N. R. Hall
S. Preiser

Combined Network Complexity Measures

Most of the considerable work that has been done in the measurement of software complexity during the past several years has addressed complexity measurement of source code or design languages. Here we describe techniques to measure the complexity of large (>100 000 source lines of code) systems during the software architecture phase, before major design decisions have been made. The techniques to measure and reduce complexity are intuitively reasonable, easy to apply, and produce consistent results. Methods developed include (1) an extension of the graph-theoretic measure developed by McCabe to software architecture, as represented by networks of communicating modules, (2) a general technique that allows the complexity associated with allocation of resources (CPU, tape, disk, etc.) to be measured, and (3) a method that combines module complexity and network complexity, so that design trade-offs can be studied to determine whether it is advantageous to have separate modules for service functions, such as mathematical subroutines, data management routines, etc.

1. Introduction

In recent years, software complexity measurement has been the subject of considerable research. Intuitively, high software complexity contributes to difficulty in development, testing, and maintenance, as well as adding to reliability problems. The problems resulting from complexity are particularly acute in large (>100 000 source lines of code), complex software systems.

Much of the research in software complexity has been aimed at existing programs or, more recently, at structured design languages. We are interested in the software architecture phase of software development, since it is our view that complexity measurement techniques, if applied earlier in the life cycle, can aid in identifying areas in software systems that are unnecessarily complex. Methods for reducing complexity can then be applied to such areas before major design decisions have been made.

The system representation that we have chosen to study is that of networks of modules. It is felt that this representation is simple, direct, and in particular allows the modeling of systems with concurrent processing or real time considerations. It is our goal to develop techniques that would allow comparison of the design alternatives which abound in such systems and to provide assistance in choosing among them.

In the next section of this paper, we consider the factors that led to our selection of a model for concurrent systems and to our choice of a measure of complexity for such systems. In the following section, the application of McCabe's complexity measure to concurrent systems represented as networks of modules is described, and a generalized complexity measure is developed that can account for the effects of such resources as channels, disk drives, CPUs, etc. The final section describes a technique for combining the measurement of complexity of networks of modules with conventional measures of complexity to provide an overall indication of system complexity that can aid in making design trade-offs.

2. Model and complexity measure selection

- *Choice of a concurrent systems model*

A model for concurrent systems should be able to accommodate

1. Synchronization and communication,
2. Deadlock avoidance and/or detection,
3. Rigorous methodologies for software architecture/design,
4. Resource allocation.

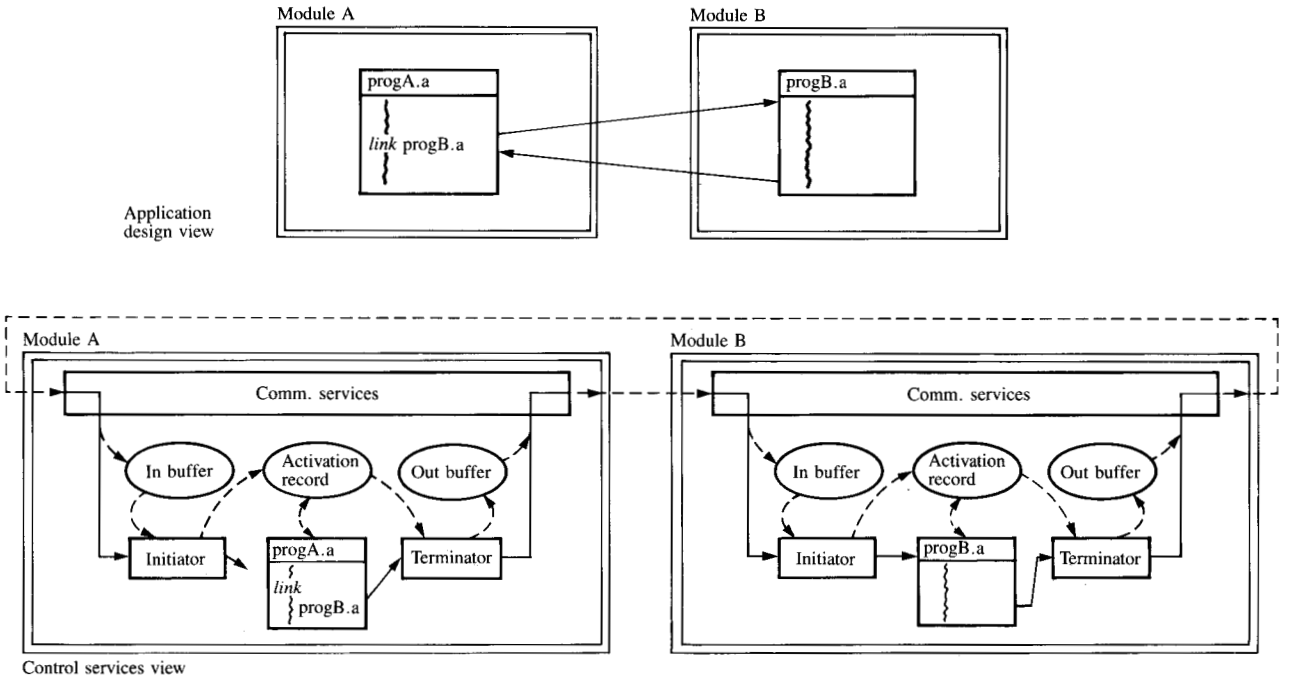

Figure 1 Applications and control services.

Many of the models for concurrent systems reported in the literature provide these attributes. Some of the more traditional models available include those described by Mattheyses and Conry [1], Hoare [2], and Owicki and Gries [3]. However, they do not in general provide a unified, internally consistent model for design that can model existing complex systems.

Witt's model [4] provides a unified approach to the design of systems capable of concurrency. It allows stepwise refinement, with the state machine as the underlying model. Witt's communicating-modules model addresses the issues of synchronization, deadlock, and resource allocation. Therefore, it was decided to use this model to represent concurrent processes. More important, if we are able to develop complexity measures for this model, we will be able to apply complexity measurement techniques to the software architecture and design process. This would be a great advantage, since complexity measurement techniques are currently used primarily for code, as described by Woodward et al. [5], and in some experimental situations, for process design languages and clusters of programs. The work on clusters was done by Belady and Evangelisti [6].

Some of the terminology of Witt is used throughout the remainder of this paper, including the following definitions.

Definition 1 [4] Procedures are formed from declarations of data combined with structured programs, i.e., control structures whose "do" parts describe data transformations. One procedure may invoke others. Hence, "programs" are seen as hierarchies of procedures.

Definition 2 [4] A module, represented by a state machine, is an aggregate of programs sharing a common perspective and common data objects. The common data are shared by all programs in the aggregate, are inaccessible to any other program, and are retained between successive invocations of the module.

In the communicating-modules model, the components of a software system are partitioned into one of three classes: applications, control services, and hardware services. Application programs compute the information required by the client without regard to the potential interference of other programs being executed. They map client-oriented input into client-oriented output, and they modify client-oriented state data. Control services, which include run-time services such as creation of address space, intermodule communication, and noninterfering access to common data, are dependent on neither specific hardware nor specific client programs. Control services intercept program invocations, embed parameters into messages, and send messages to the addressed module. In a similar way control services return output parameters to the invoker. The applications programs appear to control services as a network of modules. Hardware services are concerned with presenting a "friendly"

representation of the physical hardware being used. They are not concerned with the network of programs communicating between different memories; they are only concerned with the execution of one or more processors in a single memory.

We examine applications programs as viewed by control services, i.e., as a network of communicating modules, with each module possibly in a different memory. An illustration of this model can be seen in Fig. 1.

• *Selection of a complexity measurement model*

The initial step in developing a suitable complexity measure for networks of modules, particularly those that allow concurrency, was to examine existing software complexity measures for sequential processing. As noted before, most of the existing measurement techniques were developed to measure complexity of the code or design associated with an individual module.

Complexity measurement is ill-defined in software engineering. There are many measures of complexity, such as algebraic complexity, computational complexity, etc., with considerable overlap among them. Belady [7] believes that program complexity is perceived in at least two different ways: It is a measure of uncertainty or surprise, or it is deterministically defined as a count of magnitude (such as amount of storage, number of instructions, etc.). The deterministic approach, which we are interested in here, consists of selecting a countable property of the program, which is then asserted to be related to complexity. This definition is also used by Ruston [8]. Storm and Preiser [9] compare a complexity measure to a norm, with a nonnegative number being assigned to a complex object in order to assess its "length." This is also analogous to the deterministic approach.

Evaluation criteria

The existing software complexity measures were evaluated to determine which technique provided the best model for measuring the complexity of networks of modules. Identification of this method would provide a starting point for dealing with some of the concurrency issues discussed later.

Some more definitions are needed.

Definition 3 Network complexity is a function based on some countable properties of the modules and inter-module connections in the network that are believed to be related to complexity.

Definition 4 A subnet S of a network N is a network consisting only of modules and intermodule connections of N. If S is not identical to N, then we say that S is a proper subnet of N, written

$S \subset N$.

Definition 5 A network complexity measure C is called consistent if for $S \subseteq N$,

$C(S) \leq C(N)$.

The criteria used for our evaluation are as follows:

1. The measure should be easy to apply to concurrent systems.
2. It should agree with intuitive ideas about system complexity.
3. The results obtained using the measure should be consistent.
4. The measure should be suitable for application to a network representation.

Measurement techniques

Several of the existing complexity measurement techniques were selected for more careful examination. These included McCabe's complexity measure (cyclomatic number) [10], Halstead's software science measure [11], source lines of code (SLOCs) as described by Walston and Felix [12], Storm and Preiser's index of complexity for structured programs [9], Laemmel and Shooman's complexity measure using Zipf's Law [13], and Ruston's polynomial measure of complexity [8]. An overview of these techniques is presented here.

Halstead's (software science) complexity measure

Halstead states that the complexity of any algorithm can be measured directly from a static expression of that algorithm in any language. Given an implementation of the algorithm in any language, it is possible to identify all the operators and operands. It is then possible to define a number of measurable characteristics from which Halstead's software science measures are derived. The following definitions are used.

Definition 6 [11] An operand is a variable or constant that is used in the implementation.

Definition 7 [11] An operator is a symbol or combination of symbols that affects the value or ordering of operands.

Halstead shows that estimated effort, or programmer time, can be expressed as a function of operator count, operand count, or usage count. It is thus possible to predict how much programmer time will be required by analyzing the program. Halstead conducted some experiments to show that predicted and actual programmer times have coefficients of correlation which are on the order of 0.9.

Halstead's method has been used by many organizations, including IBM at its Santa Teresa Laboratory [14], General Electric Company [15], and General Motors Corporation [16], primarily in software measurement experiments. It has also been used by several organizations on actual projects. Although the use of Halstead's method may allow the

reviewer to determine the complexity of a particular program, techniques for reducing complexity are not discussed.

The Halstead measure provides a very concise technique for assessing program complexity, by computing the number of operators and operands. But this technique is difficult to apply to networks of modules, because operators and operands are not easily identified in that context.

Source lines of code

The source lines of code (SLOC) measurement of program size has been used for years in software cost estimation, and also as a complexity factor. Estimated and actual source lines are used for productivity estimates, as described by Walston and Felix [12], during the proposal and performance phases of software contracts. The way that source lines are counted varies among organizations. For example, source lines can be viewed as 80-column cards, lines of source code, lines of executable code, etc. Variations in complexity are estimated by assigning a difficulty factor (e.g., easy, average, difficult). SLOCs have provided a straightforward method of cost estimation and complexity measurement for some time and will probably continue to do so in the future. This measure is both elementary and easy to calculate, and remains a useful "quick and dirty" estimate of complexity.

In examining the use of source lines as a complexity measure, it became clear that this technique did not take into account concurrency problems, such as allocation of resources, and the more general problem of control flow complexity. It could only account for the number of lines of code needed, and not for the added complexity inherent in such networks.

Storm and Preiser's index of complexity

The index of complexity was developed especially for structured programs. It provides an *a priori* measure of program complexity, so that the programmer can be given an indicator when the program is likely to exceed the limit of easy comprehension. Since this limit can easily be computed before compilation, if the index is "too large," reduction in the complexity is strongly suggested. The primary result of Storm and Preiser's study [9] is the following theorem.

Theorem 1 [9] The index of complexity for structured programs is less than or equal to the index of complexity for unstructured programs.

Storm and Preiser's technique provides a good way of assessing program structures, but these are not comparable to the kinds of structures that one would find in a network of modules, unless one assumed at the ouset that all networks were structured.

Storm and Preiser's index is discussed in more detail later.

Laemmel and Shooman's complexity measure

Laemmel and Shooman [13] have examined Zipf's Law, which was developed for natural languages, and extended the theory to apply the technique to programming languages. Analogies between natural language and computer programming languages are drawn to show that there is overall agreement in the buildup of the language from basic constructs.

Zipf's Law is applied to operators, operands, and the combinations of operators and operands in computer programs. The results show that Zipf's Law holds for computer programming languages, and complexity measures can be derived which are similar to those of Halstead. This method provides the precision associated with the Halstead method but is somewhat easier to apply.

Although Laemmel and Shooman's method provides a concise technique for measuring program complexity, it is more suitable for application to programs or modules than to networks. In this respect, it is similar to the Halstead measure.

Ruston's polynomial measure of complexity

Ruston's measure [8] describes a program flowchart by means of a polynomial. The measure takes into account both the elements of the flowchart and its structure. Rules are given for obtaining the polynomials for various flowcharts, such as structured, unstructured, etc. The measure allows the comparison of alternative designs, and gives bounds on cyclomatic (McCabe's) complexity. Ruston also makes a comparison of this measure with several other complexity measures. Ruston's method appears to be suitable for network measurement, but has not been used as widely as McCabe's method (to be described), and it results in a more complex expression.

McCabe's complexity measure

McCabe [10] developed a mathematical technique, based on program control flow, which provides a quantitative basis for modularization of software and for identification of software modules that will be difficult to test or maintain. McCabe's measure is based on a graph-theoretic approach, and McCabe shows that complexity is independent of physical size and depends only on the decision structure of a program. A bound on complexity is identified, and techniques are described for reducing complexity. This technique has been further extended and used in conjunction with a testing methodology.

Evaluation of results

The overall results of this evaluation are summarized in Table 1. Based on our evaluation of these techniques, it was decided to experiment with the McCabe model for measurement of complexity in networks of modules. It satisfied the

criteria above and provided a starting point for measuring network complexity. Other graph models have been developed for networks, such as Petri nets described by Baer [17] and could also have been used.

• *McCabe's measure for programs*

McCabe's measure is based on a graph-theoretic approach and uses some basic definitions and theorems from graph theory, which are repeated here.

Definition 8 [18] The cyclomatic number $V(G)$ of a graph G with n nodes, e edges, and p connected components is

$$V(G) = e - n + p.$$

Definition 9 [19] A linear graph is said to be strongly connected if for any two edges r and s, there exist paths from r to s and from s to r.

Theorem 2 [18] In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

The basis for McCabe's approach is as follows: Given a computer program, associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential, and the edges correspond to the branches taken in the program. It is assumed that each node can be reached by the entry node and each node can reach the exit node.

The resultant graph is known as the program control graph, and the cyclomatic number is then a complexity measure for the program. This approach suggests that complexity is increased by branches and can be reduced by removing unnecessary branches. The complexity computation is

$$v = e - n + 2p.$$

McCabe observes that the cyclomatic complexity is equivalent to the number of predicates plus one.

Alternatively, if G is a connected graph with n vertices and e edges, then the cyclomatic complexity is equivalent to the number of regions:

$$v = e - n + 2.$$

For a strongly connected graph the cyclomatic complexity is

$$v = e - n + 1.$$

3. Basic method

Based on the choices of Witt's model and McCabe's measure, simple applications of McCabe's measure to some of the networks of Witt's model are made. Recognition of some shortcomings in this approach leads to definition of a new generalized measure, and similar application of it.

Table 1 Characteristics of software complexity measures.

	Easy to apply	*Intuitively obvious*	*Consistent*	*Suitable for networks*
McCabe	Yes	Yes	Yes	Yes
Source lines	Yes	Yes	No	No
Halstead	No	Yes	Yes	No
Storm/Preiser	Yes	Yes	Yes	No
Laemmel/ Shooman	Yes	Yes	Yes	No
Ruston	No	Yes	Yes	Yes

• *McCabe's model applied to networks*

In order to apply McCabe's technique to networks, the interpretation of the program control graph must be changed. Our interpretation is that each node in the network represents a module, and each edge represents an intermodule connection (invocation of or return of control from a program in another module). We call this the network control graph. The network control graph constructed in this manner represents the software system structure, as developed during the software architecture phase. McCabe's technique can then be applied to the network control graph in the same manner as it is applied to the program control graph.

Let us make this more formal.

Definition 10 A network control graph G is a directed graph with each node corresponding to a unique module and each edge corresponding to an intermodule connection. The edges are directed from the module using a network message to the receiving module.

Definition 11 A network control graph S is a subgraph of G if S is a network control graph which consists only of nodes and edges of G. If S is not identical to G, S is called a proper subgraph of G, written

$$S \subset G.$$

• *Simple applications of McCabe's measure*

Hierarchy and pipeline networks

We need the following definitions.

Definition 12 [4] A *hierarchy* is a network of modules in which output parameters and control are always returned to the invoking module.

Definition 13 [4] A *pipeline* is a network of modules in which output parameters and control are transmitted

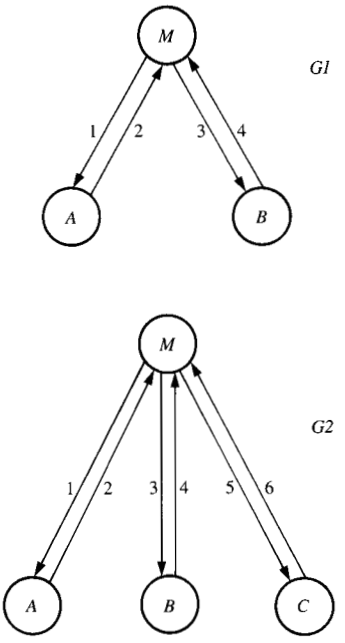

Figure 2 Hierarchy networks.

directly to successor modules, rather than being returned to the invoking module.

The hierarchy provides limited concurrency, by requiring that control be returned to the invoking module after each application program execution. The pipeline provides more potential for concurrency, insofar as many stages in the pipeline can be in execution concurrently.

The hierarchies examined have a monitor module M which controls the other modules in the network. In this context the monitor plays a supervisory role in the network. It is responsible for program invocation in the applications modules when they have work to do and for receiving control back when the work is complete. Consider the simple case where a monitor invokes a program in a module, and in turn has control returned to it. Figure 2 depicts two such cases.

First is a hierarchy with a monitor and two applications modules. The second is a hierarchy with a monitor and three applications modules. The cyclomatic numbers for the networks are, respectively,

$$V(G1) = 4 - 3 + 1 = 2,$$

$$V(G2) = 6 - 4 + 1 = 3.$$

This suggests that complexity increases as a function of the number of modules in the hierarchy. This result corresponds with our intuitive ideas about complexity.

The pipelines to be examined have a controlling program, which initiates program invocation in successor modules, and is executed at the control services level in Witt's model. At the end of the pipeline, for our purposes, control is returned to the initiating module. Figure 3 depicts two such cases.

The first is a pipeline with three modules, A, B, and C. The second pipeline has four modules, A, B, C, and D. The cyclomatic numbers for these networks are, respectively,

$$V(G1) = 3 - 3 + 1 = 1,$$

$$V(G2) = 4 - 4 + 1 = 1.$$

This suggests that all pipelines have the same complexity. This result is not altogether satisfactory, as intuition suggests that complexity increases with the number of modules. This problem is addressed in the section on the generalized measure.

The results of applying the McCabe measure can be generalized as follows.

Theorem 3 In a hierarchy of n modules (including the monitor), the McCabe network complexity is $n - 1$.

Proof Suppose there are n modules in the network. Then there are $2 \times (n - 1)$ paths connecting the modules, so that

$$V(G) = 2 \times (n - 1) - n + 1$$

$$= n - 1.$$

Theorem 4 In a pipeline of n modules, the McCabe network complexity is always 1.

Proof Given a pipeline G of n modules, observe that there are n paths connecting them, so that

$$V(G) = n - n + 1$$

$$= 1.$$

Definition and application on generalized measure

The method described in the section on McCabe's measure is readily used for evaluation of methods of transfer of control, static networks, and data acquisition. It was felt, however, that there were other resources of interest in complexity measurement, such as channels, tape drives, disk drives, CPUs, etc. In fact, resource allocation is commonly addressed in operating system designs by various scheduling algorithms, queueing techniques, interrupt handlers, etc. Proposed operating systems are routinely modeled and examined to ensure that they contain simple, low-overhead designs. The methods used in simulating operating system designs are somewhat different from the methods we discuss here. In order to address other resources of interest, a more general measure was developed which is also based on path expressions. It should be pointed out that this measure is

not a generalization of the McCabe measure, although certain elements of it are similar.

Definition

Here we define a network N consisting of nodes and edges, as follows: Each node represents a module, as before, which may or may not be executed concurrently with another module. Each edge represents program invocation and return between modules (in this case the edges are called single paths). Resources are allocated when programs are invoked in other modules, and we wish to measure complexity associated with allocation of such resources.

Given a network N of n single paths, define complexity as

$$C(N) = \sum_{i=1}^{n} \left(e_i + \sum_{j=1}^{k} d_j \times r_{ji} \right),$$

where

1. k is the total number of resources to be controlled in the network,
2. $r_{1i}, \cdots, r_{ki}$ represent those resources which must be controlled for a given path p_i,
3. $r_{ji} = 0$ if the resource is not required by the node,
   $r_{ji} = 1$ if the resource is required,
4. d_j is the complexity for allocation of each resource (e.g., the complexity associated with a procedure used to gain exclusive access to common data),
5. e_i is the complexity of program invocation and return along each path p_i (such as operating system complexity).

Simple applications of general measure

Let us apply this technique to the simple kinds of hierarchy and pipeline networks that we did for the McCabe model. For the purpose of these applications, let us assume that only program invocation and return are of interest, and that there are no resources to be controlled.

This means that $r_{ji} = 0$ in all cases. If we assume only one means of invocation, then e_i = constant, say, $e_i = 1$ for all i.

Then for the hierarchies depicted in Fig. 2,

$C(G1) = 4$ and $C(G2) = 6$.

Similarly, for the pipelines depicted in Fig. 3,

$C(G1) = 3$ and $C(G2) = 4$.

In addition the following general results are obtained.

Theorem 5 If e is the complexity associated with program invocation between modules and if no other resources are needed, then for a hierarchy N of n modules the generalized complexity is $2e \times (n - 1)$.

Proof Note that for a hierarchy of n modules there are $2 \times (n - 1)$ paths connecting them. Since there are no other resources required, the generalized complexity is

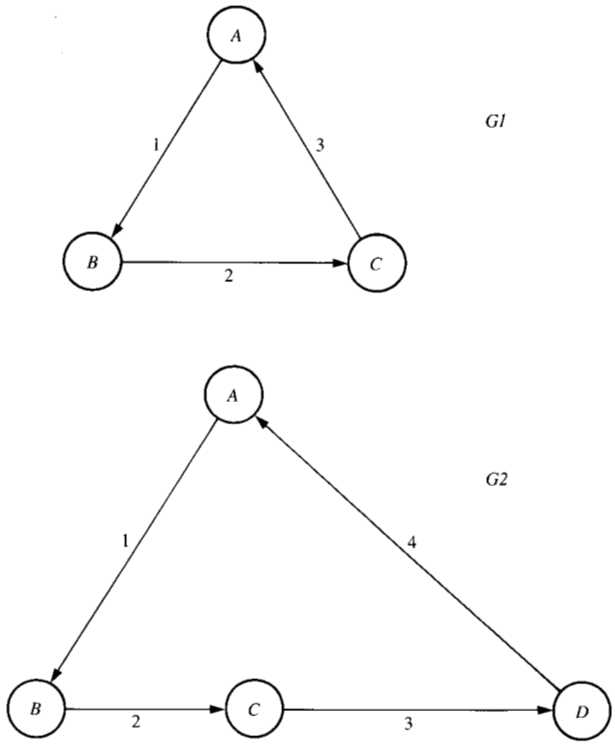

Figure 3 Pipeline networks.

$C(N) = 2e \times (n - 1)$.

Theorem 6 If e is the complexity associated with program invocation between modules and if no other resources are needed, then for a pipeline N of n modules the generalized complexity is ne.

Proof Note that for a pipeline of n modules there are n paths connecting them. Since there are no other resources required, the generalized complexity is

$C(N) = ne$.

To compare these results with McCabe's results, let $e = 1$. Then $C(N) = 2 \times (n - 1)$ for the hierarchy versus $V(G) = n - 1$, and $C(N) = n$ for the pipeline versus $V(G) = 1$.

Note that using this more general approach, we still have the result that pipelines are less complex than hierarchies with the same number of modules. However, pipeline complexity increases with the number of modules. This is a different result from the result obtained in the section on McCabe's measure, where all pipelines had the same complexity.

The difference occurs because the general-purpose measure counts the number of edges and does not subtract the

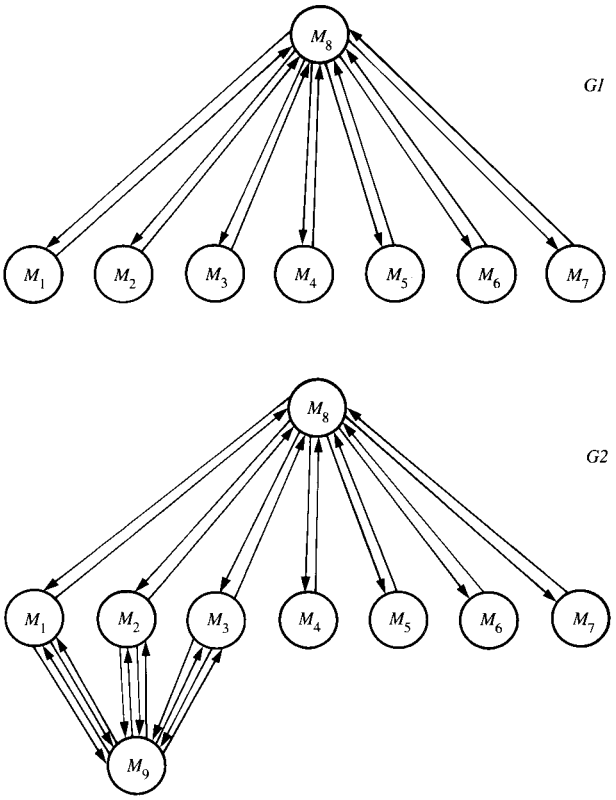

Figure 4 Data locking networks.

number of nodes. Thus with the generalized measure, complexity is a function of the total number of module invocations.

There is some disagreement as to which of these two approaches is preferable. On the one hand, one can argue that only excessive numbers of invocations are of interest, and conclude that the McCabe approach is preferable. On the other hand, one can argue that more modules in a network lead to more complexity, and conclude that the generalized measure is preferable.

With either technique there is a need to combine the network complexity with internal module complexity to obtain an overall complexity measure, called the combined complexity measure.

4. Combined measure

In examining the complexity of the network only, we are able to obtain a measure of complexity for one network design relative to another. There is, however, a danger in using the network complexity measure exclusively for these types of decisions. For example, the least complex network would result if the entire system consisted of a single module. The resultant module would then have all the network complexity embedded in the module, so that it would be hidden from the network measure, but visible to some of the more conventional complexity measures, such as were examined previously. One could also consider the situation which would result if modules to perform service functions, such as exclusive data access and release, did not exist, but the logic was embedded in the applications modules. The network would be less complex, but the applications modules would be more complex. It is therefore desirable to be able to combine the network and conventional complexity measures in such a way that a more complete picture is obtained. This combined complexity is more of an absolute measure, as it is reasonable to expect the complete system to have an overall complexity threshold which should not be exceeded.

- *Problem description*

Suppose a network consists of a monitor and seven applications modules, of which three require exclusive access to the same data. We want to evaluate whether it is preferable to have a separate lock/unlock module, or whether it is preferable to embed this logic in the three applications modules. The networks representing these alternatives are shown in Figure 4. It is assumed that communications modules are not a factor in the decision process, so these are not shown in this example. Here module M_8 is the monitor, and modules M_1 to M_7 are the applications modules. Module M_9 provides the locking and unlocking capabilities needed by modules M_1, M_2, and M_3. Network *G1* shows the locking/unlocking capability embedded in the applications modules, and network *G2* shows the locking/unlocking capability in a distinct module.

In order to develop a combined complexity measure, the following definition is used.

Definition 14 Let CN be the network complexity and $C_1, \cdots, C_k$ be the individual module complexities. Then the combined complexity is

$$C_T = w_1 \times CN + w_2 \sum_{i=1}^{k} C_i ,$$

where w_1 and w_2 are weighting factors assigned by the user, which may differ for different measurement techniques.

- *Applications of McCabe/generalized measures*

For this example we apply both the McCabe and the generalized measures to the networks shown in Fig. 4. The results for the McCabe measure are

$$V(G1) = 14 - 8 + 1 = 7.$$

For this network the combined complexity is

$$C_{T1} = 8 \times w_1 + w_2 \sum_{i=1}^{8} C_{i1} .$$

Using the McCabe measure,

$$V(G2) = 26 - 9 + 1 = 18.$$

For this network the combined complexity is

$$C_{T2} = 19 \times w_1 + w_2 \sum_{i=1}^{9} C_{i2}\,.$$

At this point some observations are made to simplify computations. We need to determine which of networks *G1* or *G2* provides a less complex design. Therefore, we want to compare C_{T1} and C_{T2}. Observe, however, that many elements are the same for both *G1* and *G2*. For example, modules M_4 through M_8 in Fig. 4 are identical in *G1* and *G2*. Therefore, to examine the differences between C_{T1} and C_{T2}, it is not necessary to compute the complexity of M_4 through M_8. In addition, those sections of M_1 through M_3 which do not relate to data acquisition or locking can also be disregarded. Therefore, only the difference will be examined:

$$C_{T1} - C_{T2} = -11 \times w_1 + w_2 \left(\sum_{i=1}^{8} (C_{i1} - C_{i2}) - C_{92} \right).$$

Observe that $C_{i1} = C_{i2}$ except for the three applications modules M_1, M_2, M_3 and the lock module M_9. Therefore,

$$C_{T1} - C_{T2} = -11 \times w_1 + w_2((C_{11} - C_{12}) + (C_{21} - C_{22}) + (C_{31} - C_{32}) - C_{92}).$$

Further observe that

$$C_{j1} - C_{j2} = (C_{j2} + C_{91}) - C_{j2}$$

for $j = 1, \cdots, 3$, where C_{91} is the average added complexity when locking/unlocking is done in the applications module; then,

$$C_{T1} - C_{T2} = -11 \times w_1 + w_2(3 \times C_{91} - C_{92}).$$

Note that in applying this technique to an actual design problem, the user might make a different set of assumptions than we have made, based on the design problem.

Alternatively, apply the generalized measure to the networks of Fig. 4 using the following algorithm:

$$C(N) = \sum_{i=1}^{n} \left(e_i + \sum_{j=1}^{k} d_j \times r_{ji} \right).$$

For this example let the cost of branching $e_i = 1$, and note that $r_{ji} = 0$, except for locking and unlocking, which have complexities d_1 and d_2. Then

$$C(N) = \sum_{i=1}^{n} 1 + d_1 + d_2$$

and

$$C_{T1} = 14 \times w_1 + w_2 \sum_{i=1}^{8} C_{i1}\,,$$

$$C_{T2} = 26 \times w_1 + w_2 \sum_{i=1}^{9} C_{i2}\,.$$

Thus

$$C_{T1} - C_{T2} = -12 \times w_1 + w_2(3 \times C_{91} - C_{92}).$$

Having obtained these results for the two measurement techniques, we must measure the complexity of the locking/unlocking function, both in a separate module and embedded in the applications modules. We must also assign an appropriate weighting factor in order to complete the trade-off analysis.

- *Combined measure*

Storm and Preiser's index of complexity/McCabe's measures Let us now turn to examine the logic of a typical locking/unlocking routine, and apply Storm and Preiser's measurement technique.

This index of complexity was developed especially for structured programs. Given the complexity index for structured programming constructs such as sequence (DO-THEN), iteration (WHILE-DO), alternation (IF-THEN-ELSE), and CASE, the complexity index of the entire program is developed.

Some of Storm and Preiser's definitions [9] are used here.

Definition 15 [9] Let $m(s_i)$ be the measure of complexity of the program segment s_i, which has ℓ_i lines of source code, so that

$$m(s_i) = \ell_i\,.$$

Let $m(c_i)$ be the measure of complexity of the test on condition c_i, so that

$$m(c_i) = d_i\,, \qquad d_i \geq 1.$$

Definition 16 [9] For the sequence (Do s_1 then s_2)

$$m(s_1) + m(s_2) = \ell_1 + \ell_2\,.$$

Definition 17 [9] For the iteration (While c_1 do s_1)

$$m(c_1) + m(s_1) = d_1 + \ell_1\,.$$

Definition 18 [9] For the alternation (If c_1 then s_1 else s_2)

$$m(c_1) + \text{avg}(m(s_1),\ m(s_2)) = d_1 + (\ell_1 + \ell_2)/2.$$

Definition 19 [9] For the case statement

$$m(c_1) + \text{avg}(m(s_1),\ m(s_2),\ \cdots,\ m(s_n)) = d_1 + \sum_{i=1}^{n} \ell_i/2.$$

For example, the simple structure in Fig. 5 has a complexity of

$$m(c_1) + \text{avg}(m(s_1),\ m(s_2)).$$

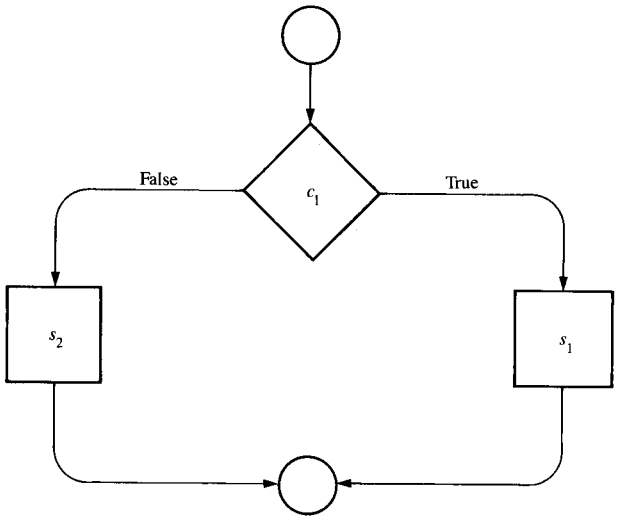

Figure 5 Storm and Preiser measure of complexity.

Note that each prime structure is assigned an individual complexity m. These are then combined to give an overall complexity for the composite program.

Since this is easily computed before compilation, if the index is "too large," some care in reducing the complexity is strongly suggested. In this case, design logic will be examined.

Apply Storm and Preiser's index as follows:

"If" constructs have a value of 1.
"Do" constructs have a value of 1.
Each line in a sequence has a value of 1.

Then for the exclusive lock design in Table 2,

$$m(\text{lock}) = 4 \times (m(\text{if})) + m(\text{dountil}) + 13 = 18.$$

The design for exclusive unlock is shown in Table 3.

Now we apply Storm and Preiser's measure to obtain

$$m(\text{unlock}) = 5 \times (m(\text{if})) + m(\text{whiledo}) + m(\text{dountil}) + 12 = 19.$$

Alternatively, apply McCabe's measure to the combination of designs for the lock and unlock functions and obtain $V(G) = 12$. The results for the exclusive lock and unlock designs can now be used in conjunction with the network complexity measures to obtain an overall complexity figure, which will allow the desired trade-off study to be done.

This example can be studied to see if more general comparisons can be made of the various measures used here.

For the lock design observe that the Storm and Preiser index is

$$m(\text{lock}) = 4 \times m(\text{if}) + m(\text{dountil}) + 13 = 18.$$

For the McCabe measure $V(G) = 5$, and finally the number of source lines is 26. This suggests the following relationship:

number of source lines $\geq$ Storm and Preiser index $\geq$ McCabe measure.

This occurs because the McCabe measure counts only the design statements, while Storm and Preiser's index counts the decision statements plus some of the source lines, and of course the source line count includes everything.

- *Combined measure results*

In order to combine the results of the last two sections, first note that the complexity of the lock/unlock design is the same regardless of whether the design is included within the module or as a separate module. This suggests that for the example C_{91} and C_{92} are identical. Since only the lock or the unlock portion of the design is used at any given time, the indices computed previously can be averaged to obtain

$$C_{91} = C_{92} = (18 + 19)/2 = 18.5.$$

For the McCabe network measure,

$$C_{T1} - C_{T2} = -11 \times w_1 + 37 \times w_2 .$$

Let $w_1 = w_2 = 1$ to obtain

$$C_{T1} - C_{T2} = 26.$$

For the generalized measure,

$$C_{T1} - C_{T2} = -12 \times w_1 + 37 \times w_2 = 25.$$

Note that in both cases the network with an independent locking/unlocking module is less complex. This corresponds to our intuitive notions for this particular case. It is possible, however, to get more general results, as will be seen later.

Alternatively, by using the McCabe measure for design in combination with the McCabe and generalized network measures a different set of results is obtained.

Combine the McCabe measures for design and network to obtain

$$C_{T1} - C_{T2} = -11 \times w_1 + w_2(36 - 12) = -11 \times w_1 + 24 \times w_2 .$$

Let $w_1 = w_2 = 1$; then

$$C_{T1} - C_{T2} = 13.$$

Table 2 Exclusive lock design.

```
If the user has not made a lock request
or if this module does not have this lock block
  terminate
Endif
If this module already has this lock block
  terminate
Endif
Save the return address to the user module
Obtain the lock
Store the time lock was received
Add this lock to the queue of those owned by
this module
Store the return address
Dountil test-and-set is successful
  Expect a 0 in the lock
  Set the user's module id in the lock
  Attempt to set the lock to the user's
    module id
  If test-and-set is successful
    The user module has the lock
  Else
    Add the user module to the queue
    of those waiting
  Endif
Enddo
Save the return address
Store the time of the lock completion
```

Table 3 Exclusive unlock design.

```
If this is not an unlock request
or if this module does not have this lock block
  Terminate
Endif
If the owner of the lock block is not this module
  Terminate
Endif
Get the queue of locks owned by this module
While
  this is not the lock
Do
  Get next lock on queue
  If this is the end of the queue
    Terminate
  Endif
Enddo
Remove the lock from this module's lock queue
If no other modules are waiting
  Release the lock
  Return if test-and-set is successful
Endif
Dountil the end of the queue of modules waiting
  Find the end of the queue
Enddo
Give the lock to the module id
Update pointers
Save return address
```

This indicates that it is advantageous to have a separate locking/unlocking module when there are three users. If there are only two users, with control graph *G3*, then

$$C_{T1} - C_{T3} = -7 \times w_1 + w_2(24 - 12) = 5.$$

In this case it is still advantageous, although less so, to have a distinct locking/unlocking module. Finally, if there is just one user, with control graph *G4*, then

$$C_{T1} - C_{T4} = -3 \times w_1 + w_2(12 - 12) = -3.$$

In this case it is clearly advantageous to not have a separate module for locking/unlocking.

By using the generalized measure for networks and the McCabe measure for design, a similar set of results is obtained. For the network where three modules require the service,

$$\begin{aligned} C_{T1} - C_{T2} &= -12 \times w_1 + w_2(36 - 12) \\ &= -12 \times w_1 + 24 \times w_2 \\ &= 12. \end{aligned}$$

When two modules require the service,

$$\begin{aligned} C_{T1} - C_{T3} &= -8 \times w_1 + w_2(24 - 12) \\ &= -8 \times w_1 + 12 \times w_2 \\ &= 4. \end{aligned}$$

Finally, when one module requires the service,

$$\begin{aligned} C_{T1} - C_{T4} &= -4 \times w_1 + w_2(12 - 12) \\ &= -4 \times w_1 \\ &= -4. \end{aligned}$$

These results give an indication of whether it pays to have a distinct service module for locking/unlocking. In the case of two or more modules it is clearly worthwhile. For one module it is definitely not worthwhile. Of course, for one module a locking service is not needed, but the example is only used for illustrative purposes. Therefore, this example illustrates how the network measure can be used early in the design phase to help make design trade-off decisions.

A more general result can be obtained for the McCabe network measure and McCabe design measure combined.

Theorem 7 Let *G1* be the network with service design embedded in the applications modules. Let *G2* be the network with service design in a distinct service module. Then if C_{T1} and C_{T2} are the combined complexities for *G1* and *G2*, respectively,

$$\begin{aligned} C_{T1} - C_{T2} &= -(\text{number of added edges} - 1) \\ &\quad + (\text{number of users} - 1) \\ &\quad \times \text{design complexity} \\ &= -(es - 1) + ((ua - 1) \times dc). \end{aligned}$$

Here *es* is the number of added edges needed to access the service, *ua* is the number of applications modules using the service, and *dc* is the design complexity of the service.

If

$C_{T1} - C_{T2} < 0$, a separate service module is desirable,

$= 0$, the service can be embedded in the applications modules or kept as a separate service,

< 0, a separate service module is not desirable.

The following general result is obtained.

Theorem 8 Given a hierarchy with two or more applications modules using a specific service, it is worthwhile to have a separate service module S if $C(S) > 3$ using the McCabe measure, or if $C(S) > 4$ using the generalized measure for the network measure (assuming we let the cost of transfer $= 1$ for the generalized measure).

Proof (McCabe measure) Suppose graph *G1* corresponds to the hierarchy of three modules with two applications modules having the service embedded. Then

$$V(G1) = 2 + 2 \times C(S).$$

Then suppose graph *G2* corresponds to the hierarchy of three modules with two applications modules and a separate service module:

$$V(G2) = 5 + C(S),$$

$$V(G1) > V(G2) \text{ if } C(S) > 3.$$

In general, for hierarchies of n modules, with $n - 1$ application modules,

$$V(G1) = n - 1 + (n - 1) \times C(S),$$

$$V(G2) = 3 \times (n - 1) - 1 + C(S),$$

$$V(G1) > V(G2) \text{ if } C(S) > 2 + 1/(n - 2),$$

which can be rounded up so that it is worthwhile to have a separate service module if $C(S) > 3$.

Proof (generalized measure) Suppose graph *G1* corresponds to the hierarchy of three modules with two applications modules having the service embedded. Then

$$C(G1) = 4 + 2 \times C(S).$$

Then suppose graph *G2* corresponds to the hierarchy of two modules with a separate service module:

$$C(G2) = 8 + C(S),$$

$$C(G1) > C(G2) \text{ if } C(S) > 4.$$

In general, for hierarchies of n modules with $n - 1$ applications modules, where $n > 2$,

$$C(G1) = 2 \times (n - 1) + (n - 1) \times C(S),$$

$$C(G2) = 4 \times (n - 1) + C(S),$$

$$C(G1) > C(G2) \text{ if } C(S) > 2 + 2/(n - 2),$$

which can be rounded up so that it is worthwhile to have a separate service module if $C(S) > 4$.

5. Conclusions and future research directions

In the previous sections techniques have been developed for measuring complexity of networks of modules, including

1. An extension of McCabe's technique which can be used for networks.
2. The generalized measure, which can be used to measure the complexity associated with resource acquisition, as well as for network measurement.
3. The combined measure, which aids in deciding whether to develop separate service modules or whether to keep service logic within the applications modules.

A more extensive treatment of these topics can be found in Hall's doctoral dissertation [20], which is the basis for this paper.

It is our view that these techniques are generally intuitive, easy to apply, and provide assistance in assessing the many design alternatives available to the architect of a complex system. They allow one to examine complexity with a consistent, plausible *a priori* method, and can be used to

1. Identify areas which are overly complex (in a relative sense) so that complexity can be reduced before major design decisions have been made.
2. Help decide on the optimal number of modules in the system.
3. Measure complexity associated with resource acquisition.
4. Aid in deciding whether to have distinct service modules.

Methods for reducing network and overall complexity include

1. Encapsulation of data to define the optimum number of modules and reduce locking overhead and complexity.
2. Use of pipelining where possible to reduce transfer complexity.
3. Use of service modules to reduce combined complexity.

In the future the following efforts could be undertaken:

1. Examination/development of tools to aid in computing complexity for large systems.

2. Extension of these results to include other methods of modeling parallel processes (e.g., Petri nets, communicating sequential processes).
3. Extension of this type of measure to other phases of the software life cycle (e.g., software specifications, testing).

It is particularly desirable to continue research in this area, so that software complexity measurement can become more rigorous. It is our view that the work described here aids in progress towards this goal.

6. References

1. R. M. Mattheyses and S. E. Conry, "Models for Specification and Analysis of Parallel Computing Systems," *1979 Conference on Simulation, Measurement and Modeling of Computer Systems*, ACM SIGMETRICS **8,** 3, 215–224 (Fall 1979).
2. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM* **21,** 8, 666–677 (1978).
3. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Commun. ACM* **19,** 5, 279–285 (1976).
4. B. I. Witt, "Communicating Modules: A Design Model for Concurrent Distributed Systems" *FSD TR86.0001, IBM Federal Systems Division*, Bethesda, MD, August 13, 1982.
5. M. R. Woodward, M. A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," *IEEE Trans. Software Engineering* **SE-5,** 1, 45–50 (January 1979).
6. L. A. Belady and C. J. Evangelisti, *Research Report RC-7560*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, March 1979.
7. L. A. Belady, "On Software Complexity," *IEEE Proceedings of the Workshop on Quantitative Software Models*, No. TH0067-9, New York, October 1979, pp. 90–94.
8. H. Ruston, "Software Modeling Studies: The Polynomial Measure of Complexity," *RADC-TR-81-183*, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, Rome, NY, July 1981.
9. I. L. Storm and S. Preiser, "An Index of Complexity for Structured Programming," *IEEE Proceedings of the Workshop on Quantitative Software Models*, New York, 1979, pp. 130–133.
10. T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Engineering* **SE-2,** 4, 308–320 (December 1976).
11. M. H. Halstead, *Elements of Software Science*, Operating and Programming Systems Series, P. J. Denning, Ed., Elsevier North Holland Co., Inc., New York, 1977.
12. C. E. Walston and C. P. Felix, "A Method of Programming Measurement and Estimation," *IBM Syst. J.* **16,** 1, 54–73 (1977).
13. A. Laemmel and M. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," *POLY/EE/E0-76-020*, Dept. of Electrical Engineering and Electrophysics, Polytechnic Institute of New York, Brooklyn, NY, August 15, 1977.
14. K. Christensen, G. P. Fitsos, and C. P. Smith, "A Perspective on Software Science," *IBM Syst. J.* **20,** 4, 372–387 (1981).
15. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," *ACM Computing Surv.* **10,** 1, 3–18 (March 1978).
16. M. H. Halstead, R. D. Gordon, and J. L. Elshoff, "On Software Physics and GM's PL.I Programs," *GM Research Publication GMR-2175*, General Motors Research Laboratories, Warren, MI, 1976.
17. J. L. Baer, "Graph Models in Programming Systems," K. M. Chandy and R. T. Yeh, Eds., *Current Trends in Programming Methodology*, III, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978, 168–230.
18. C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Co., Amsterdam, The Netherlands, 1973.
19. W. Mayeda, *Graph Theory*, Wiley-Interscience, New York, 1972.
20. N. R. Hall, "Complexity Measures for Systems Design," Doctoral Dissertation, Dept. of Mathematics, Polytechnic Institute of New York, NY, June 1983.

Received May 5, 1983; revised July 7, 1983

Nancy R. Hall *IBM Federal Systems Division, 6600 Rockledge Drive, Bethesda, Maryland 20817.* Dr. Hall received her B.A. and M.S. in mathematics in 1963 and 1967 at New York University and her Ph.D. in mathematics in 1983 at the Polytechnic Institute of New York. She joined IBM in 1966 in New York City and has since worked in the areas of software development, software management, and software technology. Currently she is an advisory programmer on the software engineering and technology staff at the Federal Systems Division headquarters. Her recent activities include development of software network measurement techniques, teaching classes in distributed systems design at the Federal Systems Division, and assisting George Mason University, Fairfax, Virginia, in the development of a computing strategy. Dr. Hall is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers.

Stanley Preiser *Polytechnic Institute of New York, 333 Jay Street, Brooklyn, New York 11201.* Dr. Preiser is professor of mathematics and computer science at the Polytechnic Institute of New York and formerly Dean of its Westchester Center in White Plains, New York. He received his B.S. in physics from the City College of New York in 1949, and his M.S. in 1950 and his Ph.D. in 1958, both in mathematics from the Courant Institute at New York University. Dr. Preiser has been a consultant to IBM and an invited lecturer over a period of years. He is a member of the American Association for the Advancement of Science, American Mathematical Society, Association for Computing Machinery, and Society for Industrial and Applied Mathematics.