A Survey of System Complexity Metrics

J. K. NAVLAKHA
Department of Mathematical Sciences, Florida International University, Miami, Florida 33199, USA

Measures of the structured design of software systems are called* system complexity metrics. *Two particularly promising system complexity metrics are described in this paper: Yin and Winchester's metric, which is derived from a system's structured design charts; and Henry and Kafura's metric, which is derived from a system's information flow. The values computed by both are available after the end of the design phase. Consequently, they are useful in the entire software development life cycle, from the design phase on. The definition, utility, interpretation and advantages of each metric are described. Validation studies and their results are also reported for each metric. It is noted that Yin and Winchester's metric is quite successfully used at Hughes Aircraft Company but that there is no published report of the use of an information flow metric by any software organisation.

Received February 1985, revised March 1986

1. INTRODUCTION

Research into the development of a metric for software complexity has attracted much attention in the computing community. Many software complexity metrics which have been developed in the past 15 years or so can be divided into two categories, those dealing with single programs (program complexity metrics) and those dealing with the structured designs of entire systems (system complexity metrics).

There have been three novel approaches used in developing program complexity metrics. The first, which was developed by Halstead,[4] uses a series of software science equations to measure the complexity of a program based on the lexical counts of symbols used. The second approach, due to McCabe,[11] uses graph-theoretic measures to define a cyclomatic complexity metric. In this approach, the control structure of a program is mapped into a structure of nodes and edges according to the control flow, and the complexity of the resulting graph is used to provide a measure for the complexity of the program. The third approach is due to Albrecht,[1] who hypothesised that the amount of function to be provided by an application program can be estimated from an itemisation of the major components of data to be used or provided by it. Accordingly, his function-point metric is based on the count of external user inputs, inquiries, outputs and master files to be delivered by the development project.

Halstead's software science metrics have been extensively studied, and two excellent reviews have been written.[3,6] McCabe's cyclomatic complexity has also been the subject of popular research, and many researchers have provided extensions to his metric. Typical extensions are described in Refs 5, 12 and 15. Albrecht's function-point metric is a newer one, but is fast gaining acceptance in the industry. The IBM Corporation uses this metric in several divisions for the measurement of software complexity, the prediction of the number of source lines in a program, the prediction of expected future errors and measurement of software productivity, etc. A comparison of these three metrics in terms of their ability to measure software productivity has led to the conclusion that in the areas where it is applicable, the function-point metric is the best of the three.[14]

There have been two noteworthy efforts in the development of system complexity metrics. The first, by Yin and Winchester computes metrics based on a system's design structure charts.[19] The other approach, due to Henry and Kafura[7,8], is based on system information flow.

It should be noted that the values of Halstead's metrics become available only after the coding is done, and therefore can be of use only during the testing and maintenance phases. The value of McCabe's metric is available only after the detailed design is done. In comparison with those two program complexity metrics, system complexity metrics are determined during the design phase, and thus their utility extends from the design phase onwards in the software development life cycle.

In this paper we shall describe the fundamentals of system complexity metrics, the results of the few experiments that have been performed to validate them and some of the application areas where they can be of use. The only other survey on system complexity metrics which is currently available in the literature is a short summary by Ince.[10] Program complexity metrics have been well reviewed, and thus it is not necessary to survey them further here.

2. YIN AND WINCHESTER'S METRICS

B. H. Yin and J. W. Winchester, of Hughes Aircraft Company, have developed system complexity metrics based on analysis of a system's design structure chart.[19] Metrics which depend on design structure charts can be useful in identifying sections of a design that may cause problems during coding, debugging, integration and modification.

2.1 Metrics definitions

To understand the metrics, consider the design structure chart given in Fig. 1.

The rectangular boxes represent processing modules, the convex boxes represent database tables, and the

arrows represent data and control transfers. The notations used are as follows.

(*a*) $L = 0, 1, 2, 3$; levels of the system.

For $i = 1, 2, 3$

(*b*) N_i; number of modules from level 0 to level i.

A_i; number of module network arcs from level 0 to level i.

$T_i = N_i - 1$; number of module tree arcs from level 0 to level i.

N'_i; number of modules and database references from level 0 to level i.

A'_i; number of module and database network arcs from level 0 to level i.

T'_i; number of module and database tree arcs from level 0 to level i.

(*c*) $\triangle T_i = T_i - T_{i-1}$

$\triangle A_i = A_i - A_{i-1}$

$\triangle T'_i = T'_i - T'_{i-1}$

$\triangle A'_i = A'_i - A'_{i-1}$

These values can be determined from a design structure chart. Based on them, two types of *primary metrics* are defined.

(*a*) Excluding database references:

$C_i = A_i - T_i$

$R_i = 1 - T_i/A_i = C_i/A_i$;

$D_i = 1 - \triangle T_i/\triangle A_i$

(*b*) Including database references:

$C'_i = A'_i - T'_i$

$R'_i = 1 - T'_i/A'_i = C'_i/A'_i$

$D'_i = 1 - \triangle T'_i/\triangle A'_i$

Additionally, two *secondary metrics* are also defined.

(*a*) *Fan-in* of a given module or database A, the number of modules that call or directly reference A.

(*b*) *Fan-out* of a given module A, the number of modules that are directly called by A.

For the example design chart of Fig. 1, the values of the primary metrics are as shown below.

$N1 = 3$; $A1 = 3$; $T1 = 2$;

$N2 = 6$; $A2 = 8$; $T2 = 5$;

$N3 = 8$; $A3 = 10$; $T3 = 7$;

$N1' = 3$; $A1' = 3$; $T1' = 2$;

$N2' = 7$; $A2' = 10$; $T2' = 6$;

$N3' = 11$; $A3' = 15$; $T3' = 10$;

$\triangle T1 = T1 - 0 = 2$; $\triangle A1 = A1 - 0 = 3$;

$\triangle T2 = T2 - T1 = 3$; $\triangle A2 = A2 - A1 = 5$;

$\triangle T3 = T3 - T2 - 2$; $\triangle A3 = A3 - A2 = 2$;

$C1 = A1 - T1 = 1$; $R1 = C1/A1 = 1/3$;

$C2 = A2 - T2 = 3$; $R2 = C2/A2 = 3/8$;

$C3 = A3 - T3 = 3$; $R3 = C3/A3 = 3/10$;

$D1 = 1 - \triangle T1/\triangle A1 = 1 - 2/3 = 1/3$;

$D2 = 1 - \triangle T2/\triangle A2 = 1 - 3/5 = 2/5$;

$D3 = 1 - \triangle T3/\triangle A3 = 1 - 2/2 = 0$;

2.2 Properties and interpretation of the metrics

Metrics like Yin and Winchester's which assess the form, structure and complexity of the design, code or documentation of a system in order to predict future events involving that system are called *predictive* metrics. In the case of Yin and Winchester's metrics, the metrics are available from the design phase onwards and hence can be used to predict such values as the number of errors the system is likely to have, the time that will be required to test the system, the time that will be required to correct errors, and the like.

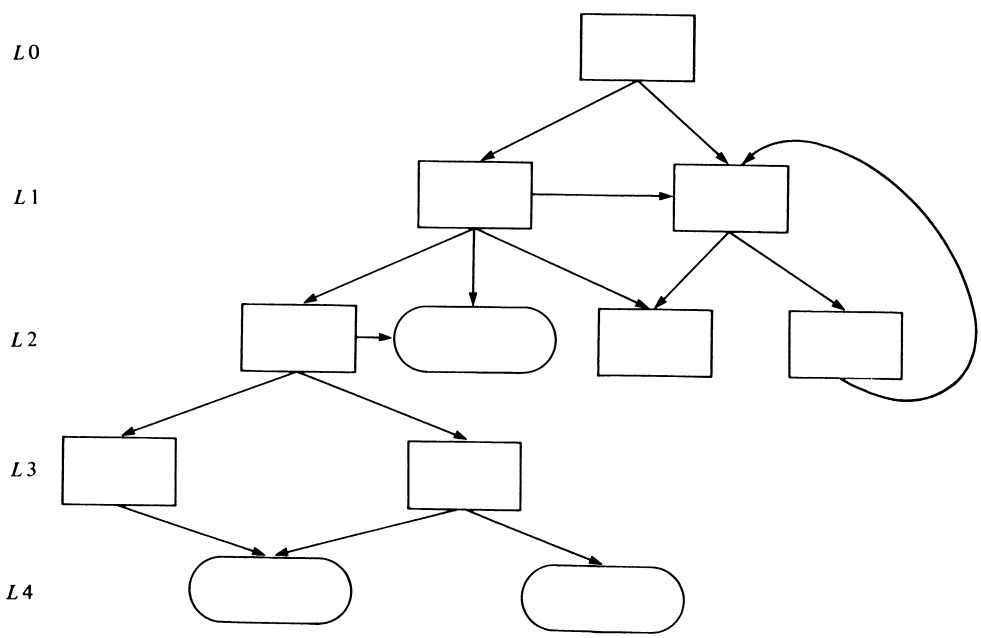

Figure 1. A sample design structure chart. Taken from Ref. 18, copyright © 1980 – IEEE.

These metrics are based on two identifiable attributes of any structured design, namely coupling and simplicity. Schneidewind[16] has defined coupling between modules of an operating system as the number of arcs in the system. McCabe[11] has stated that program complexity is independent of the program's physical size but depends only on the decision structure of a program.

The *C*-metric is a measure of the intermodule coupling. C_i is a monotonically non-decreasing function which indicates the fluctuation of the increment of T_i against A_i. This difference of network arcs and tree arcs in the design from level 0 to level i effectively measures the network complexity. Its value should be kept as low as possible. A sharp increase of C_i from one level to the next signals an extreme complexity increase and may suggest a place for design changes.

The *R*-metric is a ratio of module coupling to system size. R_i and D_i measure the tree impurity at level i. The distinction between them is that R_i measures the tree impurity of level i against level 0 whereas D_i measures that of level i against level i-1. Values of R_i and D_i lie between 0 and 1, with both being 0 for a tree structure and increasing as the system moves away from a tree structure.

The definition of level 0 depends on the designer and metric evaluator in the sense that the top level of a subsystem of a major system is as good a candidate for level 0 as the top level of the system itself. When examining a particular section of a design, if *C* is high but *R* is not, this may indicate that the section being examined is too large and should be broken into smaller components. After 5 or 6 levels, *R* should become more insensitive to the system. It is desirable to get *R* to flatten out at a low value since a low value of *R* represents a low *C* value. *D* is not sensitive to *L*. It is better to have only mild fluctuations in *D*.

Comparison of primed and unprimed metrics indicates database activity and dependency. The secondary metrics can provide an indication of the use of a particular model or database table in the entire design. High fan-in suggests low cohesiveness while high fan-out suggests scope of effect/control problems. They also provide an indication of the use of a particular module or database table in the entire design. Modules having high fan-in should be checked to ensure that they do not perform more than one function and are not too tightly coupled.

Yin[18] has described a software design and testability analysis system developed at Hughes which is configured for the AMDAHL/470 and accessible via a HP2648A graphics terminal. The first part of this system is a structure chart graphics system which allows designers to draw and modify structure charts and have the design stored in the database. The second part, called the design quality metrics system (DQM), accesses this database and Fig. 2 shows typical DQM output.

In the example of Fig. 2, the *C*-curve suggests a complexity increase between levels 2 and 4 as well as high connectivity among modules at those levels. The *R*-curve suggests high fan-in or high connectivity between modules at levels 2 and 4. The oscillations in the *D*-curve indicate the addition of a number of new modules at about every other level. The *D*-value is high at level 7, but the fact that *R* flattens out after level 5 indicates that compared to the size of the system, very few non-tree arcs have been added after level 5. This shows that level 7 is not really a problem. All three metrics point to the same place, namely between level 2 and level 4, as a potential problem area for more intensive inspection.

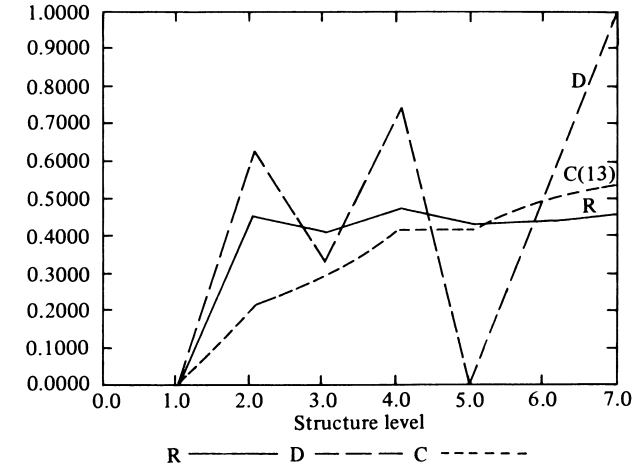

Figure 2. Example graphics output from DQM. Taken from Ref. 18, copyright © 1980 – IEEE.

2.3 Validation of the metrics

These metrics have been in use at Hughes Aircraft Company for some years now, and all reports indicate their success in pinpointing error-prone areas in the design which could cause problem in subsequent life-cycle phases. One of the first validations of the metrics was done on two projects completed at Hughes in the late seventies. The first was a large software project containing over 1,000 modules, while the second was considerably smaller. Twenty-seven primary metric calculation charts representing 15% of the design were produced for project 1 and three charts representing the complete design were produced for project 2. Validation was done by performing trend and correlation analyses between the program error data and metric calculations under the assumption that a good design should yield low-error software.

In attempting to validate the metrics, the first step was determination of the correlation between *R* and number of errors, and between *C* and number of errors. For all 27 charts of project 1, a consistently high correlation was found between R_i and number of errors from level 0 to level i (NE_i), and C_i and NE_i (all except one with correlation coefficient greater than 0.6 and most with correlation coefficient greater than 0.9). Correlations between C_i and NE_i were generally higher than those between R_i and NE_i because both C_i and NE_i are monotonically increasing functions. Although R_i is not monotonically increasing, it attains an asymptotic value, and the probability of reaching this value increases with the number of levels. For those subsystems with over 10 levels, the correlations were close to 0.97. Thus there was clearly an association between *C* and number of errors, and between *R* and number of errors.

Computing correlations between the final *C* value and the total number of errors in the system gave correlation coefficients of 0.98 for the 27 subsystems of project 1 and 0.99 for 3 subsystems of project 2. This confirms a design guideline that *C* should be kept as low as possible to reduce the total number of errors in the system.

Another validation test performed was the trend analysis between D_i and F_i, which is the number of errors in level i divided by the number of modules in level i. Recall that D_i measures the tree impurity between levels i and i-1. A statistical test was performed to determine if D_i and F_i have the same trend from one level to the next. It was found that for subsystems with a reasonably large number of levels the association between D_i and F_i was mildly significant. The trend analysis also showed that the general trend of number of errors in modules at level i can be predicted from D_i.

The design and testability analysis system developed at Hughes also contains a testability analysis system. It isolates design areas that can be independently tested, singles out the modules that destroy the independence between areas, produces a hierarchy of independent tests based on this analysis and produces a measure of difficulty of the test based on the complexity of interactions between modules. Thus the complete system identifies the error-prone areas in the design, identifies the difficult-to-test areas by measuring testability and provides a measurement of software quality in terms of reliability, maintainability and testability.

3. HENRY AND KAFURA'S METRICS

Whereas Yin and Winchester's work focused on the interface between the major levels in a large hierarchically structured system, Henry and Kafura's work[7,8] uses the information flow approach. Their approach is much more detailed because it observes all information flow rather than just flow across level boundaries. It has another major advantage in that this information flow method can be completely automated. This is unlike other information-theoretic concepts like Channon's,[2] where each assumption of each procedure must be explicitly determined.

It has been argued by the authors of this technique that it is an appropriate and practical basis for measuring large-scale systems. The major elements in the information flow analysis can be directly determined at design time, thereby allowing any corrections in the system structure with the minimum cost. Also, by observing the patterns of communication among system components, it is possible to define measurements for complexity, module coupling, level interactions and stress points in the design. These critical system qualities cannot be derived from simple lexical measures. Furthermore, this technique reveals more of the system connections than other ordering relations such as 'calls', 'uses' and 'dependency'.

3.1 Metrics definition

The following definitions* are required to understand and describe the ideas of information flow precisely.

Definition 1. There is a *global flow of information* from procedure A to procedure B through a global data structure D if A deposits information into D and B retrieves information from D.

Definition 2. There is a *local flow of information* from procedure A to procedure B if one or more of the following conditions hold:

* Taken from Ref. 8, copyright © 1981–IEEE.

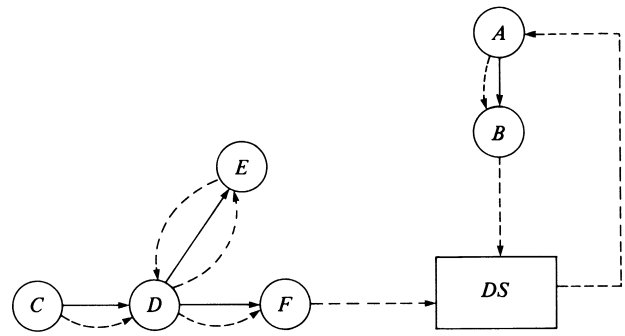

Figure 3. A sample flow diagram. Taken from Ref. 8, copyright © 1981 – IEEE.

(1) if A calls B,

(2) If B calls A and A returns a value to B, which B subsequently utilises, or

(3) if C calls both A and B, passing an output value from A to B.

Definition 3. There is a *direct local flow of information* from procedure A to procedure B if condition (1) of definition 2 holds for a local flow.

Definition 4. There is an *indirect flow of information* from procedure A to procedure B if condition (2) or condition (3) of definition 2 holds for a local flow.

In order to understand the metrics, consider the flow diagram of Fig. 3 taken from Ref. 8.

Fig. 3 shows six modules, A to F, a data structure DS, and the connections among them. Module A retrieves information from DS and then calls B, which then updates DS. C calls D, which calls E and E returns a value to D which D utilises and passes to F. F updates DS.

Informally, the *direct local flows* of information generated here are

$$A \rightarrow B$$

$$C \rightarrow D$$

$$D \rightarrow E$$

and $$D \rightarrow F$$

These flows are simply the ones observed in the calling sequence. The *indirect local flows* of information are

$$E \rightarrow D$$

$$E \rightarrow F$$

$E \rightarrow D$ results when E returns a value used by D and $E \rightarrow F$ results when D passes information received from E to F. This is the information flow where no control flow exists. There also exists a flow of information through the global data structure DS again without a control flow. These *global flows* are

$$B \rightarrow A$$

$$F \rightarrow A.$$

Note that these flows are not affected by the syntax of the source language being used. Also, no distinction is made between a flow of information established by a passed parameter and one established by a shared global data structure. Computation of all information paths is easy once the flow of information is available for each procedure in a source-target type of relationship. If sufficiently precise design language is used these relations can be generated from the design code and the information flow analysis can be done at design time.

Based on these information flows, Henry and Kafura defined a metric to determine the complexity of a procedure. This complexity depends on two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment. A simple length measure (the number of lines of text in the source code of the procedure) was used as an index of the procedure code complexity. (It is pointed out, however, that Halstead's length measure or McCabe's cyclomatic complexity may be substituted for it and they may improve upon the accuracy.) The procedure's connections with the environment can be expressed as some measure of fan-in and fan-out of the procedure.

The *complexity metric of a procedure* is then defined as

$$\text{length}*(\text{fan-in}*\text{fan-out})^2.$$

The product fan-in*fan-out represents the total possible number of combinations of an input source to an output destination. The weighing of this component is based on the belief that the complexity is more than linear in terms of the connections which a procedure has to its environment. After the detailed design is done, the value of fan-in*fan-out is known. The code length is only a weak factor in the complexity measure and hence may be substituted by an intelligent estimation.

The procedure complexities are used to establish module complexities. A *module* with respect to a data structure *D* consists of those procedures which either directly update *D* or directly retrieve information from *D*. The *complexity of a module* is defined to be the sum of the complexities of procedures within the module. Another interesting measurement easily calculated for a given module is the number of paths of information possible among its procedures through the module's data structure. The formula used to calculate the number of global flows is

$$(\text{write}*\text{read})+(\text{write}*\text{read\_write})+(\text{read\_write}*\text{read})+(\text{read\_write}*(\text{read\_write}-1))$$

where 'read', 'write' and 'read_write' are the number of read-only, write-only and read_write procedures of the module respectively.

3.2 Utility of the metrics

The procedure and module complexities which are available after the design phase can be utilised to improve the design and reduce the complexity of software development tasks in subsequent phases of the life cycle. The procedure complexities can reveal three potential problem areas in a given procedure.

(1) *Lack of functionality.* A high fan-in and fan-out reveals a large number of connections of a procedure to its environment, indicating that it may perform more than one function.

(2) *Stress point in a system.* A high complexity shows a stress point in a system. That is, it indicates a high information flow through that procedure. At such a stress point it is difficult to implement changes to the procedure because of the large number of potential effects on its environment and on other procedures.

(3) *Inadequate refinement.* A high fan-in or fan-out may indicate a missing level of abstraction in the design process. Perhaps the procedure should be divided into two or more procedures.

The global flows and the module complexities can show four areas of potential design or implementation difficulties for the module.

(1) *Poorly refined* (i.e. overloaded) *data structure.* Overloading may be avoided by redesigning the data structure to segment it into several pieces.

(2) *Improper modularisation.* It is desirable that a procedure be in one and only one module. Violations of this property can be found when computing the module complexities.

(3) *Poor internal module construction.* High global flows and a low module complexity indicate poor internal module construction. This indicates that many procedures access the data structure directly but there is little communication among them.

(4) *Poor functional decomposition.* A low global flow and high module complexity indicate a poor functional decomposition within the module or a complicated interface with other modules.

3.3 Validation of the metrics

Henry and Kafura validated their metric by evaluating version 6 of the UNIX operating system. When doing the complexity calculations, procedures written in assembly language and certain 'memoryless' procedures which do not communicate information across successive invocations and thus terminate an information flow path were not considered. Only the non-trivial modules of UNIX were examined. The hypothesis used for validation was that the complexity metric is highly correlated with the occurrence of system changes.

It was found that one data structure U of UNIX was heavily overloaded with 3,303 global flows. The module corresponding to U contained 84 procedures. The primary reason for this is that a function of this structure is to pass error codes across levels in the system. It was discovered that for all but one module, at least 85% of the module complexity was attributable to the three largest procedures.

Improper modularisation was clearly visible in this experiment. One goal of modularisation is to ensure that a procedure belongs to only one module. If a procedure violates this principle, it is more error-prone due to its connections with more than one module. It was found that 38 out of 53 (72%) procedures belonging to more than one module were included in the list of procedures in which changes occurred. On the other hand only 42 out of 112 (38%) procedures that belonged to only one module went through changes.

The authors considered four factors: length, (length**2), (fan-in*fan-out) and (fan-in*fan-out)**2, and wanted to determine which of them contributes most to the complexity correlations. They found that (fan-in*fan-out)**2 is an extremely good indicator of complexity, having a correlation coefficient of 0.98 with changes incurred.

It should be noted that a wide range of measurements can be derived from information flow. One example refers to a design goal to minimise connections among modules. The information flow metrics can recognise 'content' coupling and 'common' coupling[13] fairly easily. Content coupling which addresses direct references between modules is equivalent to the direct local flows. Common coupling refers to the sharing of global data structures

and is equivalent to the global flow. A high coupling between two modules indicates that a substantial change in one module will quite likely force changes in the other also. Thus coupling provides a measure of modifiability.

Information flow metrics which include procedure, module and interface measurements help in locating potential design and implementation weaknesses. Although not many large systems are evaluated, it seems that this is a good technique for measurement of software quality for large systems.

4. CONCLUSIONS

The study of system complexity metrics has not received as much attention as the area of program complexity metrics. Whereas program complexity metrics have been subjected to a considerable number of experimental studies for their validation, the same is not true of system complexity metrics.

Yin and Winchester's metrics have been validated on many large software development projects at Hughes and are reported to be in continuous use there. They defined system complexity in terms of a system design's deviance from a pure tree structure. Troy and Zweben[17] discovered that metrics concerned with inter-module coupling were the most important out of a large set studied, thereby providing confirmation of Yin and Winchester's approach.

Henry and Kafura's metric, on the other hand, has been validated only on the UNIX operating system. It is a very appealing metric but there is no published report of its regular use in any software organisation. Henry *et al.*[9] have discovered that McCabe's and Halstead's metrics are orthogonal to their own metric. This, coupled with the fact that the information flow metric is applicable from the design phase onwards, increases the utility of this metric substantially.

REFERENCES

1. A. J. Albrecht and J. E. Gaffney, Jr, Software function source lines of code, and development effort prediction: a software science validation. *IEEE Transactions on Software Engineering* **SE-9** (6), 639–648 (1983).
2. R. N. Channon, On a measure of program structure. *Proceedings of the Programming Symposium, Paris*, pp. 9–16. Springer-Verlag, Heidelberg (1974).
3. A. Fitzsimmons and T. Love, A review and evaluation of software science. *Computing Surveys* **10** (1), 3–18 (1978).
4. M. H. Halstead, *Elements of Software Science*. Elsevier, Amsterdam (1977).
5. W. J. Hansen, Measurement of program complexity by the pair (cyclomatic number, operator count). *SIGPLAN Notices* **13** (3), 29–33 (1978).
6. W. Harrison, K. Magel, R. Kluczny and A. Dekock, Applying software complexity metrics to program maintenance. *IEEE Computer*, pp. 65–79 (1982).
7. S. Henry, Information flow metrics for the evaluation of operating systems' structure, *Ph.D. Thesis*, Department of Computer Science, Iowa State University, Ames (1979).
8. S. Henry and D. Kafura, Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* **SE-7** (5), 510–518 (1981).
9. S. Henry, D. Kafura and K. Harris, On the relationship among three software metrics. *SIGMETRICS Performance Evaluation Review* 81–88, (1981).
10. D. C. Ince, The influence of system design complexity research on the design of module interconnection languages. *SIGPLAN Notices* **20** (10), 36–43 (1985).
11. T. J. McCabe, A complexity measure *IEEE Transactions on Software Engineering* **SE-2** (4), 308–320.
12. G. J. Myers, An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices* **12** (10), 61–64 (1977).
13. G. J. Myers, *Software Reliability Principles and Practices*. Wiley-Interscience, New York (1976).
14. J. K. Navlakha, Software productivity metrics: some candidates and their evaluation. *Proceedings of the National Computer Conference*, 1986 69–75 (1986).
15. P. Piwowarski, A nesting level complexity measure. *SIGPLAN Notices* **17** (9), 44–50 (1982).
16. N. F. Schneidewind, Modularity considerations in real time operating system structures. *COMPSAC*, pp. 397–403 (1977).
17. D. A. Troy and S. H. Zweben, Measuring the quality of structured designs. *Journal of Systems and Software* **2**, 113–120 (1981).
18. B. H. Yin, Software design testability analysis, *COMPSAC*, pp. 729–735 (1980).
19. B. H. Yin and J. W. Winchester, The establishment and use of measures to evaluate the quality of software designs. *Proceedings of the Software Quality and Assurance Workshop*, pp. 45–52 (1978).