1981

COMPLEXITY MEASURES IN SYSTEM DEVELOPMENT

Benn Konsynski
University of Arizona

Jeff Kottemann
University of Arizona

COMPLEXITY MEASURES IN SYSTEM DEVELOPMENT

Benn Konsynski
Management Information Systems
University of Arizona

Jeff Kottemann
Management Information Systems
University of Arizona

ABSTRACT

Complexity measurement algorithms for information systems schemas are considered. Graph representations, based on an object-relation pardigm and linguistic models, are discussed. Software science metrics are evaluated as complexity measures, as is the cyclomatic complexity measure. The deficiencies of current measures are highlighted. An alternative structural complexity metric is proposed that reflects propagation effects. The system development life cycle is used to determine realms of complexity that provide a framework for evaluation of complexity of designs and for projecting complexity between system development life cycle phases.

INTRODUCTION

Complexity as Measurement Information

Decisions are based on information, value systems, and evaluation procedures. Information is frequently presented in terms of "measurements" that communicate such things as "status," "environment," "performance," etc. Measurements reflect the "values" of the observer (individual or system), and may bias the decision making process and consequently the resulting decision. This paper addresses complexity issues related to the decision making process in the information system development process. The intent of the discussion is to explore complexity measurements that may be useful in the information system design process. The emphasis of the inquiry is on the technical system characteristics, both static and dynamic. Nevertheless, many of the issues and concerns presented are applicable to important behavioral issues.

The Nature of Complexity

Complexity is a phenomenon of observation. It is easily recognized but difficult to formally define. One feels that he can make relative comparisons and partial orderings, yet explicit quantification eludes us. If one were to ask an automobile mechanic, "What is complexity?" she may well answer, "The complexity of what?" If one were to ask her, however, "What is more complex, a carburetor system or a fuel-injection system?" she would answer, "A fuel-injection system." Asked why, she might say, "Because when I have to trouble-shoot a fuel-injection system, it usually takes me twice as long."

It is less difficult to address complexity when we introduce a context of evaluation, a viewpoint, and focus for observation. By providing a context (i.e., by specifying certain tasks to be performed with certain objects), the meaning of complexity becomes more formal, and one can determine the relative difficulty of performing tasks. Furthermore, the measurement of relative difficulty can be made in terms of a unit that is meaningful in the context of the application (e.g., time to do the task). Complexity is closely allied to the notion of difficulty. Within a given context, complexity is recognizable as a measurable cost. The determination of the contexts of system complexity and the respective cost functions is one tool in the process of moving program/system design from (basically) an art to (basically) a science, or at least an engineering discipline.

Context Dependency in Complexity Assessment

In order to determine the complexity of a system, a context or perspective must be adopted. The measures reflect the values and assertions associated with a context adoption.

Graph representations are often used in modeling software systems, communication networks, projects, etc. Graph schemata of programs are used to support global optimization, the development of testing strategies, etc. Graphic representation of programs/systems can serve as a basis for complexity assessment in program/system design. The conclusions that can be drawn based on the analysis of a graph model, however, are contingent on the system being modeled; the context.

The graph in Figure 1 depicts a structure that converges along the direction of flow, while the graph in Figure 2 diverges. The distinctions that can be drawn between these two graphs is dependent upon the context of review. If the graphs are used in the context of a software system, the nodes are discrete programs. If the graphs are used to represent programs, then the node-arc pairs might represent control transfer points or some other inter-process relation. In the convergent graph of Figure 1, the complexities are dependent upon context. As a program flow graph, the figure depicts the transfer of control into sinks. Therefore, the current program state becomes simpler, that is, state information that was relevant in the initial choice of control flow path may well become unimportant as the paths converge. Indeed, this convergence to a single exit point is a major principle in structured programming. Although a convergent structure simplifies program control flow, it complicates overall system flow. A convergent system flow results in scheduling problems due to precedence. A schedule slippage along any path will result in a propagation effect throughout the entire system. In contrast to a convergent program flow, all paths must be considered simultaneously in a system. This simultaneity enters into system complexity in a combinatorial fashion. In general, convergence simplifies a program structure but complicates system structures.

In contrast to convergence, the divergent flow of Figure 2 is a complicating structure in programs, while it is simplifying in large systems. In a program, divergence is a process of path selection. The state space relevant to the current position along a path grows larger as the divergence process continues. Each decision is based upon prior states and thus the complexity (as a function of the state space) increases. Divergent flows in a system, on the other hand, are simplifying; they allow for scheduling flexibility. A slippage in the execution of a discreet task (program) will affect only those tasks which are in the transitive closure from that point. This scheduling flexibility is frequently used in the exploitation of parallel processing.

A Context Independent Basis of Complexity

In the context of project management, complexity has three basic aspects:

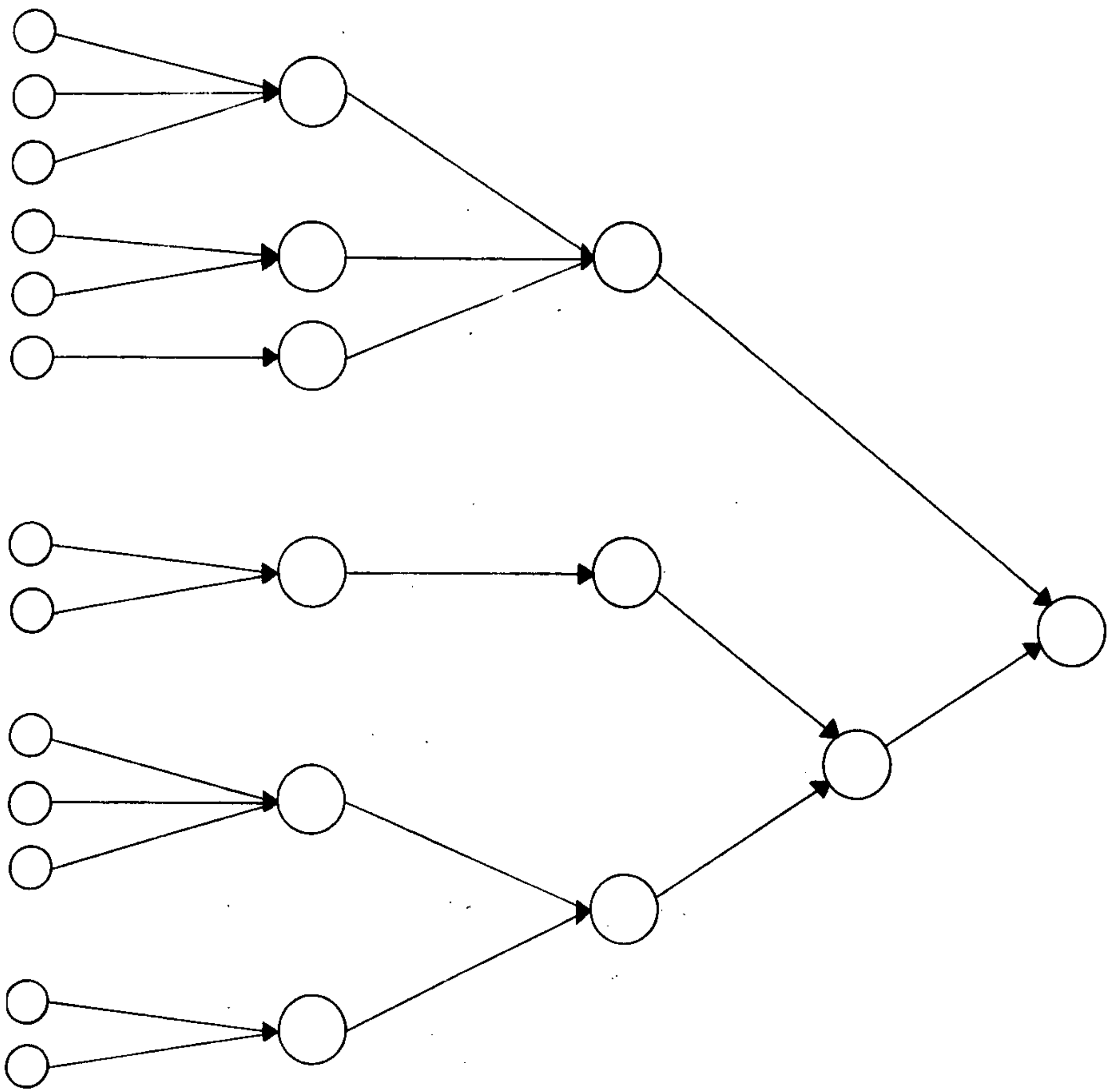


Figure 1. A Convergent Structure

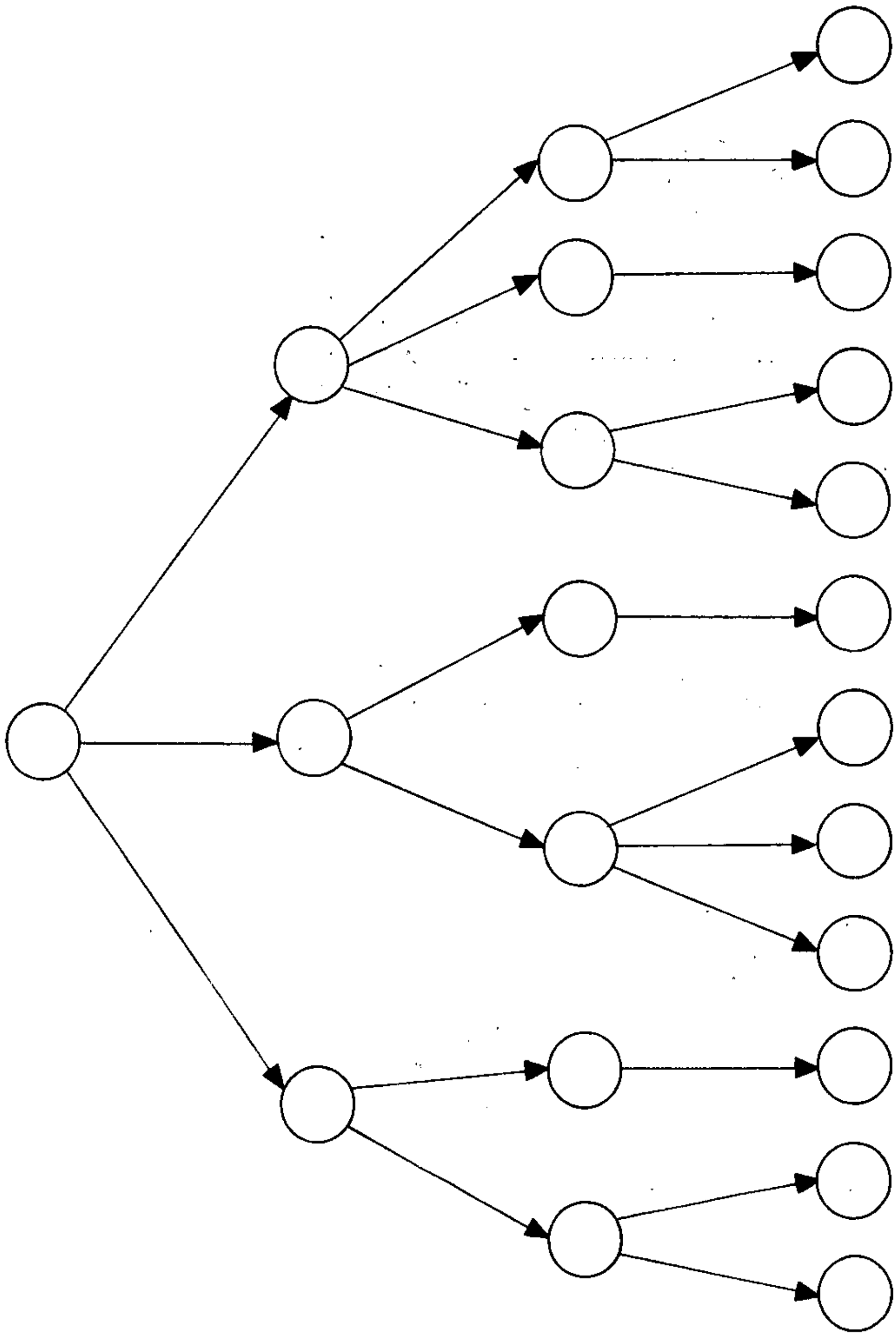


Figure 2. A Divergent Structure

1. The complexity of the totality of individual tasks.

2. The complexity of the interrelationship of tasks.

3. The complexity (volatility) of the environment within which the project is being conducted.

Task complexity is represented by the amount of operational resources (materials, man hours, etc.) necessary to complete the tasks. Task interrelatedness complexity is manifest as the amount of support resources (e.g., project manager hours) necessary to complete the project. Environmental complexity is a higher level complexity concern than are task and task interrelation complexities. At this higher level, the project is viewed as a task, and the interrelation of tasks is a function of organizational forces. Factors such as project priority and resource allocation which are set at the higher, organizational level, comprise the environmental complexity of individual projects. A combination of the three aspects of complexity indicate the overall complexity of a project (process) from a management standpoint. These three complexities will, in general, be evident in all systems of processes.

Despite the role of specific contexts in complexity assessment, a general property framework of complexity factors exists. The context dependencies of complexity stem from the realization of the three types of complexity presented above. Two graph structures are presented in Figures 3 and 4. Whether the graph structures represent projects, software systems, or programs, several context independent observations can be made. The graph of Figure 3 has eight paths. The graph of Figure 4 has eleven. Therefore, most complexities which arise due to the number of paths will apply regardless of the context. If graph 3 represents a project or a software system, timing considerations are easier to derive.

Analyses such as determining the shortest path and assessing the impact of a schedule slippage are easier to ascertain. If Figure 3 represents a program flow graph, path driven analyses will be easier. Examples of such analyses include global optimization and exhaustive program testing. These context independent factors are valuable in the initial formulation of a complexity model. They are, however, general and neglect many critical context dependent factors. One must always be alert to augmentation of complexity measures with appropriate context dependencies.

AN IS COMPLEXITY MODEL FRAMEWORK

The system life cycle is one useful model of system development and evolution. A representative system life cycle is shown in Figure 5.

For purposes of this discussion, we have partitioned the life cycle into four sets which we call complexity realms denoted by R. The choice of realm partitions is based upon temporal and operational relationships between life cycle phases.

R1 = (1,2)Requirement definition and logical design

R2 = (3,4,5,6)Physical design through implementation

R3 = (7)System performance

R4 = (8)Maintenance and modification

R1 deals with the complexity of capturing the requirement specifications in a complete and consistent logical design. R2 involves the complexity of designing, constructing, and implementing a physical design which is complete and consistent with

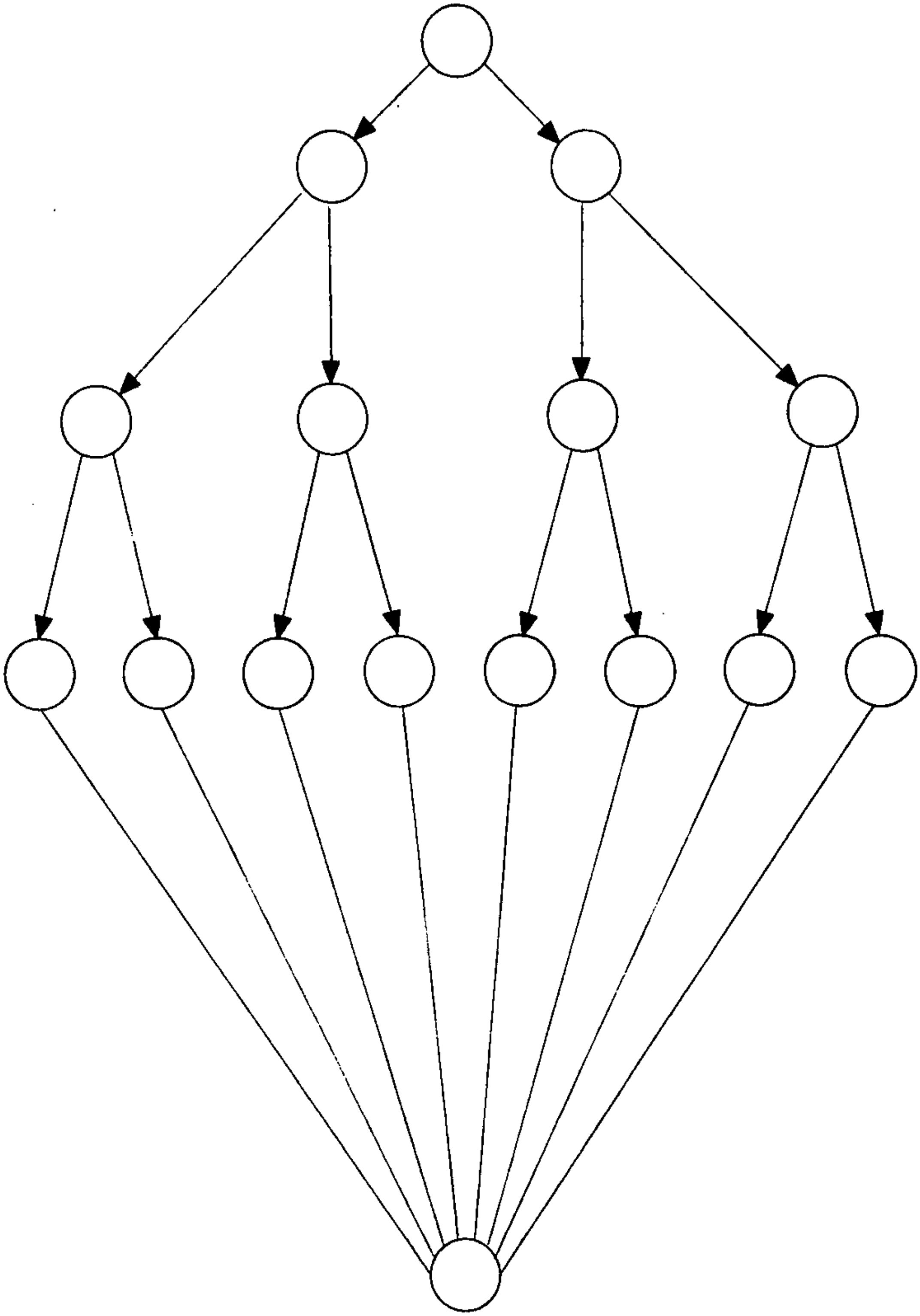


Figure 3. A Structure with Eight Paths

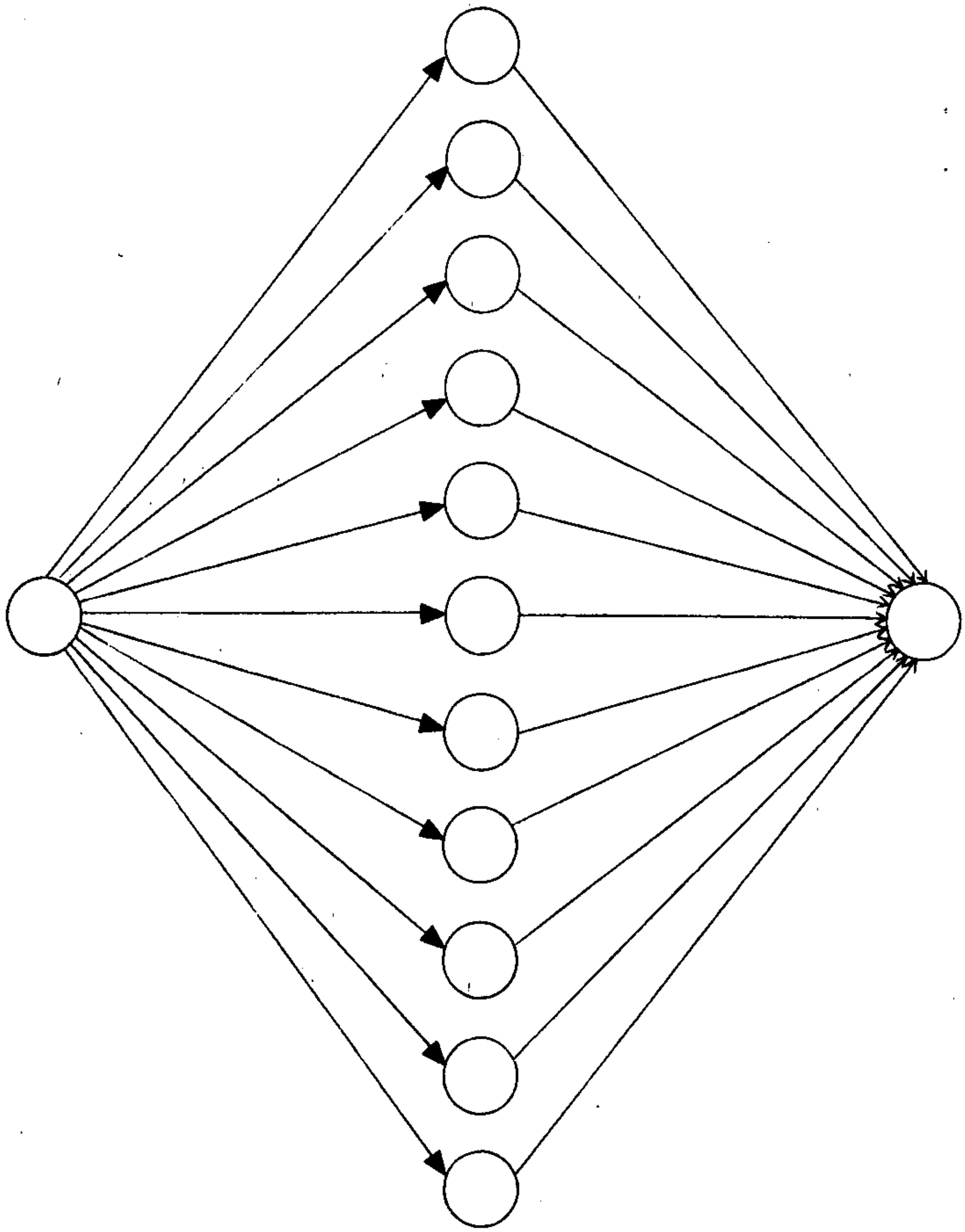


Figure 4. A Structure with Eleven Paths

1. Requirements Definition
2. Logical Design
3. Physical Design
4. Construction
5. Testing
6. Implementation
7. Operation
8. Maintenance

Figure 5. A System Life Cycle

the logical design. R3 is concerned with the complexity with respect to overall operating efficiency of the object system. In the context of a program, R3 is akin to the notion of computational complexity. R4 addresses the flexibility inherent in the system implementation.

Partitioning the system development life cycle into realms is justified for several reasons. First, the sets of attributes applicable to the different realms are to some extent disjoint. In other words, some attributes are relevant to only a subset of the realms. Separation into realms takes into account the context dependencies of complexity. Second, several attributes span realms. This spanning phenomenon stems from context independencies and may result in conflicting objectives, e.g., minimizing complexity in R3 implies a suboptimal level of R2 and R4 complexity, minimizing R4 complexity implies a suboptimal level of R2 and R3. Separation of the life cycle into realms assists the designer/manager in identifying attributes which cause conflicts in complexity concerns. A simple example of this type of complexity conflict is illustrated in Figure 6. The code in this figure was constructed to optimize run-time efficiency. Using multiplication, the 16th power of a variable X is calculated. Although this code is computationally efficient and may be desirable in R3, it is undesirable in R4.

```
      Y = X
      DO 10 I = 1,4
   10 Y = Y * Y
```

Figure 6. Trading Run-Time Efficiency for Modifiability

A third justification of realm partitioning is based on the temporal nature of the life cycle. The detail level of information that is available at each life cycle stage varies. The information space enlarges at successive phases of the system life cycle. The realms, then, are partitioned to reflect this information spae differential.

Realms are defined in terms of properties. The properties describe the basis for the "complexity space." It is desirable to identify properties that are orthogonal. The orthogonality allows us to focus on the several dimensions of complexity, making complexity analysis a modular process. Further, orthogonality will ensure that several dimensions of complexity can be explained (measured) with a minimum of information. Let the properties of complexity be denoted by P.

where

P1 = Volume

P2 = Distribution

P3 = Location

Volume is a measure of the size of an entity. An entity may be a FORTRAN subroutine, a PSL process description, etc. Distribution is a measure of the interrelatedness of components within a module. Location is a measure of the intermodular interface complexity for a given module. The location measure deals with the relative functional/conceptual distance between modules. In general, location is a measure of "environmental" interaction. The orthogonality of these properties in software is supported by empirical evidence (Tanik, 1980). As discussed earlier, these properties are also useful in the evaluation of project complexity. These three properties provide a context independent basis for complexity assessment.

Each property is defined in terms of its attributes. Let the set of attributes be denoted by A. A given element in A is an attribute which belongs to property P in realm R.

There exist two basic types of attributes. There are attributes of the system itself (e.g., the number of discreet system processes), and there are the attributes of resources applied to realize the system (e.g., the qualifications of the project members). The first set of attributes is a result of system design, whereas the second set is a result of project management (resource allocation). This distinction is important from a system development standpoint. Complexity levels are controlled via the attributes that contribute to system complexity. The control mechanisms used will depend on the type of attribute in question. If the attribute is one of design, system design methodologies are used as control mechanisms. If, on the other hand, the attributes are resource related, then possible control mechanisms include resource allocation and scheduling.

In the following section, a complexity model is synthesized for realm R2. The effort metric of software science serves as the Volume property. An alternative to McCabe's Cyclomatic complexity, v(G), is proposed for the Distribution property. In addressing the Location property, the concept of information hiding (Parnas, 1972) serves as a representative example.

A COMPLEXITY MODEL FOR R2

P1: Volume

Halstead (1977) developed a linguistic-like model of software, called software science. Software science offers several program characteristic measurements which are based upon counts and estimations of counts of operations and operands. Software science metrics are derived using the following parameters:

$n1$ = the number of unique operators

$n2$ = the number of unique operands

$N1$ = the total number of occurrences of all operators

$N2$ = the total number of occurrences of all operands

The vocabulary, n, of an algorithm is defined to be $n1+n2$. The length of an algorithm, N, is defined to be $N1+N2$. Given that n tokens are used N times, the minimum number of bits needed to represent an algorithm can be expressed $(N1+N2)\log(n1+n2)$, or simply $(N)\log(n)$. The logarithms in the following discussion are base two. This measure, based on information theory, is referred to as the volume, V, of an algorithm.

The value of V depends upon whether an algorithm is implemented in a "high" or "low" level language. The most succinct (highest level) representation of an algorithm is a call to a function or procedure. A FORTRAN program which calculates the sine of a variable x can be written as simply SIN(X). The minimum, potential volume V* of an algorithm requires two operators (N1*=n1*=2). One operator is needed to name the function and the other to serve as an assignment or grouping symbol. The minimum number of unique operands n2* is equal to the number of input/output parameters. Since each unique operand need occur only once, we know that N2*=n2*. Potential operator and operand measures can be used to derive the minimal potential volume for a given algorithm.

$$V^* = (N1^* + N2^*) \log (n1^* + n2^*)$$

and given $N1^* = n1^* = 2$ and $N2^* = n2^*$

by substitution $V^* = (2 + n2^*) \log (2 + n2^*)$

Potential volume offers insight into the level of a program. The level of a program is defined as the ratio of potential volume to actual volume; given V* and V, program level, L, can be defined as V*/V. Two observations can be made based on this formulation. First, if a given algorithm is translated into a different implementation language, as the volume increases (decreases) the level decreases (increases) proportionally. Second, the product of L and V* remains constant for a given language. Halstead uses the measures of volume and program level to derive a measure of programming effort.

The implementation of an algorithm entails N selections from a vocabulary n. Halstead reasoned that if a "mental binary search" is used to make the N selections, then on the average (N)log(n) mental comparisons are required to generate a program. The number of elementary mental discriminations necessary to make each comparison is dependent on the program level L. Further, programming difficulty is inversely proportional to the level L. Therefore, the total number of mental discriminations E (for effort) necessary to write a given program can be given by

$$E = V / L$$

and as cited earlier

$$L = V^* / V$$

thus

$$E = V^2 / V^*$$

Of the many software science equations, the EFFORT equation cited above is most closely allied to a notion of complexity. The attributes used in the EFFORT model are language-defined and user-defined tokens. This choice of factors to be considered, as with all selection factors, directly impacts the interrelationships that can be determined. Several questionable assumptions are those related to unity and linearity.

Implicit in the counting strategy is the assumption that operators are of equal significance. Figure 7 illustrates two typical COBOL statements. As the counts for n1, n2, N1, and N2 are equal for both statements, software science metrics will derive equality in the complexity assessment of the two statements.

```
If A greater B and not C less D

Compute  A = (B + C) * D
```

Figure 7. Two Typical COBOL Statements

The counting strategy is a linear framework as well as unitary. It is evident that software science metrics do not explicitly reflect differences in program structure. Figure 8 illustrates two skeletal code segments. Figure 8A depicts a recursive (nested) decision construct, and Figure 8B depicts a linear case construct. Given that

```
IF                              IF
 IF                             ELSE IF
  IF                            ELSE IF
   IF                           ELSE IF
    ELSE                        ELSE IF
   ELSE                         ELSE
  ELSE
ELSE

    (B)                            (A)

A Recursive (nested)            A Linear (Case)
Decision Structure              Decision Structure
```

Figure 8. Two Skeletal Decision Structures

the counts for each code segment are identical, the software science measure of effort will again be equal for the two segments.

Despite the fact that software science makes many subjective explicit and implicit assumptions regarding language semantics, it has several qualities valuable to our research activities. First, the operator/operand counts are easily determined in the compilation process. Second, the model accommodates a view of abstraction level via the notion of language level. Third, by manipulating the counting strategy, software science can be made selective. Software science metrics are useful in assessing complexities arising from system volume (P1). As we have seen, however, they do not explicitly measure structural complexity.

P2: Distribution (Structure)

Process structures are frequently represented as graphs, with the nodes of the graphs representing control transfer points in a computer program, the discrete programs of a software system, the discrete systems in a distributed network, etc. The interrelationships between nodes are represented as connective arcs. In general, the arcs indicate some form of communication or relation between nodes. The interprocess communications include such relations as control transfers, exchanges of data sets, etc.

In assessing the complexity of a given graph structure, a major concern is the complexity of the collection of interrelations--the configuration of arcs. Figure 9 illustrates three nodes with their incident arcs. How can the complexity of each node be quantified? If it is presumed that complexity is a noncombinatorial phenomenon, then a simple count of the incident arcs would suffice. In the context of a graph this measure simplifies to the counting of arcs. Using an arc counting strategy, the complexities of the graphs in Figure 9 are 2, 4, and 6 respectively. In order to get a useful measure of system complexity, the relationships between nodes and arcs must also be addressed.

The cyclomatic number of a graph has properties that suggest its utility as a complexity metric. The cyclomatic

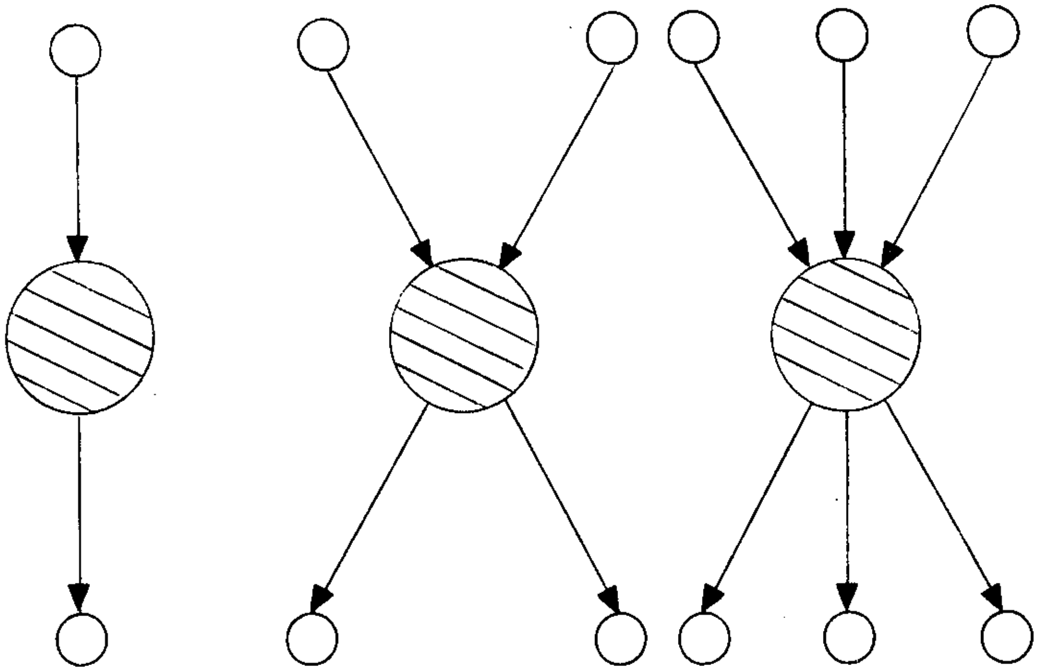


Figure 9. Nodal Complexity

number has been proposed as a measure of complexity in computer programs (McCabe, 1976). The cyclomatic numbers of the graphs in Figure 9 are all 1.

All of the above approaches assume that complexity is a linear function. The assumption of linear complexity increase is suspect. Intuitively, the entry of new complexity factors frequently has more than an additive affect on overall complexity. Intuitively, we would anticipate that linear metrics either neglect important dimensions of complexity or they do not effectively reflect complexity growth.

Given that structural complexity is heavily influenced by the combinatorial relationship between nodes, it is desirable to derive a function that reflects the combinatorics. One function that satisfies this criterion is

$$\text{nodal complexity} = \text{indegree} * \text{outdegree}$$

where indegree and outdegree are the counts of arc "heads" and arc "tails" that are incident to graph nodes. The measures of complexity for the three nodes in Figure 9 are 1, 4, and 9 respectively.

As dicussed earlier, process structures may be represented as graphs. In turn, precedence graphs are frequently represented as n by n boolean matrices. Given a matrix representation of a graph, B, the indegree of a given node is the sum along the respective column of B. The outdegree is the sum along the respective row. The ordered set of indegrees will be referred to as the CONVERGENCE of a graph. The ordered set of outdegrees will be termed the DIVERGENCE of a graph. The convergence of the graph forms an n-member (non-boolean) vector, C. The divergence of the graph forms an n-member vector, D. The vector product CD is an approximation of the connective complexity in a graph and, thus, of the underlying system being modeled.

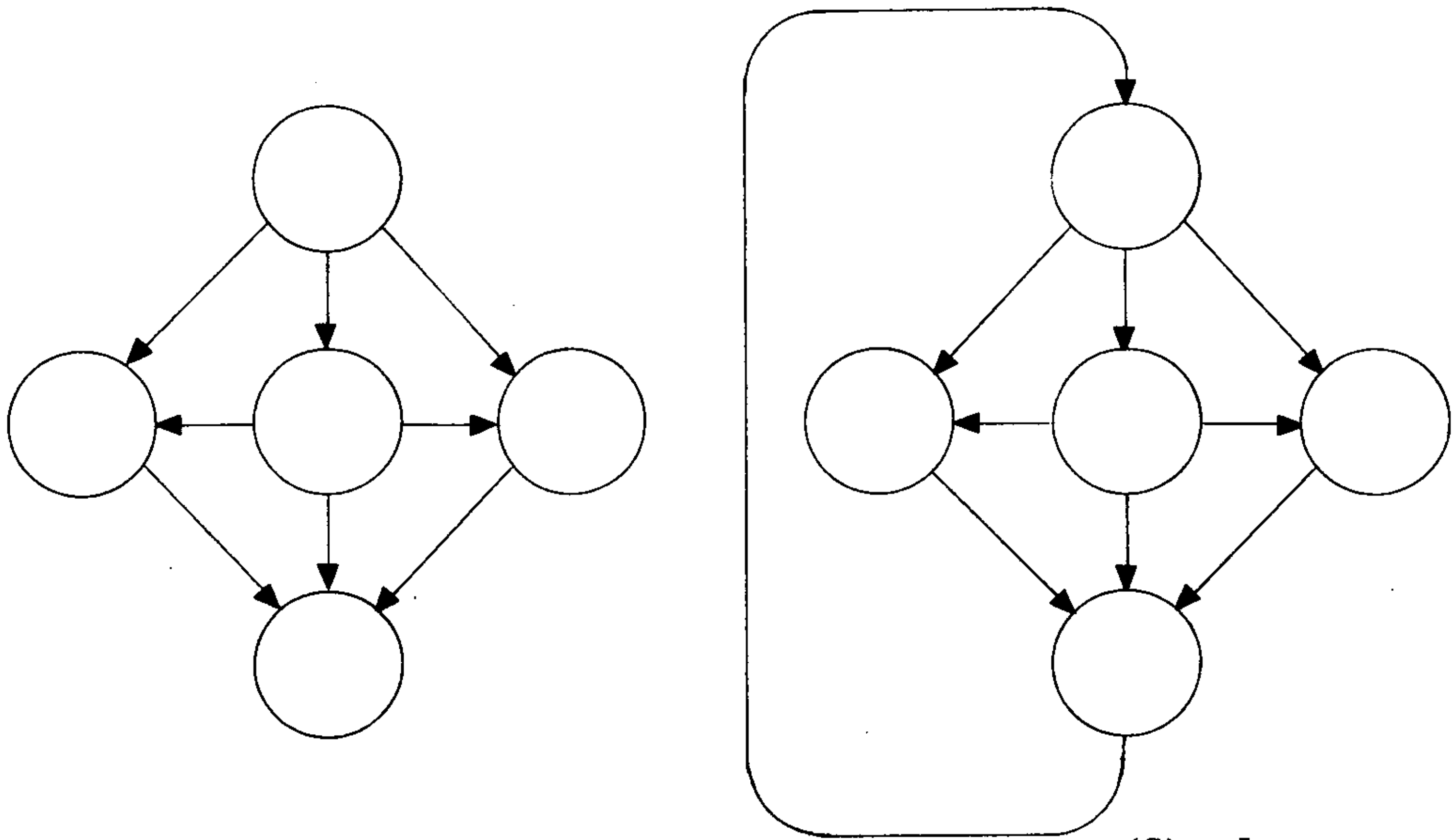
The structural complexity (distribution) measured by the product CD reflects only first order connectivity. The true complexity of a process is a function of the density of the transitive closure of the process as well. It may be desirable, then, to include higher order connectivity in the analysis. By taking successive powers of B, the product CD can be derived for all orders (dimensions) of connectivity. These orders can also be obtained by a traversal of the graph itself. Further, the reachability matrix might be used to reflect higher order connectivity. In the following discussion we will apply the three complexity measures discussed above to the analysis of structural complexity of software.

A frequently cited control flow complexity metric is the graph cyclomatic number. McCabe developed a complexity metric based upon the cyclomatic number of a program flow graph. The cyclomatic number of a graph represents the number of linearly independent circuits in the graph. These basic paths, when taken in combination, can be used to generate all possible paths through the graph. The metric defines the complexity of a program, v(G), to be the number of nodes (n) minus the number of arcs (e) plus 2.

The cyclomatic number is a measure of cycles (circuits). Therefore, the assumption that the graph is strongly connected underlies the measure v(G). The measurement of v(G) formally requires that the graph be strongly connected. To transform a given program flow graph into a strongly connected structure, cycles must be introduced. Cycles, however, are critical factors in themselves when assessing program complexity. The validity of "inventing" such cycles is questionable. Figure 10 depicts one such transformation. The invention of the single return arc has an unbounded impact on the number of paths through the graph, but has only a minor impact on the value v(G). The large increase in the number of paths would have significant influence on many aspects of program analysis--exhaustive testing, for example. An alternative way to force leaf nodes into the complexity analysis is to assume that the program graphs are well-formed. This transformation has no impact on the number of paths and has a trivial impact on v(G)--at most a delta v(G) of 1. In this discussion, it is useful to make the assumption that the graphs are well-formed rather than strongly connected. Note that well-formedness is a major premise of structured programming--one entrance, one exit--while the introduction of cycles in the form of "backward gotos" is discouraged.

Figures 11 through 14 are reproductions of several program flow graphs used by McCabe in illustrating the "behavior" of the cyclomatic complexity metric. The figures also include the values of v(G) and first order CD for each of the graphs. The relative ordinal ranking of the graph complexities are similar between v(G) and CD. The relative differences (ratios) in measured complexities between each of the graphs, on the other hand, vary for each of the v(G) and CD metrics.

Figures 15 and 16 are graphs of "extreme" situations. The cases illustrate the utility improvement of CD over v(G). Figures 15 and 16 present two program graph topologies and their respective skeletal code representations. Figure 15 depicts a program graph of a recursive decision structure; a completely balanced nested IF block. Figure 16 presents a program graph of a linear decision structure; a case IF block. As v(G) does not recognize the role of convergence and divergence and the propagation effect, it will judge both pro-



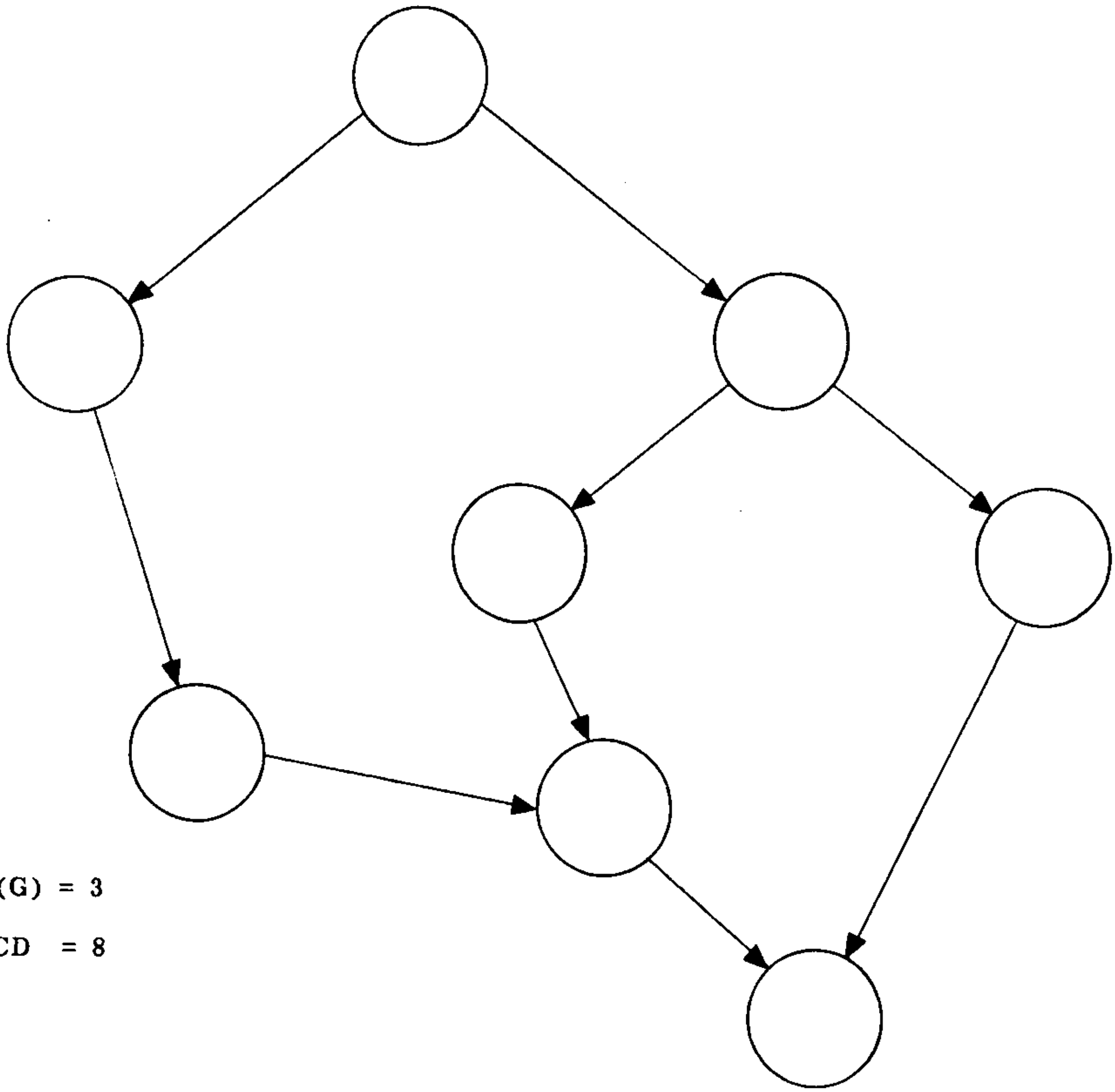
v(G) = 4
Number of paths = 5

(a)

v(G) = 5
Number of paths = ∞

(b)

Figure 10. "Invention" of a Return Arc



v(G) = 3

CD = 8

Figure 11

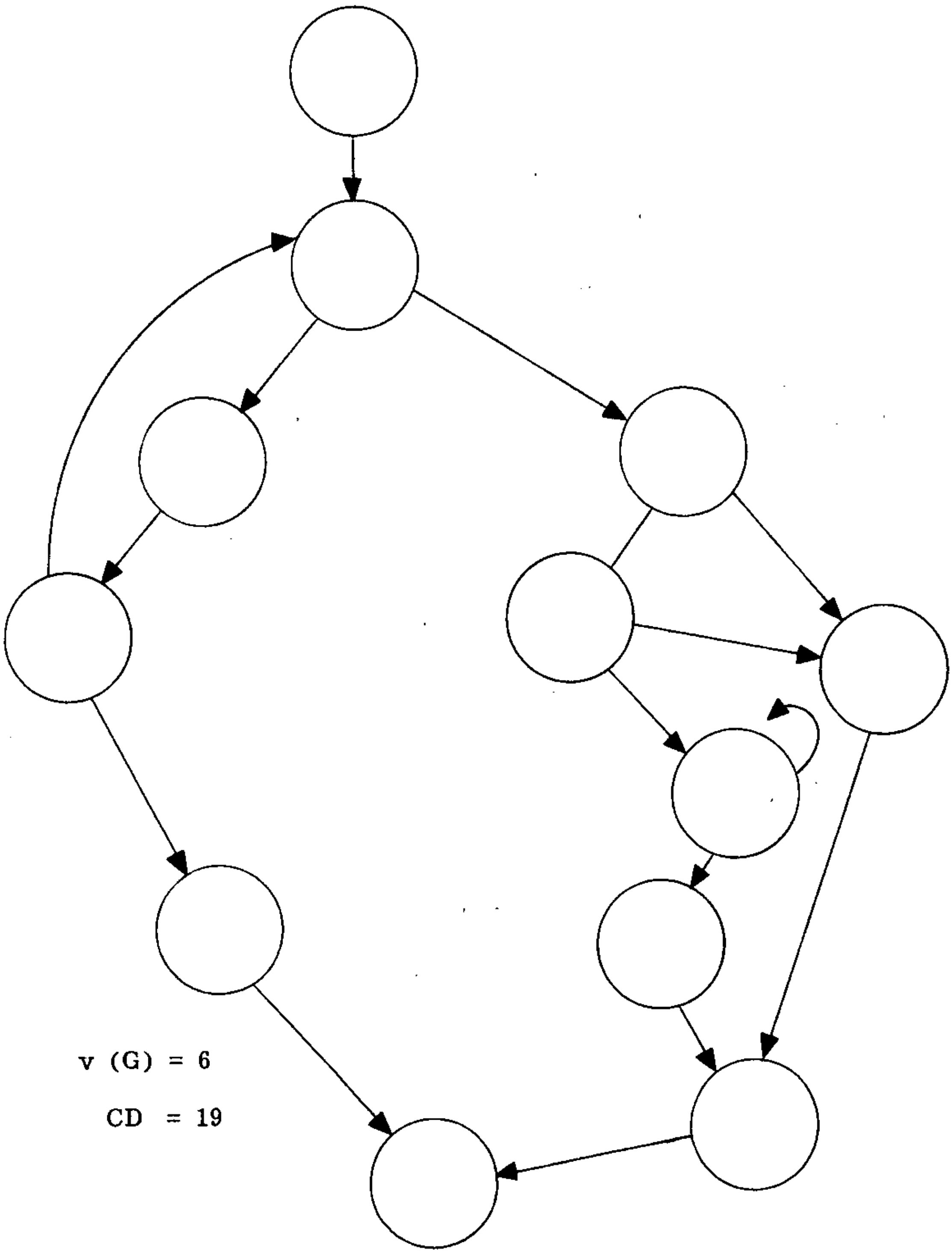


Figure 12.

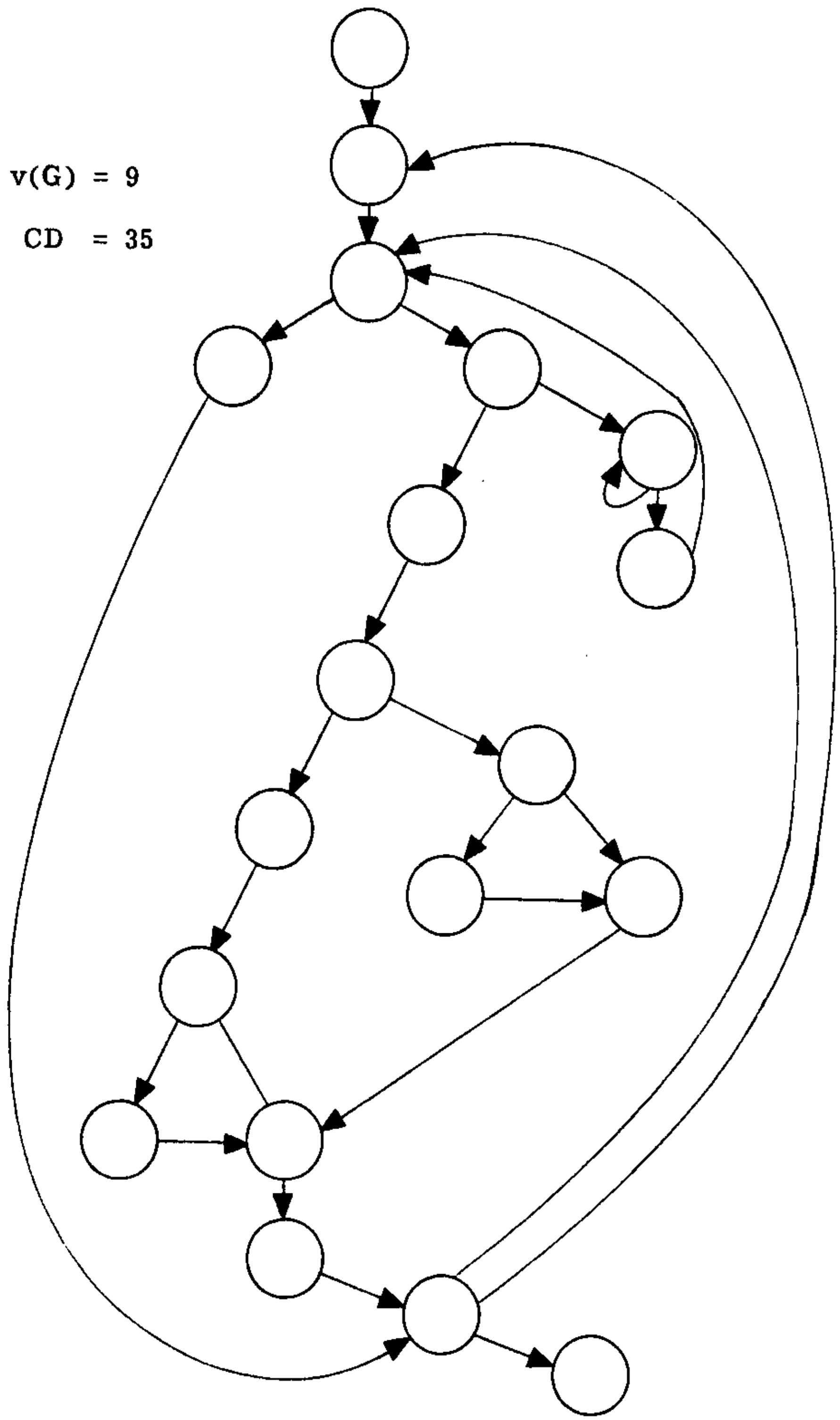
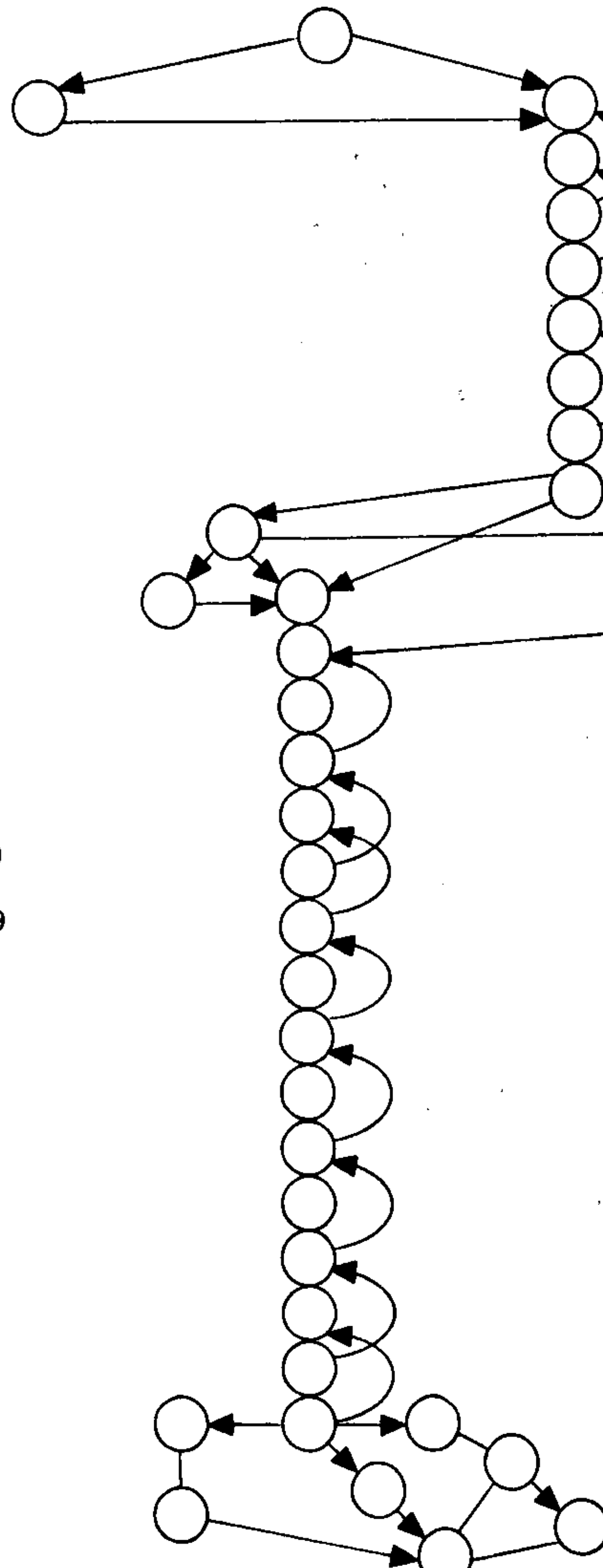


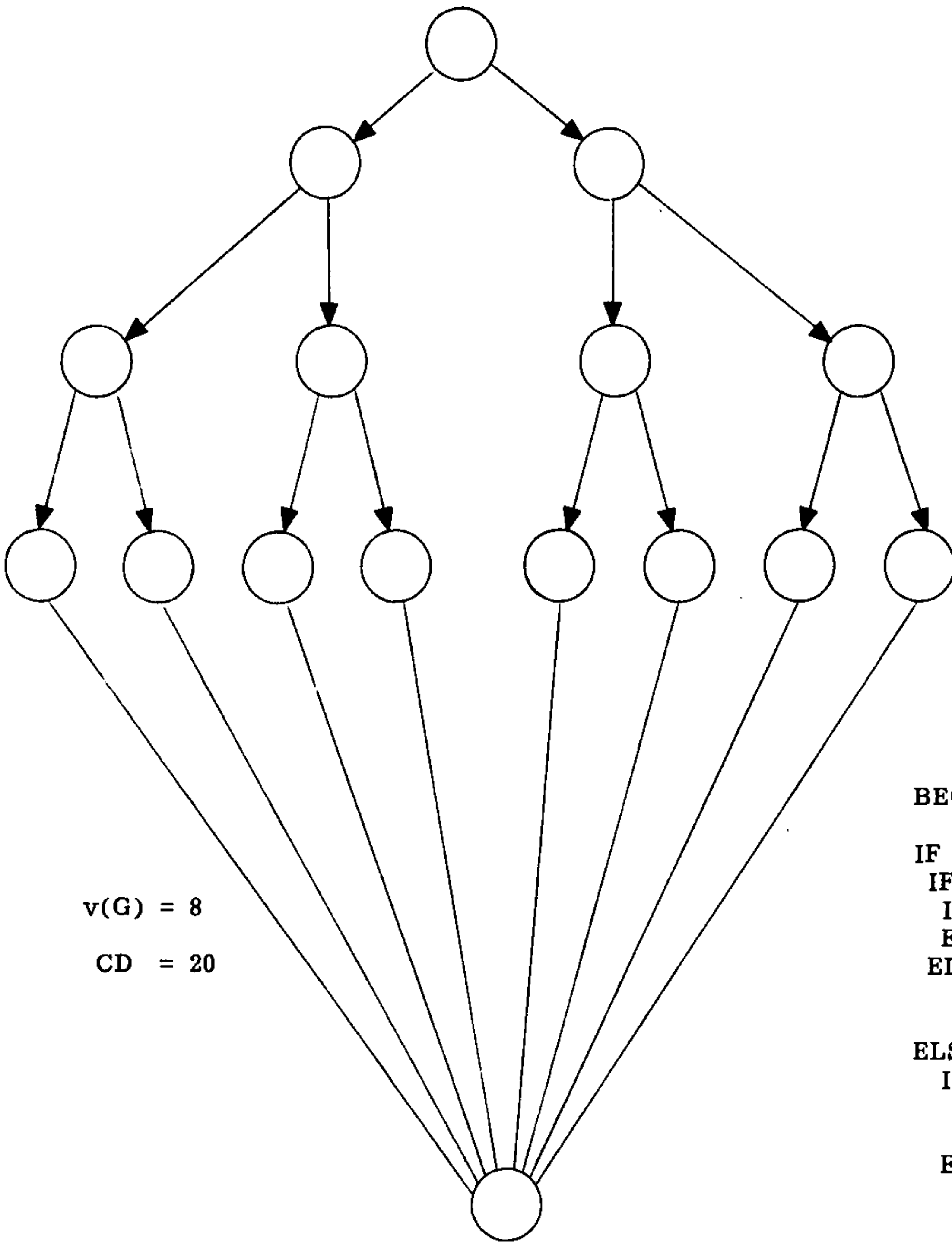
Figure 13.



v(G) = 19

CD = 79

Figure 14.

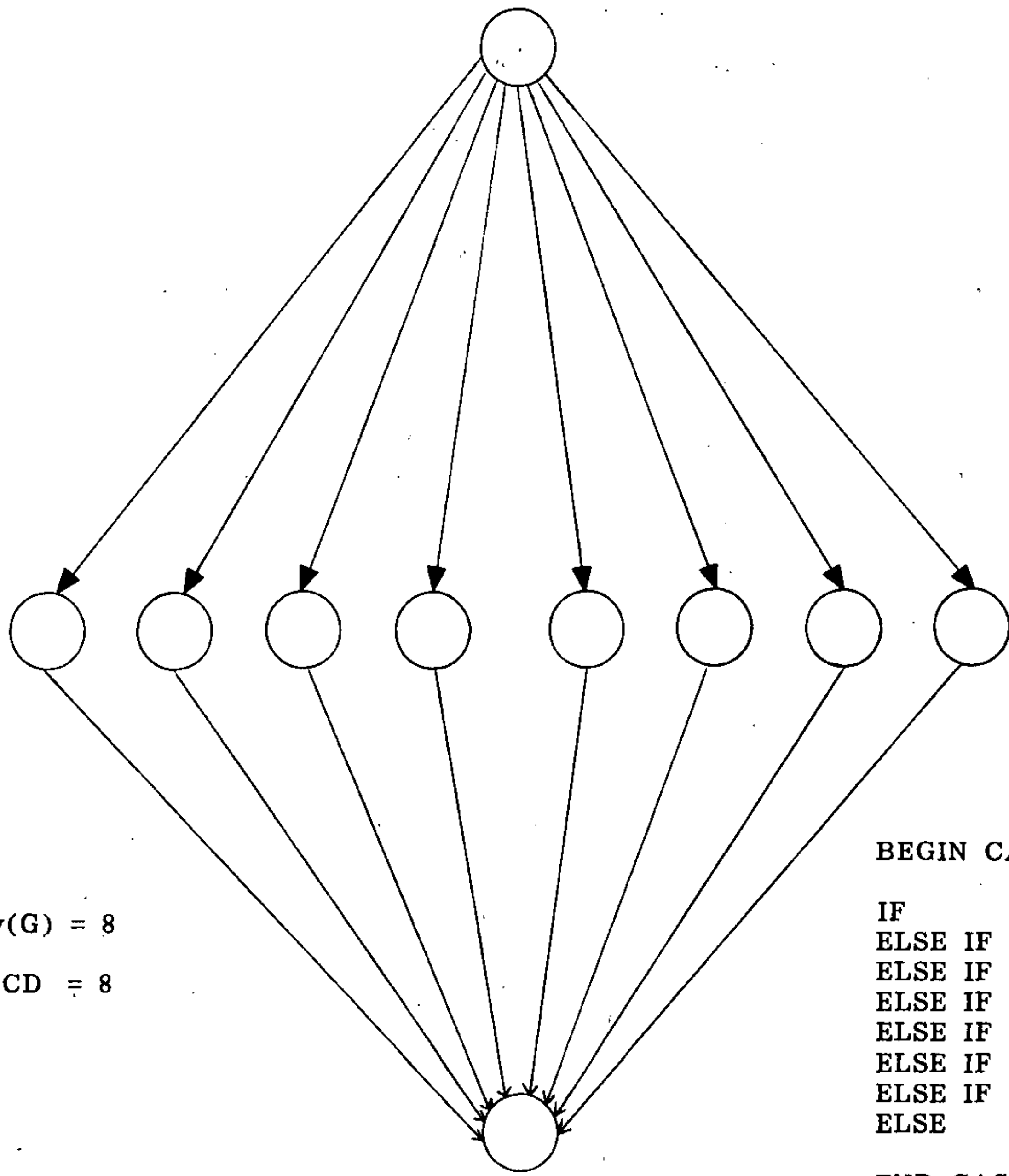



```
BEGIN NEST

IF
 IF
  IF
  ELSE
 ELSE
    IF
    ELSE
ELSE
  IF
    IF
    ELSE
  ELSE
    IF
    ELSE

END NEST
```

Figure 15. A Recursive (Nested) Program Flow Graph and Code Representation



v(G) = 8

CD = 8

```
BEGIN CASE

IF
ELSE IF
ELSE IF
ELSE IF
ELSE IF
ELSE IF
ELSE IF
ELSE

END CASE
```

Figure 16. A Linear (Case) Program Flow Graph and Code Representation

grams to be of equal complexity: v(G) in Figure 15 = 8, v(G) in Figure 16 = 8. The measure CD, on the other hand, does recognize the distinction between these two programs: CD in Figure 15 = 20, CD in Figure 16 = 8.

As the number of arcs is dependent on the number of nodes in the above topologies, the functions v(G) and CD can be reduced to functions of the number of nodes. A plot of v(G) for each topology is presented in Figure 17, and a plot of CD is presented in Figure 18. We should note that n less than or equal to 4 is a degenerative case where the two topologies are identical.

Based upon the plot of measured complexity as a function of the number of nodes in Figure 17, the measure v(G) asserts that:

1. Given an equal number of nodes (n >4), the complexity of a case structure is greater than that of a nested structure.

2. As the number of nodes increases, the complexity of a case structure increases at twice the rate of a nested structure.

Based upon the plot in Figure 18, the measure CD asserts that:

1. Given an equal number of nodes (n >4) the complexity of a case structure is less than that of a nested structure.

2. As the number of nodes increases, the complexity of a nested decision structure increases at one and one half times the rate of a linear case structure.

When analysis of CD is extended to higher orders of connectivity, the measured complexity of the recursive decision structure of Figure 15 will increase. Higher order analysis does not increase the measured complexity of the linear case structure of Figure 16.

The distribution complexity (P2) of a module is a measure of intra-modular structure. It should measure the density of connection between components in a module. A simple count of arcs does not reflect relative degrees of connective density. The measure v(G) treats density as a global, linear property of a graph. Further, CD treats the density of connection of a given node as a nonlinear property. Using first-order CD, the overall complexity is reflected as the sum of individual nodal complexities. Full-order CD, on the other hand, treats both nodal and overall graph complexities as nonlinear progressions. Given Halstead's effort metric as a measure of the complexity of "describing" the components of a module and CD as a measure of the complexity of intra-modular connectivity, we must now address the complexity of a module's interface with other modules.

P3: Location (Interaction)

Parnas recognized that changes in a completed system are simplified when the connections between modules are designed to contain as little information as possible (Parnas, 1972). Design decisions should anticipate change. The basic thesis of Parnas' work in this area is that the designers who specify the system structure should control the distribution of design information, hiding details that are likely to change. One objective of this approach is the isolation of change impact.

The notion of "information hiding" addresses the design process as much as it addresses the product of design. In hiding design information, we are also making the functional responsibilities clear which affects the understandability and changability of the system. Measures must reflect the amount of knowledge that the

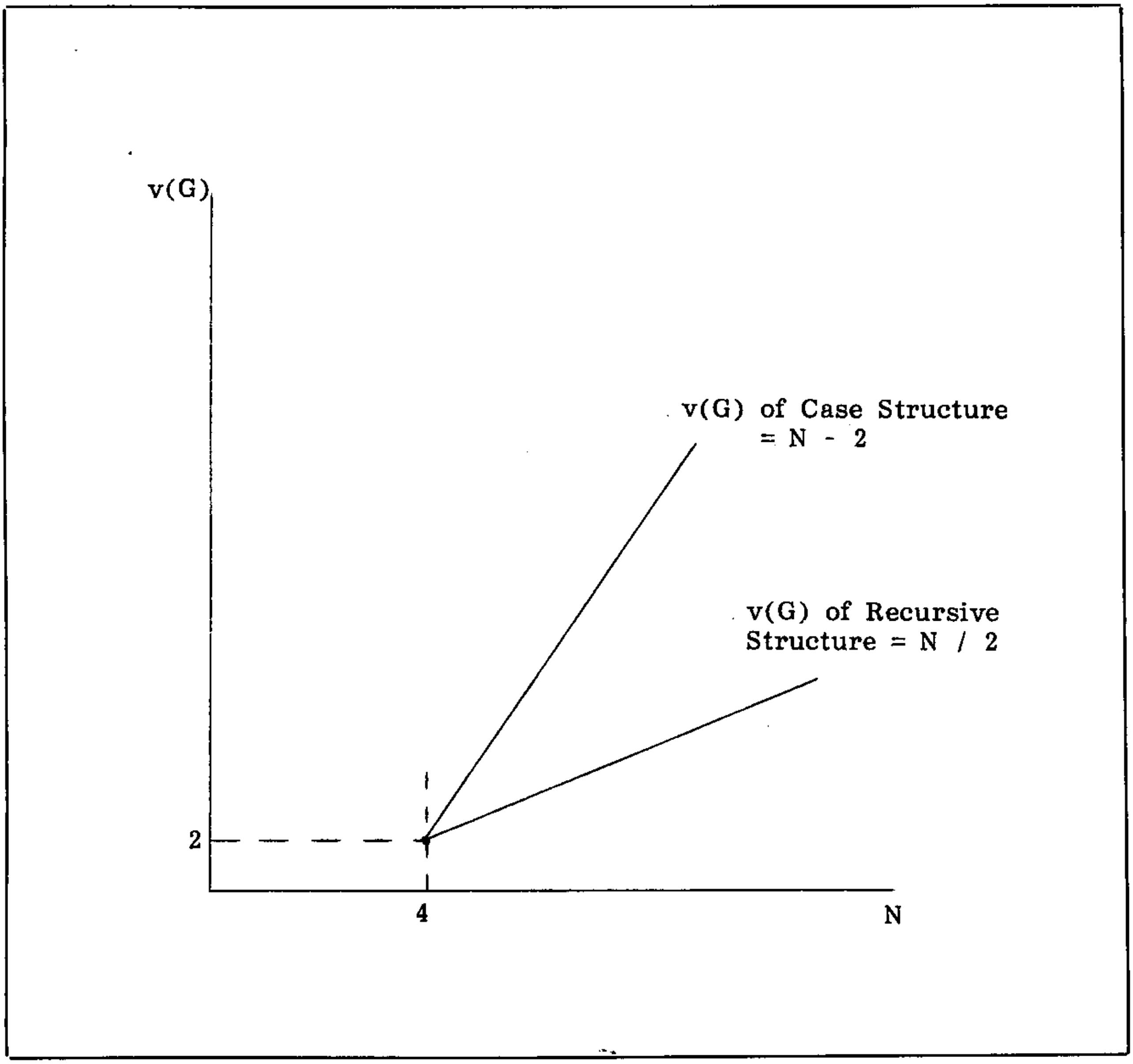


Figure 17. A Plot of v(G)

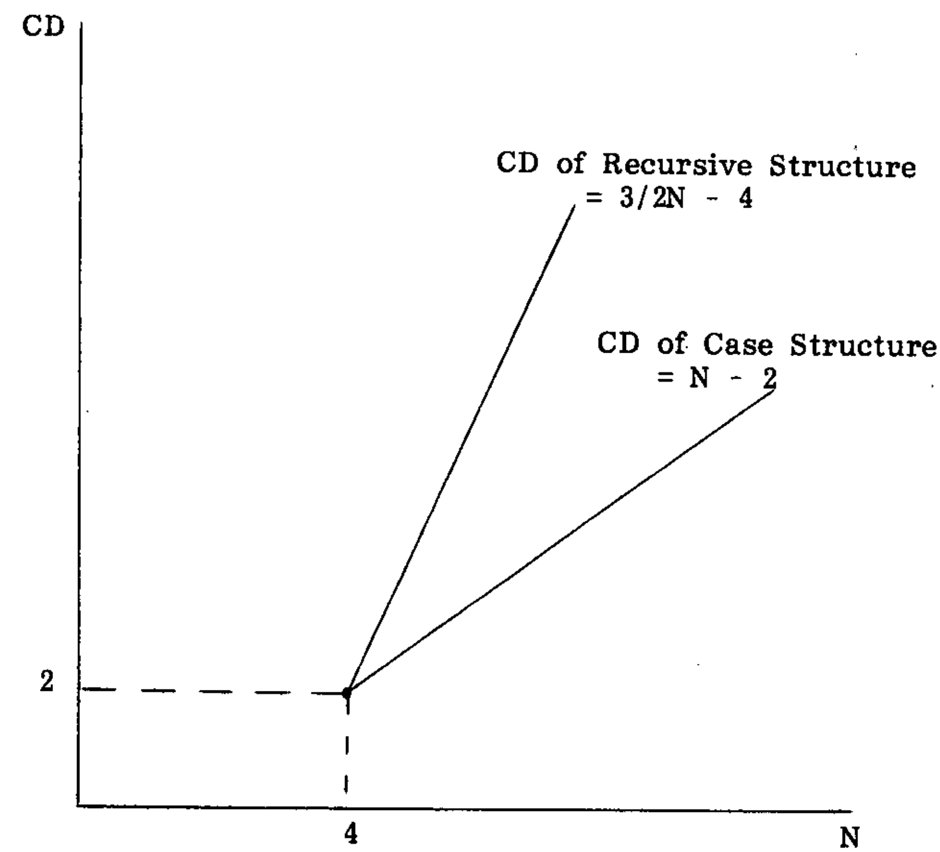


Figure 18. A Plot of CD

system modules have of each other concerning the nature of their activities. We can see that inefficiencies are introduced under this criteria. By minimizing the amount of information shared between modules one expects that the difficulty of system construction and modification will be lessened.

In terms of the location property (information hiding), we are concerned with the type and amount of inter-module communication. A straightforward measure of the location property in realm R2 is

$$\text{Location} = \sum_{I} (Wi)(Ci)$$

Where Ci is a count of the number of communicated items of type i, and Wi is the weight attached to communicating items of type i. Examples of Ci include the number of input parametes and the number of output parameters.

Synthesis of an R2 Complexity Model

The above complexity metrics may be applied in synthesis of a complexity function for realm R2. The EFFORT metric of software science can serve as the Volume property, the measure CD as a Distribution property metric, and information hiding as the Location property. A weighted combination can be used to measure complexity in R2.

$$R2 = \sum_{P} (Wp)(f(Ap))$$

Where Ap is the set of attributes belonging to property P, f(Ap) is the function defining the relationship between each Ap in P, and Wp is the relative weight (coefficient) of property P in R2. An example taxonomic realm/property/attribute breakdown is depicted in Figure 19.

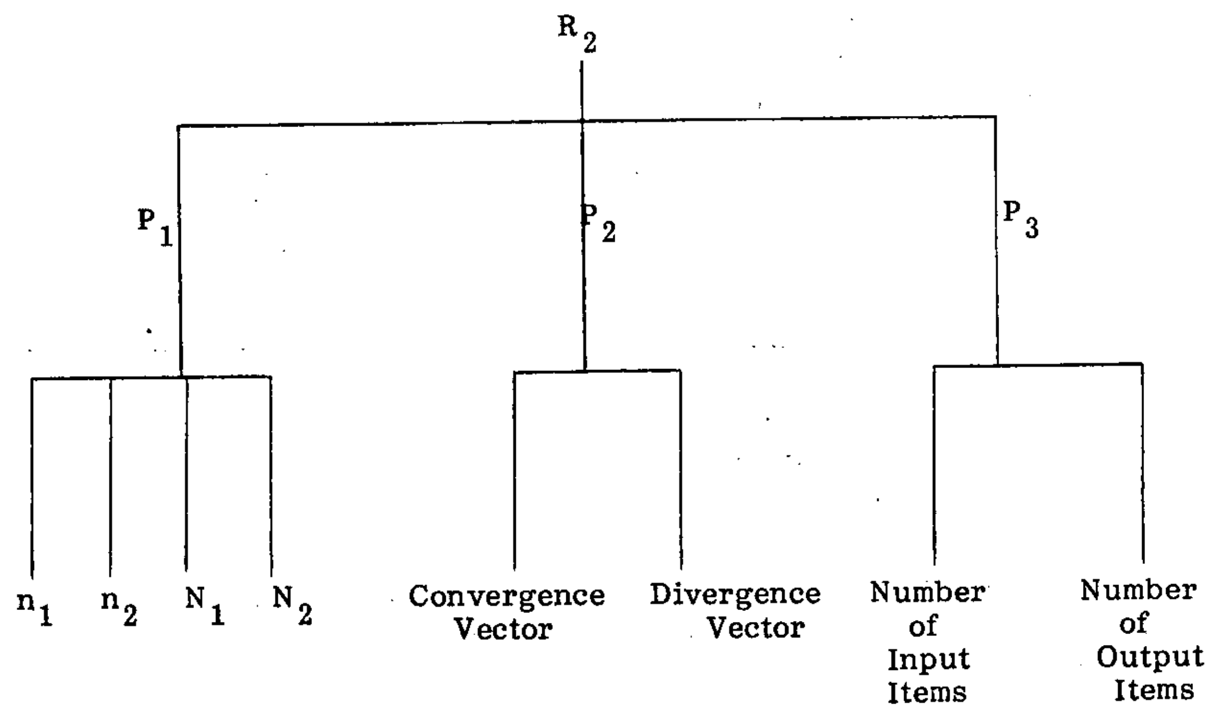


Figure 19. Attribute Taxonomy for Realm R2

Complexity Measurement in Other Realms

Existing software complexity models are concerned in large part with the stages of physical design and construction. These stages, however, are relatively inexpensive in terms of resource consumption. Further, the complexities manifested at these stages are often a direct result of complexities introduced at earlier stages of the life cycle, and complexities introduced in early stages manifest themselves heavily in the later phases of operation and maintenance. Problems of logical design complexity and maintenance complexity are more acute and costly. If a complexity model is to serve as a robust tool for software engineering and software project management it must encompass the entire life cycle. The benefits of a broader view of complexity will be far reaching.

The formulation of complexity models in each realm is constrained by the amount and nature of information available in the realm. The attributes available in each realm are candidates for inclusion in the complexity model. Further, attributes that are available in more than one realm offer the potential for complexity projection.

Complexity Projection

Complexity measures within a given realm are useful in the evaluation of productivity of the realm activities. The measures serve as feedback mechanisms and guidelines for programmers, analysts, and designers. To be of use to project managers, however, a complexity model must go further--it must support project scheduling and resource allocation. Given, for example, information available at the end of the logical design phase, R1, it would be useful to project the complexity in R2. As mentioned earlier, several attributes may span realms. Although the spanning phenomenon introduces conflicting objectives, it does make complexity projection feasible.

Chrysler (1978) has conducted research related to complexity projection. Chrysler used step-wise multiple regression to correlate program and programmer characteristics with the time taken to complete a programming task. Using a sample of 31 COBOL programs, five independent variables were found that resulted in a multiple correlation coefficient of .836.

1. Programmer experience at the facility.

2. Number of input files.

3. Number of control breaks and totals.

4. Number of input edits.

5. Number of input fields.

The values for the five independent variables are available at the conclusion of the logical design phase, and are available for projecting complexity from R1 to R2. Further, the attributes can be broken into the two classes--design attributes and resource attributes. Whereas the variables two through five are design attributes and must be controlled during logical design, the first attribute is a human resource attribute and can be controlled at the end of R1. A similar approach can be used to develop complexity projection functions between all realms. For example, a measure of environmental volatility made during logical design (R1) can be used to project maintenance complexity (R4).

Once complexity measures have been formulated for each realm, a similar approach can be used to derive complexity projection functions. A first approximation of the projection functions can be derived by identifying the attributes available in realm Rn which were included as independent variables in the complexity model for realm Rn+1. Then, using a generalized model management approach

(Elam and Henderson, 1980; Konsynski, 1981), the complexity measurement and projection models can be refined on a continuing basis.

The refinement process takes two forms--projection refinement and model refinement. Projection refinement is a function of the temporality of the system life cycle. In R1 a relatively limited amount of eventual implementation information is available. The validity of a projection, though, is a function of the amount of available information. The complexity in R4, for example, is better projected from R2 than from R1. As the design process continues, then, the complexity projections can be refined.

If a complexity model is to reflect the evolutionary nature of system development, the model must "learn." Model refinement via feedback will ensure that the model reflects reality. When, for example, the actual maintenance complexity for a given system is attained, this information can be used to update the forecasting/projection functions, R1 to R4 and R2 to R4.

Conclusions

A robust Information System complexity model must accommodate a holistic view of the system development process. We benefit little from application of control mechanisms late in the life cycle when the factors contributing to system complexity were introduced in a much earlier phase. By iterating toward a thorough complexity model, relevant factors of complexity will be unveiled. Complexity control mechanisms can then be initiated to control relevant factors during the proper life cycle phase.

The authors have examined the characteristics of current alternative complexity measures with the objective of determining useful measures during different stages in the system design process. Both the linguistic and graph theoretic models offer useful information at different stages. Further, both have limitations that make them insufficient as stand alone measures. An object/relation paradigm offers a basis for comparison between alternative techniques.

Four realms in the design process were discussed in which differing complexity concerns exist. These realms deal with initial requirements specification and analysis, the design and implementation process, the system in operation, and the maintenance and modification activities. The differing "environmental concerns" call for different complexity concerns in each realm. A general framework was proposed to realize an overall system development complexity model which will support complexity assessment and projection.

BIBLIOGRAPHY

Chen, E.T. "Program Complexity and Programmer Productivity," IEEE T on Software Engineering, Software Engineering, Volume 4, Number 3, May 1978, pp. 187-194.

Chrysler, E. "Some Basic Determinants of Computer Programming Productivity," CACM, Volume 21 (1978), pp. 471-483.

Curtis, B., et al. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE T on Software Engineering, Software Engineering, Volume 5, Number 2, March 1979, pp. 96-104.

Elam, J., Henderson, J., and Miller, L. "Model Management Systems: An Approach to Decision Support in Complex Organizations," Proceedings of the Conference on Information Systems, 1980, pp. 98-110.

Fitzsimmons, A. and Love, T. "A Review and Evaluation of Software Science," ACM Computing Surveys, Volume 10, Number 1, March 1978, pp. 2-17.

Halstead, M.H. Elements of Software Science, North-Holland Publishing Co., 1977.

Hansen, W.J. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," SIGPLAN Notices, Volume 13, Number 3, March 1978, pp. 29-33.

Kearney, J., Sedimeyer, R., Thompson, W., Adler, M., and Gray, M. "An Analysis of Software Complexity Measurement," Technical Report 81-26, Computer Science Department, University of Minnesota, Minneapolis, Minnesota, July 1981.

Konsynski, B.R. "Model Management in Decision Support Systems," Presented at the NATO Advanced Study Institute on Data Base and Decision Support Systems, Estoril, Portugal, June 1981.

McCabe, T.J. "A Complexity Measure," IEEE T on Software Engineering, Software Engineering, Volume 2, Number 4, December 1976, pp. 308-320.

Oulsnam, G.G. "Cyclomatic Numbers Do Not Measure Complexity of Unstructured Programs," Info Processing Letters, Volume 9, Number 5, December 16, 1979, pp. 207-211.

Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules," CACM, Volume 15, Number 12, December 1972, pp. 1053-1058.

Schneider, G.M., Sedlmeyer, R.L., and Kearny, J. "On the Complexity of Measuring Software Complexity," AFIPS Conference Proceedings, May 1981, pp. 317-322.

Tanik, M.M. "A Comparison of Program Complexity Prediction Models," ACM SIGSOFT, Volume 5, Number 4, October 1980, pp. 10-16.

Tausworthe, R.C. Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

Woodward, M.R., et al. "A Measure of Control Flow Complexity in Program Text," IEEE T on Software Engineering, Software Engineering, Volume 5, Number 1, January 1979, pp. 45-48.

Zolnowski, J.C. and Simmons, D.B. "Measuring Program Complexity," Digest of Papers of Fall COMPCON77, pp. 336-340.