Artificial Intelligence and Language Processing

Jacques Cohen Editor

Design Complexity Measurement and Testing

System designers can quantify the complexity of a software design by using a trio of finely tuned design metrics.

Thomas J. McCabe and Charles W. Butler

During the past decade, software development concepts have undergone a dynamic revolution. Software development methodologies have evolved to meet changing life cycle patterns which have had as their objective, emphasis on analysis and design. In addition, computer-assisted software engineering (CASE) has emerged to meet the unprecedented growth in analysis and design activities and is affecting how future systems will be developed and tested. These methodologies reinforce the need to accurately define systems specification prior to implementation.

The concern over present and future software quality has grown as the volume and complexity of applications increase. Software applications are critical to business operation and lead to severe maintenance problems when they fail. This fact has prompted many firms to develop software engineering programs which attempt to define and implement techniques for software validation, verification, and testing throughout the development life cycle.

THE PROBLEM

Since there is increased emphasis on earlier stages of the life cycle, there is a need to build design complexity measures for the development of software systems and to formulate a procedure to utilize them in testing. Users, software managers, and project leaders try to justify projected cost and time for development as the development cycle proceeds. Projections made during design have to improve upon those made during specification and should generate a more accurate projection of future testing requirements. Since complexity is a significant determinant of a system's success or failure, the risk is high for development decisions based strictly on qualitative evaluations. A number of the same concerns which are addressed for program complexity apply equally as well at the design level:

1. Designs with which we deal are overwhelmingly complex.
2. Designs cannot be fully comprehended by developer or reviewer.
3. Designs are not rigorously verified through a testing process. In this instance, testing strategy is not derived directly from the design specification (nor is the level of testing directly proportional to the complexity of the design).

© 1989 ACM 0001-0782/89/1200-1415 $1.50

Quality software should have the characteristics of being understandable and measurable [1]. Measurements are closely linked to software metrics theory which has received growing attention over the past decade. Much of the research on software metrics has been involved with program complexity languages such as PL/I, Pascal, Fortran, and C. While many studies dealt with the subject of measuring program complexity, few studies have concentrated on measuring complexity of development specifications.

THE OBJECTIVE

The software engineer should consider the design complexity and should understand its implication before proceeding into the construction stage of software development. Furthermore, few formal theories exist from which a design complexity measure can be generated. One such approach, cyclomatic complexity, is a mathematical technique for program modularization and unit testing. Before cyclomatic complexity can be used in measuring design complexity, certain aspects must be examined.

The objective of this article includes two elements. The first is to extend the mathematical basis of cyclomatic complexity into architectural design of a system. In this context, architectural design is defined as the framework or structure of a system such as a hierarchy chart with its associated functions and control interrelationships. The second element is to develop a testing methodology integrating the intuitive notions of design complexity and integration testing requirements.

To accomplish the stated objectives, cyclomatic complexity is applied to architectural hierarchical design. The cyclomatic complexity approach is to measure and

control the number of paths through a program. It limits the number of basis paths in a source module. The analogous design entity is the subtree. Therefore, a major component of the methodology is to measure the number of subtrees through an architectural design.

In addition, fundamental tenets of structured testing for programs are utilized to build a testing methodology for integration testing of an architectural design. Two tenets, cyclomatic complexity and a basis set of test paths, are utilized as components of structured integration testing. From this methodology, refinements are added and important integration strategies, such as top down and critical piece implementation are considered.

CYCLOMATIC COMPLEXITY

Since 1976, a number of software metrics have been developed. From the wide range of software metrics, four basic theories have been the source of the majority of the research conducted on software metrics. The first three theories were defined by Halstead, Albrecht, and DeMarco [2, 8–11, 13]. The last of these theories was defined by McCabe, as cyclomatic complexity, a measure of the number of paths through a program [16]. The number of paths can be infinite if the program has a backward branch. Therefore, the cyclomatic measure is built on the number of basis paths through the program.

Cyclomatic complexity, $v(G)$, is derived from a flowgraph and is mathematically computed using graph theory. More simply stated, it is found by determining the number of decision statements in a program and is calculated as:

$$v(G) = \text{number of decision statements} + 1$$

By counting the decision statements, called predicates, the complexity of a program can be calculated. However, many decision statements contain compound conditions. An example is a compound IF statement:

IF A = B AND C = D THEN

If the predicates are counted in this example, $v(G)$ is equal to 2 (1 IF statement + 1). If compound conditions are counted, the statement could be interpreted as:

IF A = B and IF C = D THEN

Therefore, $v(G)$ would be 3. Cyclomatic complexity recognizes that compound predicates increase program complexity and integrates individual conditions in order to calculate $v(G)$. An upper limit of 10 for program complexity is proposed because greater complexity would be less manageable and testable [16].

Software Metric Research

Although cyclomatic complexity yields quantification, there is no research which has established absolute thresholds for quality software. A number of studies, however, have investigated its significance.

In one study, Myers calculated $v(G)$ for the programs contained in the classic text by Kernigan and Plauger. For every case in which an improved program was suggested, this improvement resulted in a lower value for cyclomatic complexity [15, 17]. In a second study, Walsh collected data on the number of software errors detected during the development phase of the AEGIS Naval Weapon System. The system contained a total of 276 modules, approximately half of which had a $v(G)$ of 10 or less and half a $v(G)$ more than 10. The average error rate for the modules in the first group was 4.6 per 100 source statements while the corresponding error rate for the more complex modules was 5.6 [21]. In a series of controlled experiments conducted at General Electric, $v(G)$ was found to predict the performance of programmers on comprehension, modification, and debugging tasks [19]. Finally, Henry, Kafura, and Harris reported empirical error data collected on the UNIX operating system. The correlation between cyclomatic complexity and the number of errors was above 0.90 [14].

More recently, research has provided evidence of the management potential of software metrics. A study by Butler, Richardson, and Hodil utilized software metrics as an important ingredient for a prototype knowledge-based system [3, 4]. The knowledge representation scheme was rule-based, and the rules were developed to evaluate and prescribe maintenance action for commercial software. Since cyclomatic metrics correlated with key performance measures such as job failure and success, they were integrated into the rules for the KBS.

Taking a different approach, Carver measured the effects of program modification during testing on numerous complexity metrics [5]. One of the conclusions reached by Carver was that if viable estimates of complexity increases are computed early in the software development process, the software designer can determine when a module should be subdivided. Subdivision, itself, creates a practical dilemma within the testing phase.

Petschenik outlined the need for practical priorities in system testing [18]. While identifying three priority rules that provide criteria for selecting test cases, he found that developers focused on individual system components rather than how those components worked together. However, as Carver alluded, it would represent an important management tool for the software designer, if the metric could be used to quantify individual components and their integration needs.

DESIGN COMPLEXITY METRICS

Background

The current principles of the cyclomatic complexity metric are to (1) apply it to source code; (2) avoid excessive complexities that cause reliability problems; and (3) use the quantification to drive a testing process that will detect errors. Given a program, a flowgraph can be associated with it. In the graph, each node corresponds to a block of code where the flow is sequential and arcs correspond to branches in the program. The cyclomatic

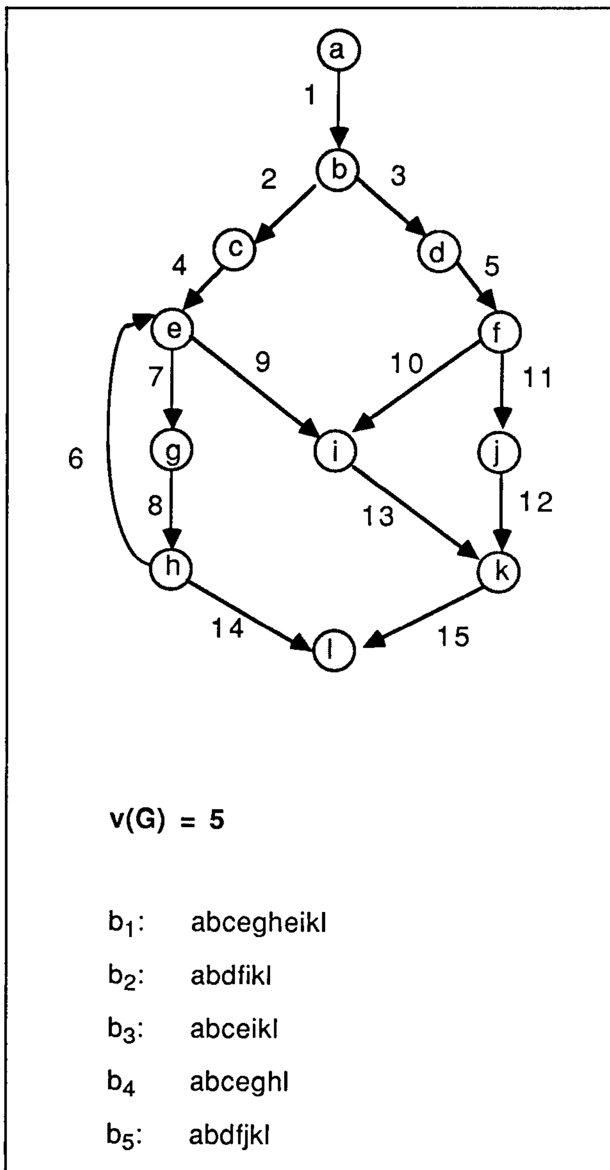

FIGURE 1. Sample Flowgraph with Cyclomatic Complexity

complexity of a graph with n vertices, e edges, and p connected components is [20]

$$v(\mathrm{E}) = e - n + p$$

Based on established mathematical properties, the cyclomatic complexity is equal to the maximum number of linearly independent paths through the program. Thus, as illustrated in Figure 1, the cyclomatic complexity of the flowgraph is

$$\begin{aligned} v(\mathrm{E}) &= 15 - 11 + 1 \\ &= 5 \\ &\quad \text{for } e = 15, \quad n = 11, \quad \text{and } p = 1 \end{aligned}$$

A basis set of five paths is also defined representing the maximum number of linearly independent paths through the program.

Path Subtree Analogue

In structured design, the primary design instrument is the structure chart or hierarchy tree. As stated earlier, the previous cyclomatic work was path-based. The analogous design entities are a design tree and a design subtree. A design tree is the ordering established by the hierarchical relationship among modules of a system. A design subtree is a realizable subset of this hierarchy that can be executed through a design's input data. Invoking a design tree means that it is entered at the top, executes lower-level modules and eventually exits through the top. This process results in a subtree within the original design tree structure. Just as a program can have an inordinately high number of paths, it is possible that a design tree can have an overwhelming finite number of subtrees.

An example will help illustrate the design tree and design subtree concepts. Figure 2 shows a design with complexity 1, the only subtree is the design structure itself. The design tree in Figure 3 has the same number of modules and interrelationships as the design tree in Figure 2. It is, however, more dynamic as there are six decisions within it, noted by the decision (dot) and repetition (loop) conditions. One subtree is 1, 2, 5, 10, and 11 indicating the functions 1, 2, 5, 10, and 11 are invoked. However, function 2 does not invoke function 6 and function 1 does not invoke functions 3 and 4 and their associated subordinate functions.

Assume in the design structure illustrated in Figure 3 that the loop in Module 1 iterates between 1 and 3 times. The following expression quantifies the number of possible subtrees in the design.[1]

$$\begin{aligned} \text{Subtrees} &= 2 \times \sum_{i=1}^{3} ([1 + 1(1 + 2^2)] \times [(1 + 1(2^2)) \times 2^2])^i \\ &= 3{,}485{,}040 \end{aligned}$$

The reader can see that compared to everyday designs the illustrated design is relatively straightforward, but it yields a total of more than three million distinct subtrees. For this reason, the total number of subtrees cannot be used as a practical design structure quantification.

In building the architectural design analogue, design reliability is enhanced when (1) design complexity is quantified; (2) design complexity is limited; and (3) the integration testing process is driven with design metrics. Given the fact that the number of distinct subtrees can be arbitrarily high even within a relatively simple design, counting the subtrees for our design complexity is nonsensical. We will, therefore, define integration complexity to be a basis set of subtrees that, when

[1] The following may help the reader track the subtree quantification with the design structure. There are two possible subtrees through module 2 (M2), (M1-M2-M5-M10-M11) and (M1-M2-M5-M10-M11-M6). Since the loop at M1 iterates between 1 and 3 times, we have $2 \times \sum_{i=1}^{3}$ (# of subtrees from M3 and M4). The expression 2^2 represents the number of subtrees from M12 (M12, M12-M13, M12-M13-M14, M12-M14); the expression $(1 + 2^2)$ represents the number of subtrees from M8. The desired result for the # of subtrees from M3 and M4 is the number of subtrees from M3 $(1 + 1(1 + 2^2))$ multiplied by the number of subtrees from M4 $[(1 + 1(2^2)) \times 2^2]$.

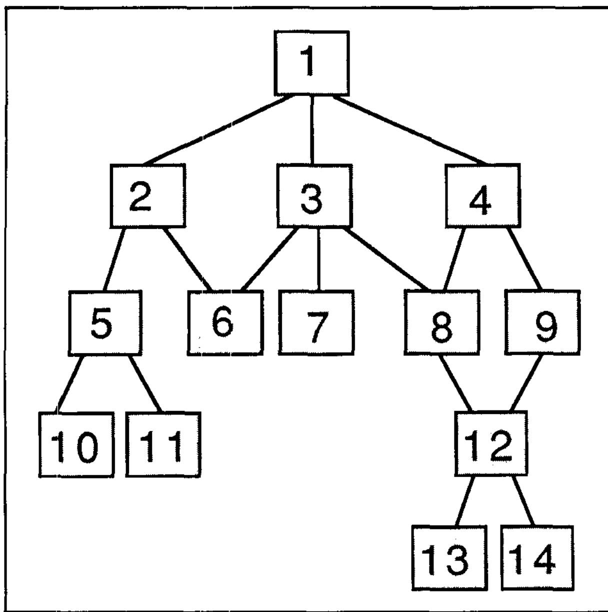

FIGURE 2. Design Trees with Complexity = 1

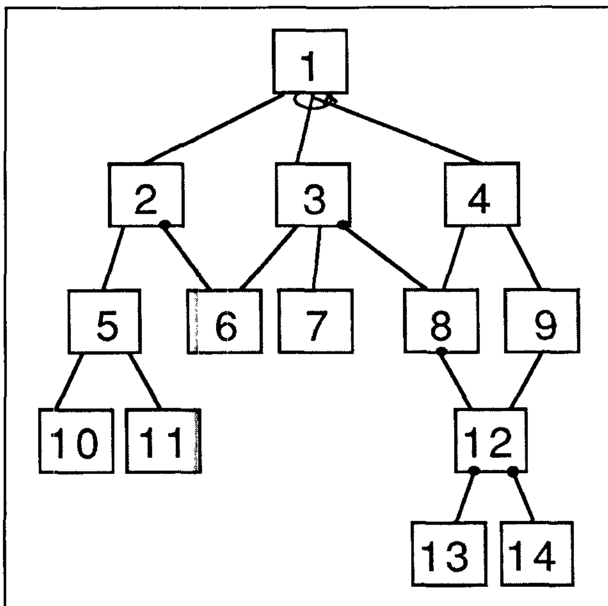

FIGURE 3. Design Tree with Complexity > 1

taken in linear combinations, yield the entire set of subtrees.

Subtrees vs. Paths

Generally, the structure chart defines how modules work together. A structure chart does not define how each individual module works. Given this premise as a basis, consider the design tree C in Figure 4, Module M conditionally invokes modules A and B. Pseudocode for module M yields the flowgraph in Figure 4. In the flowgraph, darkened nodes A and B represent the CALL's to invoke subordinate modules A and B. The cyclomatic complexity $v(M)$ is 4. However, a portion of the complexity of module M has no influence on module M's control over A and B. Module M's flowgraph can be reduced to R as shown in Figure 5. The complexity of the reduced graph, $iv(M)$, is 3. Each basis path through R yields a subtree in the design tree C. First, path EAX yields subtree MA. Second, path EBX yields subtree MB. Finally, path EX yields subtree M. Thus, the reduced flowgraph corresponding to the control structure between modules generates the subtrees with the design.

Module Design Complexity

This process produces the first design metric. *Module design complexity* of a graph G, $iv(G)$, is the cyclomatic complexity of its reduced graph. Reduction is performed to eliminate any complexity which does not influence the interrelationship between design modules.

There are four reduction rules which are applied to produce a module's design complexity. The key to the reduction principles is the existence and relationship of predicate nodes and black dots (calls), illustrated in Figure 6.

The reduction rules are as follows:

1. *Sequential black dot*: a call to a subordinate module cannot be reduced.

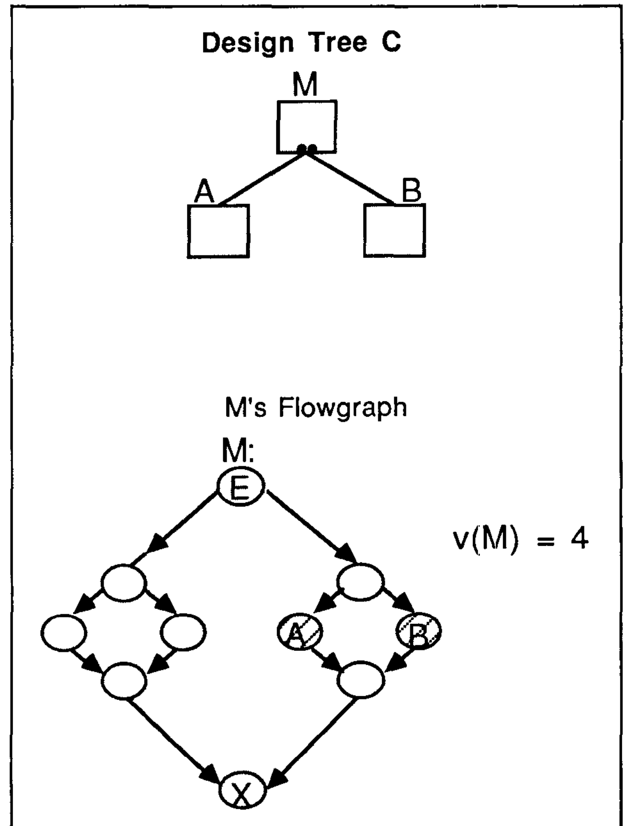

FIGURE 4. Subtrees vs. Paths

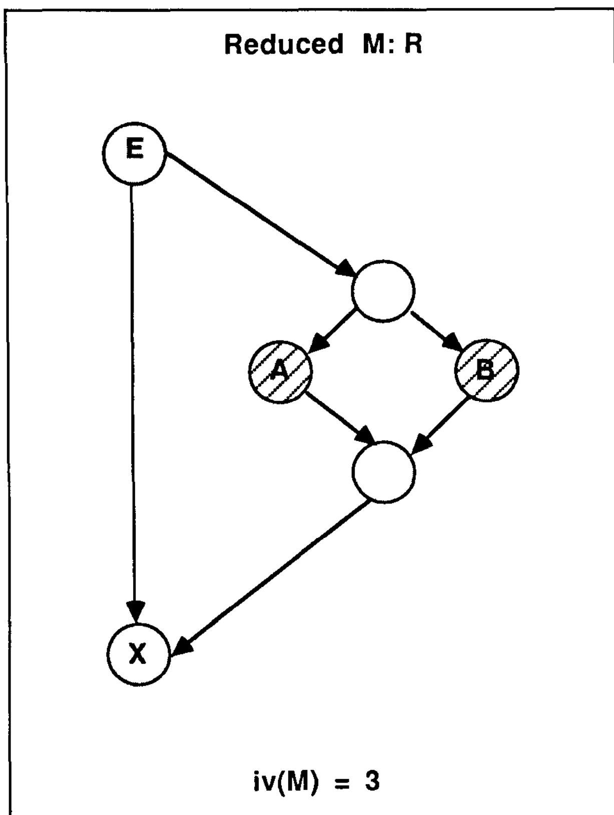

FIGURE 5. Module M's Reduced Flowgraph

2. *Sequential white dot*: a sequential node can be reduced to a single edge.
3. *Repetitive white dots*: a logical repetition without a black dot can be reduced to a single node.
4. *Conditional white dots*: a logical decision with two paths without a black dot can be reduced to one path.

Applying these four reduction principles to a module's flowgraph results in a module's primary control structure for calling subordinate modules.

The application of the first three reduction rules is straightforward; rule 4, however, will be elaborated upon. Consider the initial flowgraphs in Figure 7. Assume that this flowgraph represents module A with cyclomatic complexity of 4.

Step 1: Nodes 5 and 7 are eliminated using rule 2.
Step 2: An edge from node 2 to node 6 is removed using rule 4.
Step 3: Node 2 is eliminated using rule 2.
Step 4: Node 6 is eliminated using rule 2.
Step 5: The edge from Node 1 to Node 8 is removed using rule 4.
Step 6: Node 8 is eliminated using rule 2.

The resulting subalgorithm is a module design complexity $iv(A) = 2$. Notice that a set of 2 basis paths through the reduced graph is the desired integration test strategy.

Design Complexity

After determining the module design complexity of the individual components of a design, it is possible to calculate the design complexity of a structure chart. The *design complexity*, called S_0, of a module M is defined as

$$S_0 = \sum_{i \in D} iv(G_i)$$

where D is the set of descendants of M unioned with M. Both design and module design complexities are calculated for the modules in Figure 8. In the illustrated design, modules C, D, E, and F have module design complexity of 1, since none call descendants. Reduced subalgorithms for M, A, and B produce a module design complexity of 3, 2, and 2 respectively. Module A design complexity is its own module design complexity plus its descendant's module design complexity ($S_0(A) = iv(A) + iv(D) = 2 + 1 = 3$). The same computation is used to calculate Module M and B design complexity. Module B is the sum of its module design complexity ($iv(B) = 2$) plus its descendants ($iv(E) = 1$ and $iv(F) = 1$). Module M design complexity for this illustration is 11 which represents the complexity for the entire structure chart.

Figure 8 is a design which is a pure tree meaning there are no common modules. In this case, design complexity is upwardly additive ($S_0(M) = iv(M) + S_0(A) + S_0(B) + S_0(C)$). Designs are typically not pure trees in which case S_0 is not upwardly additive.

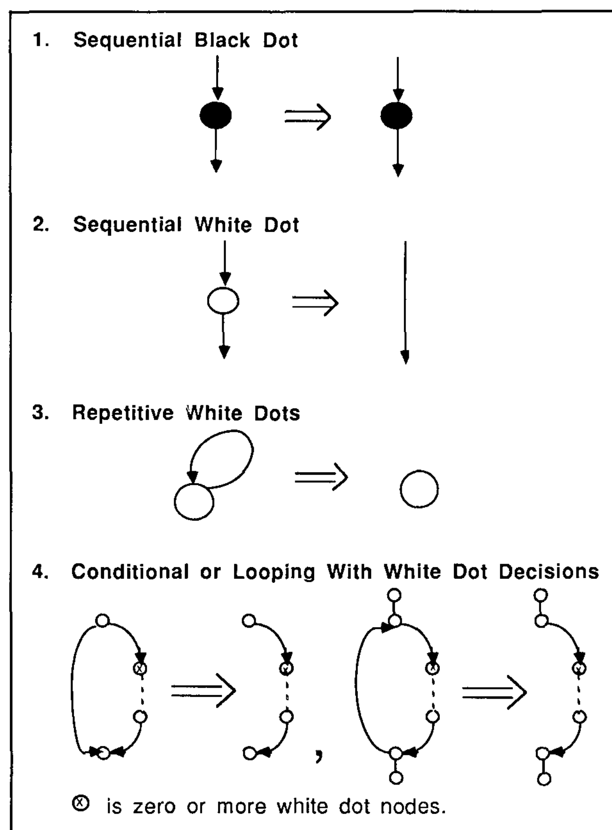

FIGURE 6. Module Design Complexity Reduction Principles

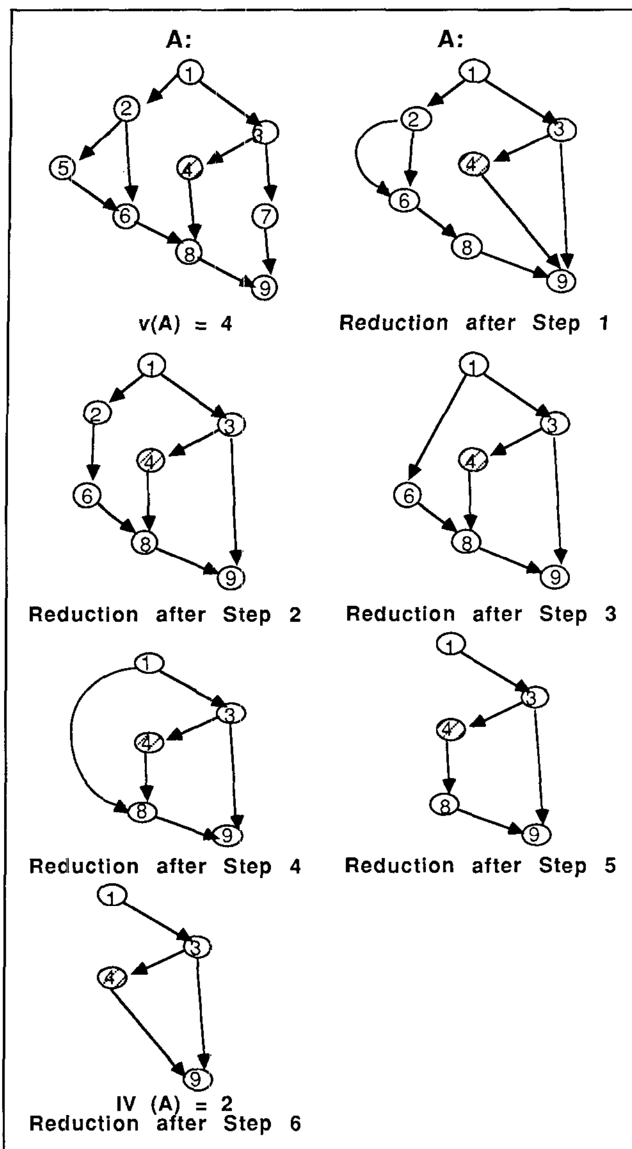

FIGURE 7. Reduction Example 1

Figure 9 is a design with a common module, E. In this case, design complexity calculations are completed by applying the formula for S_0. For module A, its design complexity is its own module design complexity plus its descendants' module design complexity

$$S_0(A) = iv(A) + iv(C) + iv(D) + iv(E)$$

$$= 2 + 2 + 1 + 1 = 6$$

Thus, S_0 can be calculated by summing the individual module design complexities.

Integration Complexity

The last design metric, *integration complexity*, is a measure of integration tests. This measure, S_1, is a function of S_0 and the number of modules, n. In general, integration complexity is

$$S_1 = S_0 - n + 1$$

In Figure 9, $S_1 = 4$ indicating there should be 4 integration tests to qualify the design. More detailed discussion of S_1 follows in the section on a test methodology.

The importance of the integration complexity is inherent to its testing requirements. Studies have shown that integration errors are as much as 30 times more costly to fix than unit errors. Most of the errors found in the later stages of development are integration errors. Consequently, properties of integration complexity include the following:

1. S_1 should be used to qualify the integration tests on a project and their validation should be part of an early validation of the design.
2. S_1 quantifies a basis set of integration tests.
3. Each S_1 test validates the integration of several modules.

Each of these properties plays an important role in the development of a testing methodology.

Properties of Design Complexity

Design complexity, S_0, as a measure, exhibits a number of important properties. First, S_0 is bounded as:

$$n \leq S_0 \leq \sum_i^n v(G_i)$$

where n = number of modules in the design

A design where $S_0 = n$ always behaves the same way. There are no conditional calls to subordinate modules. Therefore, in this case, $iv(G) = 1$ for each module. On the other hand, if every decision within the design affects intermodule flow, then $S_0 = \sum v(E_i)$, or S_0 is equal to the summation of cyclomatic complexity for all modules.

The concept of design predicates can be used to calculate design complexity. At the design stage, when the structure chart is the only product to analyze, design predicates are a feasible way to bound design complexity. With this approach, a design is evaluated without

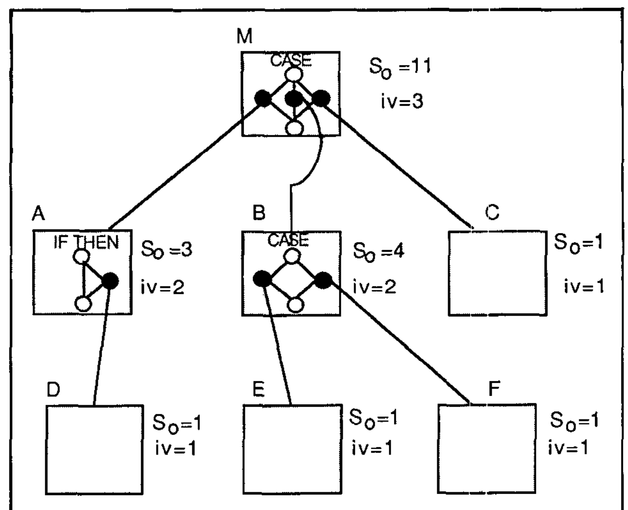

FIGURE 8. Additive Design Complexity

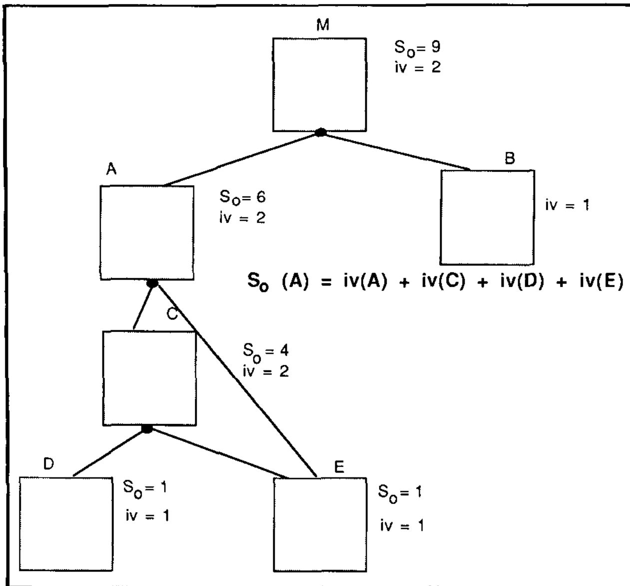

FIGURE 9. Nonadditive Design Capacity

looking at the internal pseudocode for each module. Generally, the convention is to show a conditional CALL from a superordinate to a subordinate module with a dot, as illustrated in Figures 3 and 10. Repetition is illustrated with an arc as shown for modules 3 and 4 in Figure 3. In these cases, we define condition and repetition as design predicates. Moreover, for a module M, iv(M), is equal to the number of design predicates plus one.

There are a number of additional properties of design complexity.

1. Adding a module to a design increases S_0 by at least 1.
2. Adding a decision to call a module increases S_0 by 1.
3. Multiple calls to a module M (reusable code) reduce S_0 by S_0(M).
4. S_0 is defined for any subdesign D (a module M and all its descendants) by:

$$S_0(D) = \sum_{i \in D} iv(G_i)$$

When two subsystems A and B are integrated into a larger system M the following holds:

a. The modules of M are determined by the union denoted as ∪ of A and B:

$$M = A \cup B$$

b. The number of modules (denoted n(M)) is $n(M) = n(A) + n(B) - n(A \Omega B)$ where the intersection, denoted as Ω, is the number of modules common to A and B.

c. The design complexity S_0(M) is determined by

$$S_0(M) = S_0(A) + S_0(B) - S_0(A \Omega B)$$

where $S_0(A \Omega B)$ is the design complexity of modules common to A and B.

d. The number of integration tests S_1(M) is

$$S_1(M) = S_1(A) + S_1(B) - S_1(A \Omega B)$$

where $S_1(A \Omega B)$ is the integration complexity of modules common to A and B. Figure 10 illustrates the integration of two systems, M and N.

Assume two designs M and N are integrated through an interface at Module B. Systems M and N have S_0 of 9 and 9 respectively. For the new system, M^1, the union of M and N yields

$$\begin{aligned} S_0(M \cup N) &= S_0(M) + S_0(N) - S_0(M \Omega N) \\ &= 9 + 9 - 0 \\ &= 18 \\ S_1(M \cup N) &= S_1(M) + S_1(N) - S_1(M \Omega N) \\ &= 5 + 5 - 1 \\ &= 9 \end{aligned}$$

Proper utilization of these properties provides the designer with a tool to efficiently package and test the design.

A STRUCTURED INTEGRATION TEST METHODOLOGY

Structured integration testing utilizes the design metrics developed in the previous section to produce a testing strategy derived from the design specification. Generally, the methodology is applied at two levels:

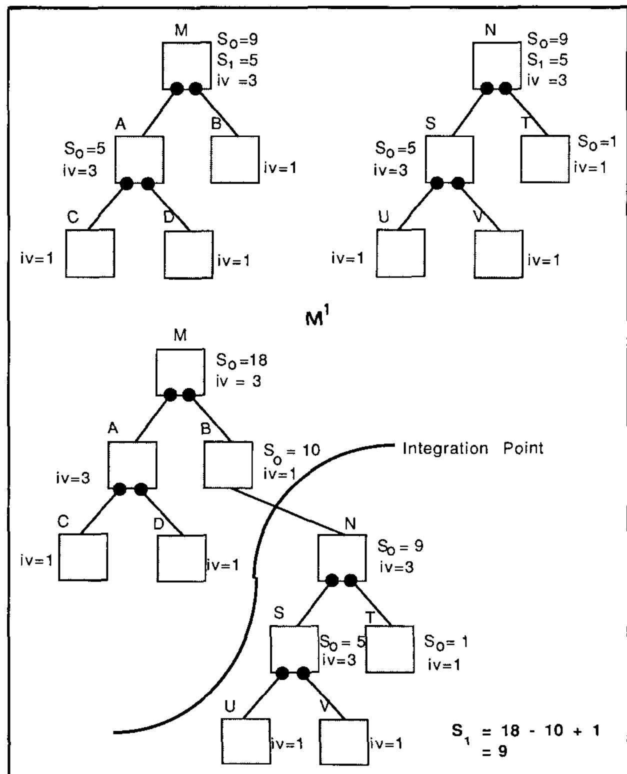

FIGURE 10. Integrated Properties of S_0

module integration testing and design integration testing.

Module Integration Testing

The scope of module integration testing is a module and its immediate subordinates. This testing requirement is a particularly significant activity when packaging modules into single programs. Generally, the methodology is implemented using three steps.

1. Apply the reduction rules to the selected module.
2. The cyclomatic complexity of the subalgorithm is the module design complexity of the original algorithm. Module design complexity determines the number of module integration tests required to qualify integration of the module with its immediate subordinates.
3. The baseline method applied to the subalgorithm yields the design subtrees and the module integration tests.

In order to illustrate the methodology, review the design shown in Figure 9. Assume that module C's integration test strategy with modules D and E is to be developed. The flowgraph in module C represents the subalgorithm after applying the reduction rules (step 1). Module design complexity, iv(C), is 2 (step 2). Therefore, two tests are required to qualify module C's integration with modules D and E. The design subtrees to be executed are CD and CE. Moreover, the module integration test procedure should be applied to modules A and M since they are superordinate functions.

Design Integration Testing

A second level of testing is derived from integration complexity. Since integration complexity quantifies a basis set of integration tests, it also can be used to establish the integration test strategy of design. Consequently, it determines the level of effort for testing a design. Generally, the methodology is applied using the following steps.

1. Calculate iv for each module.
2. Calculate S_0 for each module.
3. Using the top level module, calculate S_1.
4. S_1 is the number of basis subtrees required to qualify the design.
5. Build a path matrix, which is $S_1 \times n$, to establish the basis set of subtrees.
6. Identify and label each predicate on the design tree.
7. Place the predicate label above each column in the path matrix corresponding to the module it influences.
8. Apply the baseline method to the design to complete the matrix. Use 1 to indicate a module is executed and 0 to indicate it is not executed.
9. Identify the subtrees for the matrix.
10. Identify the conditions which drive the subtrees.
11. Build the corresponding test cases for each subtree.

In order to illustrate the entire methodology, consider the design structure in Figure 11. The design is composed of six modules, and it contains two design predicates. Since module D is a shared module, the design structure is nonadditive. After applying steps 1 through 4, the iv's, S_0's, and S_1 for the design yield design and integration complexity of 8 and 3.

The path matrix (steps 5 through 7) for the integration tests is built as a 3×8 matrix as shown in Figure 12. In this example, the two design predicates are labeled p_1 and p_2, and they have been placed over modules A and E, since they affect module execution. Applying the baseline method, a baseline is selected which executes all the modules in the system. Consequently, 1's are placed under each module to indicate that it is executed. Alternate subtrees are found by evaluating the predicates on the baseline, generally working from left to right. When predicate, p_1, is evaluated on the baseline, the condition is negated. Since p_1 was previously set to execute module A, it now is set to not execute module A. Therefore, each module subordinate to module A alone will not be executed. For the third subtree, predicate, p_2, is evaluated. In this example, it also was set to execute in the baseline. By negating p_2 and identifying its subordinate modules, the third subtree is defined. As a result, the path matrix would appear as shown in Figure 12.

The subtrees are identified, and the conditions which drive the subtrees are established using the completed path matrix. Assume that p_1 and p_2 are simple predicates such as W = X and Y = Z, respectively. Then, the integration test requirements for the design would contain three subtrees with their associated conditions (steps 9 and 10).

It is at the architectural level that the design metric can provide additional development support. Historically, there have been a number of design approaches such as top down, bottom up, critical piece first, and others. These approaches can be used to evaluate the testing requirements and establish a test plan to support the selected design approach. We can return to the example in Figure 11 and assume that a baseline is chosen given a critical piece first design approach. If mod-

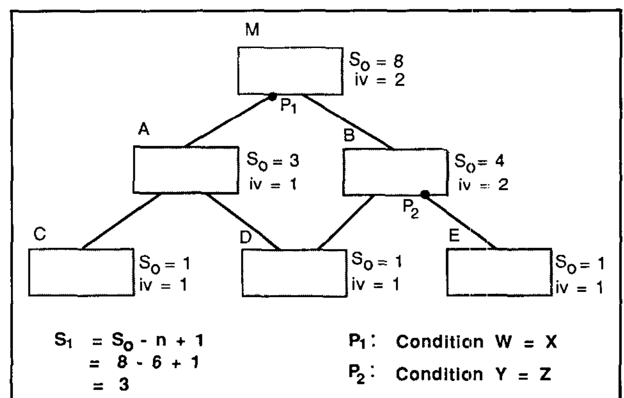

FIGURE 11. Design Integration Test Example

		p_1				p_2		
PATH	M	A	B	C	D	E	Test Condition	Expected Execution
Baseline	1	1	1	1	1	1	W = X and Y = Z	Invoke A & E
Subtree2	1	0	1	0	1	1	W ≠ X and Y = Z	Invoke E, not A
Subtree3	1	1	1	1	1	0	W = X and Y ≠ Z	Invoke A, not E
Relative Frequency	3	2	3	2	3	2		

FIGURE 12. Integration Path Test Matrix

ule E is critical due to performance needs of a real-time system, the baseline can be selected to isolate, as much as possible, module E. Only those unconditional modules and module E's superordinate modules are included. Applying the testing methodology, with a focus on module E, generates an alternative path matrix.

Readers should note in Figure 13 that the relative frequency of execution of a number of modules during the test is altered. In this simple illustration, modules A and C are not executed as frequently. Given individual design approaches, the baseline can be established to focus on key components during the actual tests. Consequently, the software designer can implement a testing strategy best suited for the selected design approach and derived from the design specification.

SUMMARY

Quantifying the complexity of a design provides the system developer metrics which represent an important management tool. Our design metrics—module design complexity, design complexity, and integration complexity—are three such measures. These metrics are derivatives of the well-founded cyclomatic complexity. As the decision structure of a program is an important indication of program complexity, the design structure which specifies the relationship among modules in a design also defines the overall design complexity. The reduction technique used to determine design complexity addresses the need to test how modules work together rather than how each module works. More importantly, the quantification can be used to drive the testing process at two levels: individual modules and overall design framework. Calculation of design metrics represents a new management and testing tool previously unavailable to software developers.

Intuitively, the design complexity metrics exhibit a number of desired properties which support their applicability.

- *The metric intuitively correlates with the difficulty of comprehending a design.* When we view large complicated designs, the metric should yield a high number. Designs we intuitively deem as simple should have a relatively low number. As a minimum, the complexity metric will certainly initiate hot debate about particular designs and design methodologies in general.
- *The metric is objective and mathematically rigorous.* In addition to being intuitive, it is critical that the metric be objective. The same design viewed at two different times or by two people should yield the same complexity. If it is not objective, the various vested interests involved in a development job will no doubt have differing interpretations.
- *The metric should be related to the effort to integrate the design.* The most costly activity associated with a de-

		p_1				p_2
Path	M	A	B	C	D	E
Baseline	1	0	1	0	1	1
Subtree2	1	1	1	1	1	1
Subtree3	1	0	1	0	1	0
Relative Frequency	3	1	3	1	3	2

FIGURE 13. Critical Module Integration Test Matrix