Further, a new metric to reflect the levels of nesting is proposed.

THE CYCLOMATIC COMPLEXITY METRIC

The cyclomatic complexity metric was proposed by McCabe.[15] His metric is based on the decision structure of a program and the cyclomatic number[16] (also called the cycle rank,[17] or the nullity[18]) of the classical graph theory. The cyclomatic complexity metric, $V(G)$, as defined by McCabe, is

$$V(G) = E - N + 2P$$

where E is the number of edges (or arcs), N is the number of vertices (or nodes), and P is the number of connected components. A component is a subgraph representing an external module that either is calling or is being called by another module. For example, consider a main program M and two called subroutines A and B having a control structure shown in Figure 1.

The total graph in Figure 1 is said to have three connected components and each subgraph has only one connected component (itself). Therefore, the cyclomatic complexity numbers are:

$$V(M) = 3 - 4 + 2(1) = 1,$$
$$V(A) = 2 - 2 + 2(1) = 2,$$
$$V(B) = 4 - 4 + 2(1) = 2,$$

and

$$V(M + A + B) = 9 - 10 + 2(3) = 5.$$

It can be easily shown that $V(M + A + B) = V(M) + V(A) + V(B)$.

McCabe further demonstrates two alternate ways of finding the complexity number V. One is to count the number of both inner and outer regions on the plane control graph. Notice there should be one outer region for each subgraph. In fact, if we form a closed subgraph by drawing an imaginary arc from the exit node to the entry node for each subgraph in Figure 1, and count all the inner regions afterward, we would yield the same number. We believe that the latter approach is less confusing than the former. For example, Figure 2 shows the closed subgraphs derived from Figure 1. By counting the inner regions (I_1 through I_5), we get a $V(G)$ of 5.

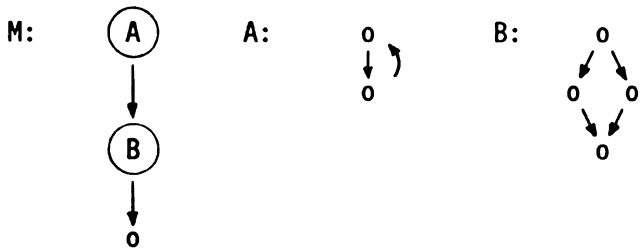

Figure 1—A graph with three connected components

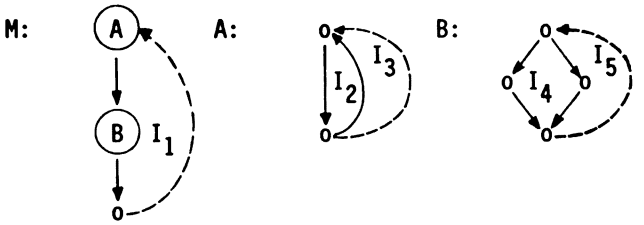

Figure 2—A graph with three closed subgraphs

The other way of calculating V is to count the number of predicate conditions in the program. Then the cyclomatic complexity is:

$V(G)$ = Number of predicate conditions + 1.

The attractive aspect of this method is that one can find the $V(G)$ directly from the program text without arduously constructing a flow graph. For example, consider the following PL/1 program:[19]

```
M: PROCEDURE(A,B,X);
IF ((A > 1) & (B = 0)) THEN DO;
                              X = X/A;
                            END;
IF ((A = 2) | (X > 1)) THEN DO;
                              X = X + 1;
                            END;
END;
```

Notice that each "IF" statement in procedure M has two conditions in its predicate. This type of "IF" statement is called a *compound* "IF" construct. In contrast, an "IF" statement with only one condition is called a *simple* "IF" construct, hereafter. Since each condition in procedure M contributes one cyclomatic complexity count, the complexity number is thus $V(M) = 4 + 1 = 5$.

Figure 3(a) shows that the flow graph corresponds to procedure M. Notice that it reflects the compound predicate by placing an extra exit edge for the second condition on each alternation node. For the convenience of counting, we substitute a traditional decision symbol for each alternation node and create Figure 3(b). It can be seen that Figure 3(b) is more readable and understandable than Figure 3(a). Therefore, we highly recommend adopting a decision symbol in flow-graph construction because it not only helps in counting the number of predicates but it also improves substantially the readability of the flow graph.

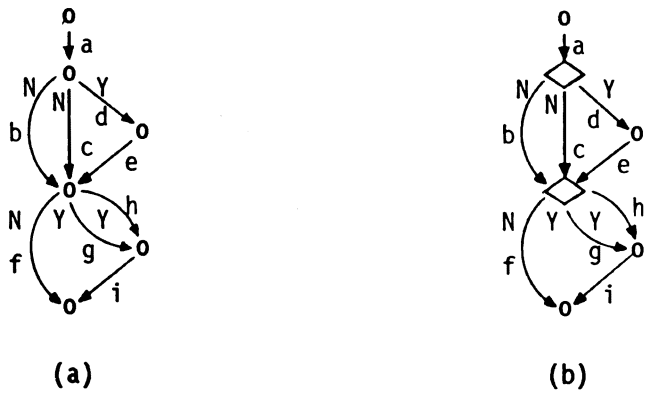

Figure 3—A control graph with compound predicates

THE ANOMALY AND THE EXTENSIONS OF THE CYCLOMATIC COMPLEXITY METRIC

One of the anomalies of cyclomatic complexity measure is that it does not accurately reflect the complexity of various "IF" structures; namely, simple "IF," compound "IF," and nested "IF." Myers[20] recommends an interval measure having one plus predicate counts as the lower bound, and one plus condition counts as the upper bound for the complexity level. Myers clearly demonstrates that this new metric can accurately reflect the complexity of various "IF" structures. However, the measure does not lend itself to quantitative analysis due to its "interval" data representation.

Hansen[21] indicates that the cyclomatic complexity metric does not reflect "expression" complexity. In other words, "a program with more operators is simply bigger . . . (and) . . . more complex" and thus [has a] higher expression complexity.[21] He proposes two measures in a pair to measure both control flow complexity and expression complexity. The former is measured by one plus predicate counts (including repetitive construct), the latter operator counts in the program. However, Hansen's metric suffers the same deficiency as Myers's; that is, it does not lend itself to quantitative analysis due to its "interval" data representation. Moreover, it is somewhat difficult to compute and can be applied only to program text.

Another major weakness of the cyclomatic complexity metric is its insensitivity to the level of nesting within various constructs. For example, three "WHILE" loops in succession result in metric values similar to those for three nested "WHILE" loops. This anomaly was brought forward by Curtis, Sheppard, Milliman, Borst, and Love,[1] but they did not offer any solution to it. Inspired by this anomaly, we examine various structures and propose a new metric to accurately reflect their complexity levels.

STRENGTHS AND WEAKNESSES OF THE CYCLOMATIC COMPLEXITY METRIC

Although the cyclomatic complexity measure has many anomalies, it has several strengths. We summarize its strengths and weaknesses in this section.

Strengths

1. It is easy to compute from the program text and the flow graph.
2. It supports a top-down development process to control module complexity in the design phase, that is, before actual coding takes place.
3. It lends itself to determining the maximum set of independent test paths.
4. It can be used to control the complexity of program modules. (McCabe recommends that an upper bound of 10 should be used as a guide to control the complexity of program modules. This recommendation is endorsed by Schneidewind and Hoffmann[13] and Walsh.[22])
5. It can be used to evaluate alternate program design to find the simplest possible program structure.

6. It serves to partition a program structure into high or low error occurrence according to its value.
7. It serves to partition a program structure into high or low error finding and removing times according to its value.
8. It can be used as a guide for allocating testing resources.

Weaknesses

1. It measures the psychological complexity, not the computational complexity.
2. It views all predicates as contributing the same amount of complexity.
3. It is insensitive to the level of nesting within various constructs.
4. It is insensitive to the frequency and the types of input and output activity.
5. It is insensitive to the size of purely sequential programs.
6. It is insensitive to the number of variables in the program.
7. It is insensitive to the intensiveness of data operations (i.e., the number of operators and operands) in the program.
8. It is insensitive to the dependency of control flows on foregoing data operations. (See, for example, the program listed on page 43 of Myers.[19])
9. It is insensitive to a situation in which one condition is "masked" or "blocked" by another within a nesting construct. (See, for example, the program listed on page 43 of Myers.[19])
10. It is insensitive to the program style and the use of "GOTO" statements.
11. It measures neither the types and levels of module interaction, nor the levels of module invocation.

DERIVATION OF THE NESTING COMPLEXITY METRIC

The purpose of our new complexity metric is to reflect the level of nesting within various constructs while keeping the computation process as easy as possible. Bearing these two objectives in mind, a new metric called "nesting complexity metric," $L(G)$, is formulated.

Consider the six structured programming control flow constructs[23] depicted in Figure 4. The "sequence" construct has a complexity V of unity while the "IF," the "WHILE," and the "UNTIL" constructs each has a V of 2, but the "CASE" construct of n branches has a V equal to $n-1$ nested "IF" statements. That is, a "CASE" statement with two branches is equivalent to a simple "IF" statement. The relationship of the complexities of various constructs is thus:

$$\text{sequence} < (\text{simple IF}) = (\text{simple WHILE}) = (\text{simple UNTIL}) = (\text{two-branch CASE}).$$

This relationship, along with our belief that nesting increases program complexity, are the premises of our metric to be derived subsequently.

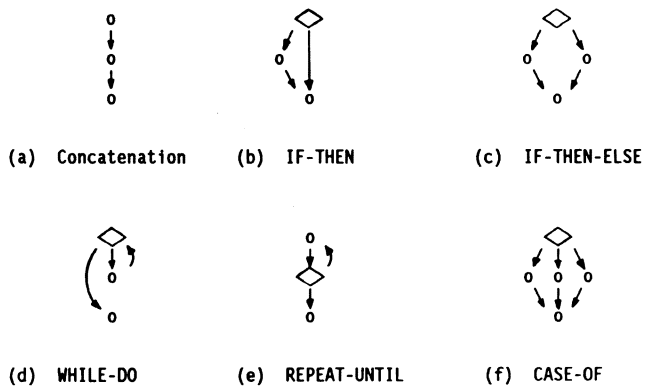

Figure 4—Structured programming control flow constructs

Now, consider the following structured programming statements:

```
A: IF (X = 0) THEN a
             ELSE b
B: IF (X = 0) AND (Y = 0) THEN a
                       ELSE b
C: IF (X = 0) THEN IF (Y = 0) THEN a
                              ELSE b
D: IF (X = 0) AND (Y = 0) AND (Z = 0) THEN a
                                      ELSE b
E: IF (X = 0) THEN IF (Y = 0) THEN
   IF (Z = 0) THEN a
              ELSE b
F: WHILE (X = 0) DO a
G: WHILE (X = 0) AND (Y = 0) DO a
H: WHILE (X = 0) DO WHILE (Y = 0) DO a
I: WHILE (X = 0) AND (Y = 0) AND (Z = 0) DO a
J: WHILE (X = 0) DO WHILE (Y = 0) DO WHILE
   (Z = 0) DO a
K: REPEAT a UNTIL (X = 0)
L: REPEAT a UNTIL (X = 0) AND (Y = 0)
M: REPEAT REPEAT a UNTIL (Y = 0) UNTIL (X = 0)
N: REPEAT a UNTIL (X = 0) AND (Y = 0)
   AND (Z = 0)
O: REPEAT REPEAT REPEAT a UNTIL (X = 0)
   UNTIL (Y = 0) UNTIL (Z = 0)
P: CASE X OF
     0 : a
Q: CASE X OF
     0 : CASE Y OF
           0 : a
R: CASE X OF
     0 : CASE Y OF
           0 : CASE Z OF
                 0 : a
S: IF (X = 0) THEN a
   IF (X = 1) THEN b
T: IF (X = 0) THEN a
   IF (X = 1) THEN b
   IF (X = 2) THEN c
U: IF (X = 0) THEN a
              ELSE IF (X = 1) THEN b
```

```
V: IF (X = 0) THEN a
   ELSE IF (X = 1) THEN b
                   ELSE IF (X = 2) THEN c
W: CASE X OF
     0 : a
     1 : b
X: CASE X OF
     0 : a
     1 : b
     2 : c
Y: CASE X OF
     0 : a
     1 : b
     ELSE : c
Z: CASE X OF
     0 : a
     1 : b
     2 : c
     ELSE : d
```

Based on the foregoing premise, we begin ranking the complexity of the constructs one pair at a time. Finally, the following complexity ordering is derived:

$A = F = K = P,$
$B = G = L,$
$C = H = M,$
$D = I = N,$
$C = D,$
$E = J = 0,$
$H = I,$
$M = N,$
$Q = C,$
$R = E,$
$S = W = Y,$
$T = X = Z,$
$U = C,$
$V = E,$

and

$A < B < D,$
$B < C,$
$D < E,$
$C < E,$
$F < G < I,$
$G < H,$
$I < J,$
$H < J,$
$K < L < N,$
$L < M,$
$N < O,$
$M < O,$
$P < Q < R,$
$S < T,$
$U < V,$
$S < U,$
$T < V,$
$W < X,$
$Y < Z,$
$P < \mathrm{W},$
$P < Y,$
$W < Q,$
$Y < Q,$
$X < R,$
$Z < R.$

Therefore, the final relationship is

$$\{A,F,K,P\} < \{B,G,L,S,W,Y\} < \{C,D,H,I,M,N,Q,T,U,X,Z\} < \{E,J,O,R,V\}.$$

In contrast, the relationship from McCabe's cyclomatic metric is

$$\{A,F,K,P\} < \{B,C,G,H,L,M,Q,S,U,W,Y\} < \{D,E,I,J,N,O,R,T,V,X,Z\},$$

which does not reflect the proper complexity ordering depicted above.

After comparing both relationships illustrated above, ten constructs—*C, E, H, J, M, O, Q, R, U, V,* which were ranked differently by the authors and McCabe's metric—are identified. All of these constructs are nested. We decided to increase the complexity number of any nested construct by one less the number of its nested levels. This practice consequently allows us to derive the following process of calculating the nesting complexity metric.

PROCESS OF CALCULATING THE NESTING COMPLEXITY METRIC

The final relationship derived in the last section is the premise of our proposed metric. Since the purpose of our new complexity metric is to reflect the level of nesting within various constructs while keeping the computation process simple, we have formulated the process of calculating the "nesting complexity metric," $L(G)$, from the program text as follows:

1. Count and mark all the Boolean logical operators, "AND," "OR," and "XOR," in the program and assign *one* unit of complexity to each occurrence. However, do not count "NOT."
2. Count and mark keywords "IF," "WHILE," "UNTIL," and "CASE" at the first level of nesting within each flow construct. Once again, assign each occurrence with *one* unit of complexity. Note that if a control statement has at least one branch which leads directly to the program exit, any immediately following control statement should be considered to be at the first level of a new nested construct.
3. Count and mark all the remaining "IF," "WHILE," "UNTIL," and "CASE" keywords and assign *two* units of complexity to each occurrence.
4. Count and mark all but the first and the "ELSE" conditions in each "CASE" statement, and assign each occurrence with *one* unit of complexity. Remember to ignore the first and the "ELSE" conditions, otherwise each

level of the construct will be unnecessarily inflated by two units of complexity.
5. Sum up all the complexity units derived by the previous four steps and then add one into it. The total is our new complexity measure, $L(G)$.

Alternately, we can obtain $L(G)$ from the following two-step process:

1. Find the cyclomatic complexity measure, $V(G)$, using McCabe's approach.
2. Identify the second, the third, the fourth, and subsequent levels of nesting constructs and assign each occurrence with *one* unit of complexity. Do not forget that if a control statement has at least one branch which leads directly to the program exit, any immediately following control statement should be considered to be at the first level of a new nested construct.

Notice that the new nesting metric, $L(G)$, possesses all but one of the strengths and weaknesses of the cyclomatic metric, $V(G)$. It exchanges a weakness of $V(G)$ for one of its own strengths; namely, the $L(G)$ is now sensitive to the level of nesting but no longer is able to determine the maximum number of independent test paths. Three characteristics of the $L(G)$ metric are worth mentioning. First, the $L(G)$ metric assumes that nested constructs are more complex than simple constructs, but a nested control statement having one branch directing to the end of the program does not contribute additional complexity. Second, the $L(G)$ penalizes the excessive use of nested constructs and encourages substituting the "CASE" statement for the nested "ELSE IF" construct. Third, the $L(G)$ converges to the cyclomatic complexity metric if there is no nested construct in the program text.

After applying the counting procedure to the constructs A through R of the previous section, we have four groups of complexity:

1. Complexity of 2: $\{A,F,K,P\}$,
2. Complexity of 3: $\{B,G,L\}$,
3. Complexity of 4: $\{C,D,H,I,M,N,Q\}$,

and

4. Complexity of 6: $\{E,J,O,R\}$.

The relationship established earlier is therefore preserved.

In contrast, McCabe's $V(G)$ gives the following three groups of complexity:

1. Complexity of 2: $\{A,F,K,P\}$,
2. Complexity of 3: $\{B,C,G,H,L,M,Q,S,U,W,Y\}$,

and

3. Complexity of 4: $\{D,E,I,J,N,O,R,T,V,X,Z\}$.

Those constructs that have different metric values between $L(G)$ and $V(G)$ are shown in Table I.

TABLE I—Value of $L(G)$ versus $V(G)$ under the same construct

Construct	L(G)	V(G)
C	4	3
E	6	4
H	4	3
J	6	4
M	4	3
O	6	4
Q	4	3
R	6	4
U	4	3
V	6	4

TABLE II—Comparison of the outcomes of four complexity metrics

Algorithm from Kernighan and Plauger [24]	McCabe	Myers	Hansen	L(G)
A checkers move generator: ([24], pp. 41-42)				
Original version	17	(17:17)	(15,60)*	25
Improved version	17	(10:17)	(10,46)	17
Julian to Gregorian date conversion: ([24], pp. 43-46)				
Original version	19	(17:19)	(17,85)	30
Improved version #1	10	(5:10)	(5,25)	11
Improved version #2	11**	(6:11)**	(6,25)**	13
Merge two lists: ([24], pp. 18-19)				
Original version	5	(5:5)	(8,10)***	7
Improved version	5	(3:5)	(3,16)	5
Computer dating service: ([24], pp. 21-22)				
Original version	7	(7:7)	(7,10)	8
Improved version #1	6	(3:6)	(3,12)	6
Improved version #2	3	(3:3)	(3,6)	3

* Computed GOTO construct is regarded as CASE construct in this case.

** One unit of complexity is introduced by an additional DO-WHILE construct.

*** Arithmetic IF construct is counted as twice the complexity of logical IF's in this case.

APPLICATIONS OF THE NESTING COMPLEXITY METRIC

To validate the $L(G)$ metric, four algorithms from Kernighan and Plauger[24] were measured. A comparison of the outcomes of applying four different complexity metrics is illustrated in Table II. After scrutinizing the outcomes of these metrics, four major findings are notable:

1. The difference between the two elements in Myers's metric indicates the number of logical compound operators, that is, "AND," "OR," and "XOR."
2. The first element of Hansen's and Myers's metrics are the same if the program contains no "CASE" construct.
3. The only metric that does not reflect program improvement correctly is McCabe's metric. Although program improvement might introduce more operators, the control flow complexity should definitely be reduced.
4. The difference between $L(G)$ and $V(G)$ metrics indicates the number of nested levels. When $L(G)$ equals $V(G)$, the program contains no nested construct.

CONCLUSION

We have reviewed the cyclomatic complexity metric and its extensions, and have discussed strengths and weaknesses of the metric. An extension of the cyclomatic complexity metric, the "nesting complexity metric," has been proposed herein to remove the weakness of being insensitive to the level of nesting. Although the "nesting complexity metric," $L(G)$, is no longer able to directly determine the maximum number of independent test paths, it is superior to the cyclomatic complexity metric because it is now able to reflect the level of nesting structure and to penalize the excessive use of nested constructs thus encouraging the practice of substituting the "CASE" statement for the nested "ELSE IF" construct. Therefore, the nesting metric $L(G)$ is better than the cyclomatic metric $V(G)$ in measuring program complexity. However, we highly recommend using a pair metric of $(V(G),L(G))$ because is supplies more information than the $L(G)$ alone. Besides, the $V(G)$ number is readily obtained since it is a by-product of finding the $L(G)$ value.

REFERENCES

1. Curtis, B., S.B. Sheppard, P. Milliman, 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 (1979) 2, pp. 96–104.
2. Mills, H.D. "Mathematical Foundations for Structured Programming." FSC 72-6012, Gaithersburg, MD.: IBM Federal System Division, 1972.
3. Fitzsimmons, A.B. and L.T. Love. "A Review and Evaluation of Software Science." *ACM Computing Surveys,* 10 (1978) 1, pp. 3–18.
4. Mohanty, S.N. "Models and Measurements for Quality Assessment of Software." *ACM Computing Surveys,* 11 (1979) 3, pp. 251–275.
5. Berlinger, E. "An Information Theory Based Complexity Measure." *AFIPS Proceedings of the National Computer Conference,* Vol. 49, 1980, pp. 773–779.
6. Bulut, N. and M.H. Halstead. "Impurities Found in Algorithm Implementation." Technical Report CSD-TR-111, Computer Sciences Department, Purdue University, 1974.
7. Cornell, L. and M.H. Halstead. "Predicting the Number of Bugs Expected in a Program Module." Technical Report CSD-TR-205, Computer Sciences Department, Purdue University, 1976.
8. Elshoff, J.L. "Measuring Commercial PL/1 Programs Using Halstead's Criteria." *ACM SIGPLAN Notices,* 11 (1976) 5, pp. 38–46.
9. Fitzsimmons, A.B. "Relating the Presence of Software Errors to the Theory of Software Science." Presented to the 11th Hawaii International Conference of Systems Sciences, January 1978.
10. Funami, Y. and M.H. Halstead. "A Software Physics Analysis of Akiyama's Debugging Data." Technical Report CSD-TR-144, Computer Sciences Department, Purdue University, 1975.
11. Halstead, M.H. "An Experimental Determination of the "Purity" of a Trivial Algorithm." Technical Report CSD-TR-73, Computer Sciences Department, Purdue University, 1972.
12. Love, L.T. and A.B. Bowman. "An Independent Test of the Theory of Software Physics." *ACM SIGPLAN Notices,* 11 (1976) 11, pp. 42–49.
13. Schneidewind, N.F. and H.-M. Hoffmann. "An Experiment in Software Error Data Collection and Analysis." *IEEE Transactions on Software Engineering,* SE-5 (1979) 3, pp. 276–286.
14. Sunohara, T., A. Takano, K. Uehara, and T. Ohkawa. "Program Complexity Measure for Software Development Management." *Proceedings of the Fifth Internatinal Software Engineering Conference,* San Diego, California, 1981, pp. 100–106.
15. McCabe, T.J. "A Complexity Measure." *IEEE Transactions on Software Engineering,* SE-2 (1976) 4, pp. 308–320.
16. Berge, C. *Graphs and Hypergraphs,* Amsterdam, The Netherlands: North-Holland, 1973.
17. Harary, F. *Graph Theory,* Reading, Massachusetts: Addison-Wesley, 1969.
18. Chan, S.-P. *Introductory Topological Analysis of Electrical Networks,* New York: Holt, Rinehart and Winston, 1969.
19. Myers, G.J. *The Art of Software Testing,* New York: Wiley-Interscience, 1979.
20. Myers, G.J. "An Extension to the Cyclomatic Measure of Program Complexity." *ACM SIGPLAN Notices,* 12 (1977) 10, pp. 61–64.
21. Hansen, W.J. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)." *ACM SIGPLAN Notices,* 13 (1978) 3, pp. 29–33.
22. Walsh, T.J. "A Software Reliability Study Using a Complexity Measure." *AFIPS Proceedings of the National Computer Conference,* Vol. 48, 1979, pp. 761–768.
23. Ledgard, H.F. and M. Marcotty. "A Genealogy of Control Structures." *Communications of the ACM,* 18 (1975) 11, pp. 629–639.
24. Kernighan, B.W. and P.J. Plauger. *The Elements of Programming Style,* New York: McGraw-Hill, 1974.