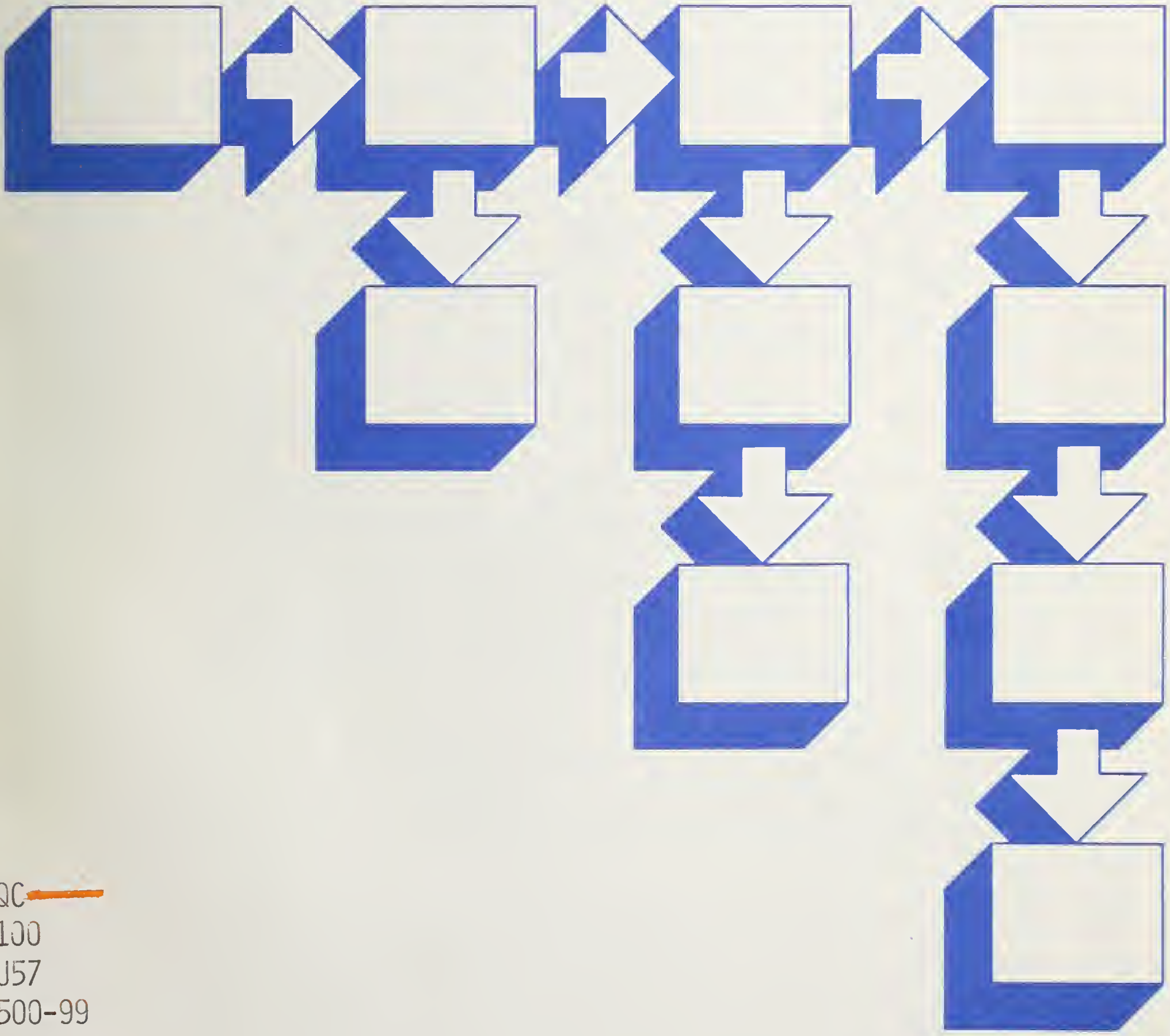
National Bureau
of Standards

Computer Science
and Technology

NBS Special Publication 500-99

Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric



Computer Science and Technology

NBS Special Publication 500-99

Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric

Thomas J. McCabe

McCabe & Associates, Inc.
5550 Sterrett Place
Columbia, MD 21044

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued December 1982

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

Library of Congress Catalog Card Number: 82-600651
Natl. Bur. Stand. (U.S.) Spec. Publ. 500-99, 72 pages (Dec. 1982)
CODEN: XNBSAV

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1982

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $5.00
(Add 25 percent for other than U.S. mailing)

TABLE OF CONTENTS

	Page
Abstract and Acknowledgements	viii
PART 1. Measuring and Limiting Program Complexity	
I. Introduction	1
-Limiting Program Complexity with a design structure	1
- Assuring testability of design modules	2
II. The Complexity Measure	4
- How physically small programs can have high complexity and how physically large programs can have relatively low complexity	4
- Graph theory representation of program logic	4
- Graph theory notions:	4
- Cyclomatic number of graphs	4
- Strongly connected graphs	4
- The cyclomatic number of a control graph as its complexity	6
- Basis paths	6
- Linear combinations of basis paths generating all paths	6
III. Examples	8
- The automation of the complexity calculation	8
- Several examples of live programs in the order of increasing complexity	9
- Examples of programs that violate the complexity axioms	11
IV. Simplification	13
- Computing complexity by counting splitting nodes in the control graph	13
- Computing complexity by counting conditions	14
- Compound predicates and case statements	14
- Euler´s formula: computing complexity by counting the regions in a planar graph	15

PART 2. Testing Methodology

V. The Structured Testing Criteria 16

- The criteria 16
- A FORTRAN example with bugs 18
- Example test data 19
- Test coverage hierarchy 19
- Experience and observations 20

VI. Identifying Test Paths: The Baseline Method 21

- The procedure to generate test paths and data 21
- An example 22
- The FORTRAN program and its bug revisited 23

VII. Integration Testing 25

- Definition of a module 25
- Measuring design complexity 26
- Quantification of the integration effort 26
- Applications 27

VIII. The Reduction Technique 28

- Actual complexity 28
- Reducing program complexity: removing untested decisions 28
- Examples: the FORTRAN program revisited yet again 29

PART 3. Maintenance Methodology

IX. Essential Complexity 33

- Structured control flow constructs 33
- The modularization process from a graph reduction viewpoint 33
- Removing one entry/one exit subgraphs 33
- The measurement of structuredness: essential complexity 35
- Examples 35

X. Program Modification 37

- The cost of maintenance 37
- Classes of change 37
  - Changes to functional statements: the retest criteria 37
  - Modifying control flow 38
    - Example of catastrophic change 38
    - Retest methodology: 39
      - Evaluation of essential complexity 39
      - Retest quantification 39

XI. Summary 42

- Tieing together all the different pieces 42

Appendix A: Empirical Evidence A-1

Appendix B: A FORTRAN Example B-1

References R-1

LIST OF FIGURES

		Page
Figure I-1	Hierarchy Chart	1
Figure II-1	Control Graph G	5
Figure II-2	Control Graph G´	5
Figure II-3	Control Graph G with Numbered Paths	6
Figure II-4	Basis for Control Graph G	7
Figure III-1	Control Graph with Complexity 3	9
Figure III-2	Control Graph with Complexity 4	9
Figure III-3	Control Graph with Complexity 4	9
Figure III-4	Control Graph with Complexity 5	9
Figure III-5	Control Graph with Complexity 5	10
Figure III-6	Control Graph with Complexity 5	10
Figure III-7	Control Graph with Complexity 5	10
Figure III-8	Control Graph with Complexity 6	10
Figure III-9	Control Graph with Complexity 6	11
Figure III-10	Control Graph with Complexity 7	11
Figure III-11	Control Graph with Complexity 9	11
Figure III-12	Control Graph with Complexity 12	11
Figure III-13	Control Graph with Complexity 47	12
Figure IV-1	Splitting node	13
Figure IV-2	Control Graph G	14
Figure IV-3	IF-THEN-ELSE Control Graph	14
Figure IV-4	CASE Control Graph	15
Figure IV-5	Plane Graph	15
Figure V-1	FORTRAN Example Graph	18
Figure V-2	Flow Graph	19
Figure VI-1	Control Graph G	22
Figure VI-2	Flow Graph	23
Figure VII-1	Design Hierarchy	25
Figure VII-2	Graph of Design Hierarchy	26
Figure VIII-1	Program Control Graph	29
Figure VIII-2	FORTRAN Program	30
Figure VIII-3	FORTRAN Program Revised	31
Figure IX-1	Structured Control Constructs	33
Figure IX-2	Reducing Complexity Example 1	34
Figure IX-3	Reducing Complexity Example 2	34
Figure IX-4	Reducing Complexity Example 3	35
Figure IX-5	Non-Reducible Control Graph	35
Figure IX-6	FORTRAN Example Rewritten	36
Figure X-1	Program Modification	37
Figure X-2	Program Patch	38
Figure X-3	Branching Graph	40
Figure X-4	Branching Graph Amplified	41
Figure XI-1	Structured Testing Technique	43

		Page
Figure B-1	Blackjack Specification	B-2
Figure B-2	Blackjack Graph	B-7
Figure B-3	Test 1	B-8
Figure B-4	Test 2	B-8
Figure B-5	Test 3	B-9
Figure B-6	Test 4	B-9
Figure B-7	Test 5	B-10
Figure B-8	Test 6	B-10
Figure B-9	Test 7	B-11
Figure B-10	Test 8	B-11
Figure B-11	Test 9	B-12
Figure B-12	Test 10	B-12
Figure B-13	Test 11	B-13

Abstract

Various applications of the Structured Testing methodology are presented. The philosophy of the technique is to avoid programs that are inherently untestable by first measuring and limiting program complexity. Part 1 defines and develops a program complexity measure. Part 2 discusses the complexity measure in the second phase of the methodology which is used to quantify and proceduralize the testing process. Part 3 illustrates how to apply the techniques during maintenance to identify the code that must be retested after making a modification.

Keywords: measures; metric; program complexity; software testing; structured testing.

ACKNOWLEDGEMENTS

This report was produced by Thomas J. McCabe for the National Bureau of Standards Institute for Computer Sciences and Technology under U.S. Department of Commerce Contract AE8892.

Acknowledgement and thanks are appropriate for the following reviewers who donated their time and energy to critiquing the document:

John Bailey
Rich Fecher
Karen Hedelius
Raymond C. Houghton, Jr.
Elizabeth Kruesi
David Salazar
Tanya Spitzer

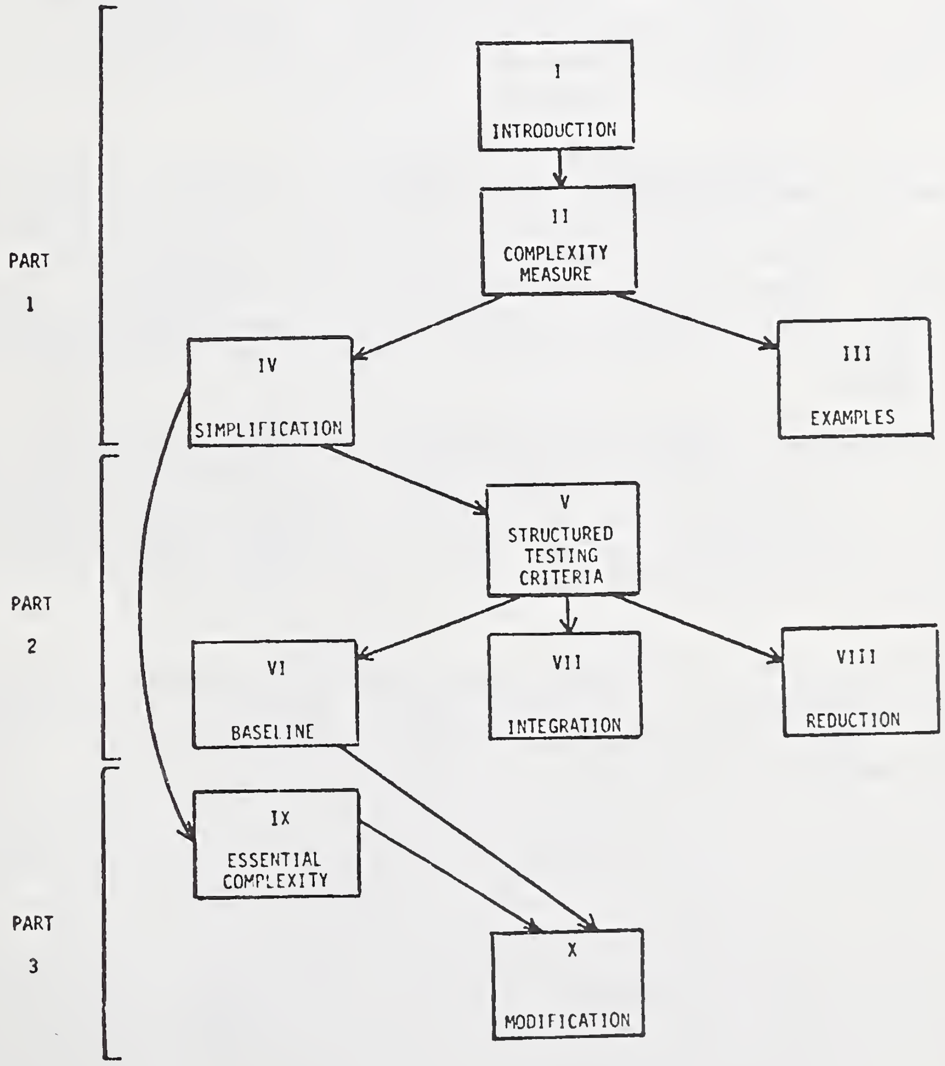
Comments pertaining to the technical content are solicited and should be directed to:

Systems and Software Technology Division
Room B266 Bldg. 225
National Bureau of Standards
Washington, D.C. 20234

PREFACE: How To Read This Document

The main advice to the reader is "Don´t become discouraged with Section II." Section II deals with the derivation of a program complexity measure from graph theory; it contains mathematical theorems and notation. While it is critical to include this material to establish a mathematical basis for program complexity, it is possible that it may frustrate some readers. Those readers whose interests lie solely in the application of the theory (as opposed to the development of the theory) may safely skip Section II. The rest of the paper contains the operational procedures which are directly applicable to the programming process.

The diagram below shows the dependencies between the various sections in the paper:



If you are a Programmer, and your interest at this time is mainly how to limit and control complexity and not so much the testing process, then the sections to concentrate on are I-IV. Start with Section I, read as much of Section II as you want to. If you get frustrated, instead of filing the document, skip to Section III where many examples are presented; then digest Section IV which explains how to compute the complexity in programming terms.

If your primary concern is the testing process, then the sections directly applicable are V through VIII. They contain both the criteria and procedures to carry out the structured testing process. If the project's design is not complete and you have some control over the modularization process, then also read Sections I-IV. If you are having trouble with an existing design, concentrate on Sections V, VI, and VIII.

If your job is software maintenance you should concentrate on Sections IX and X. They contain the operational steps for evaluating modifications and producing modification test data. It will be necessary, however, to understand Sections V, VI, and VIII to carry out this testing.

If you are a Project Leader or Manager, and your concern is with development methodology, test plans, and quality assurance, you need to understand the complete approach. The document will give you the essence of the overall methodology; the discrete operational steps in the methodology are summarized in Section XI. There is a lot of substance here for quality assurance and project planning -- don't let the theoretics in Section II abort your journey.

If your concern is Quality Assurance and Methodology Standards, Appendix A, which presents live, real world data, may be of interest. Presented are a number of studies that independently validated various aspects of the complexity measure. This validation was performed by extracting empirical data from real projects.

For readers already familiar with and practicing the structured testing methodology, we recommend Section XI - SUMMARY. The Summary lists the operational steps which can be incorporated into an overall project plan.

Part 1. MEASURING AND LIMITING PROGRAM COMPLEXITY

I. INTRODUCTION

This document discusses a software quality metric, the cyclomatic complexity metric, and how it may be applied to software testing. Discussions of other metrics and methodologies may be found in [PAIG], [HALS], [KOLE], [CHAP], [CHEN], [GILB], [JELI], [MCAL], and [MOHA].

The testing method described in this paper is performed in two phases. The first phase is to quantify and to limit the complexity of a program to permit thorough testing. This quantification is accomplished using a complexity measure that suggests a minimum on the number of distinct paths that must be tested. The second phase is the actual testing where the number of test paths is forced to meet the complexity measure.

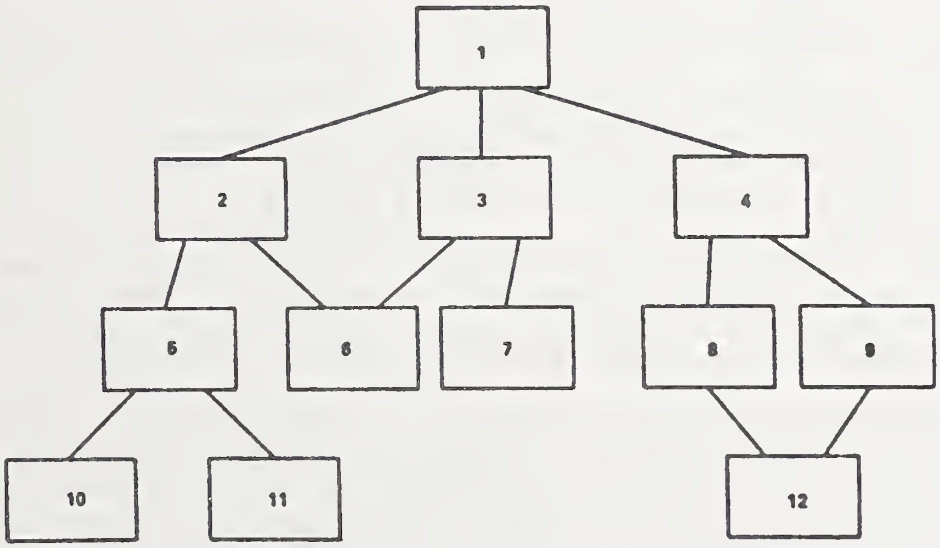


Figure I-1 Hierarchy Chart

Attention is first given to limiting the complexity of a program to assure testability. This viewpoint, however, is an oversimplification of the problem. A "program" of any reasonable size is typically developed and represented as the interaction of several procedures, subroutines, or paragraphs. In designing a "program," the design stage typically results in a hierarchy chart as shown in Figure 1-1, that explicitly shows the decomposition of a program´s functions into distinct modules. If the programming language used is FORTRAN, the top module typically is the mainline code and each of the other modules are usually subroutines, functions, or externally-called programs.

If COBOL is the programming language, the top is mainline code and the other modules are typically paragraphs.

The design modularization is critical to the quality of the end product. Each of the modules should have the following properties:

- Testability - the testing effort to validate each module should be manageable.

- Comprehensibility - each of the modules should be readable and understandable.

- Reusability - if the modules are well-defined, they may be reusable in a different system.

- Maintainability - the job of modifying and retesting each of the modules in the operational phase should be manageable.

This design modularization process is governed by two principles:

- The functional decomposition represented by the design hierarchy should result in several independent, cohesive modules which provide a natural decomposition of the problem.

- The modularization must also be governed by "size" or "testability" of each of the modules. That is, the modularization should avoid modules that are inherently so complex that they are untestable.

The process of evaluating a module´s cohesiveness and relative independence is largely a heuristic and creative process. Currently, it is virtually impossible to measure these attributes of a design. It is important to emphasize the second principle (testability of the modules) so, in fact, the design can be reliably implemented. Whereas a module´s cohesiveness or independence is not measurable, we will in this document, present techniques that quantify the testability attributes of the modules. This quantification of the testing effort can then be incorporated into test plans and a quality assurance phase that the product would have to satisfy before being accepted.

The structured testing principles in this paper will constrain a program´s complexity by limiting the complexity of each of its internal modules to the point that they can each be tested rigorously. If it is found that a particular module exceeds the complexity threshold, a further design refinement would be required. For example, if module 11 in Figure 1-1 is found to be too complex, the creation of several testable modules below 11 would be required. So, in short, a program´s testability is assured by limiting the complexity of each of the modules within the program.

II. THE COMPLEXITY MEASURE

The complexity measure will limit the number of independent paths in a program at the design and coding stages so the testing will be manageable during later stages. One of the reasons for limiting independent paths, instead of a limitation based on the length of a program, is the following dilemma: a relatively short program can have an overwhelming number of paths. For example, a 50-line FORTRAN program consisting of 25 IF statements in sequence, will have 33.5 million potential control paths. The approach taken here is to limit the number of basis (or independent) paths that will generate all paths when taken in combination.

One definition and one theorem from graph theory are needed to develop these concepts. In this section we will treat graphs with only one connected component - Section VII will deal with the more general case. See [BERG] for graph theory concepts and a more formal treatment of connected components.

<u>Definition 1</u>. The cyclomatic number $v(G)$ of a graph G with n vertices, e edges, and 1 connected component is:

$$v(G) = e - n + 1.$$

<u>Theorem 1</u>. In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent paths.

The application to computer programs will be made as follows: given a program module, associate with it a graph that has unique entry and exit nodes; each node in the graph corresponds to a block of statements where the flow is sequential and the edges represent the program's branches taken between blocks. This graph is classically known as the control graph [LEGA75]; and it is assumed that each node can be reached by the entry node and each node can reach the exit.

For example, the control graph in Figure II-1 has seven blocks ((a) through (g)), entry and exit nodes (a) and (g), and ten edges.

To apply Theorem 1, the graph must be strongly connected which means that given two nodes (a) and (b), there exists a path from (a) to (b) and a path from (b) to (a). To satisfy this, we associate an additional edge with the graph which branches from the exit node (g) to the entry node (a) as shown in Figure II-2.

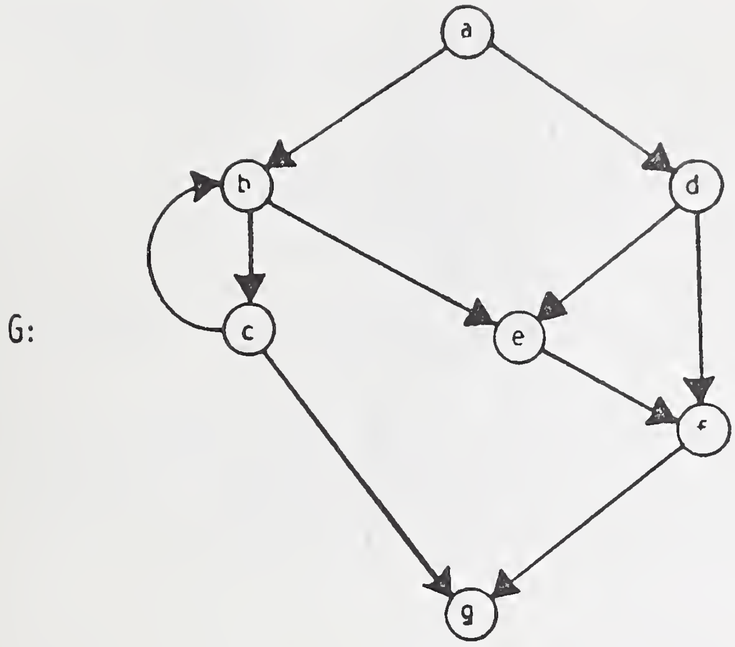


Figure II-1 Control Graph G

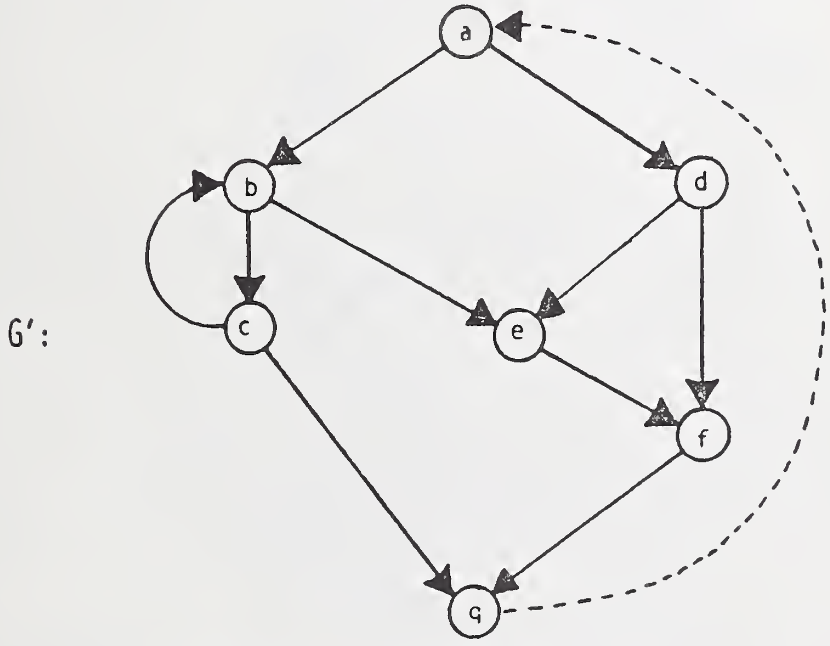


Figure II-2 Control Graph G′

Theorem 1 now applies, and it states that the maximal number of independent paths in G′ is 11 - 7 + 1. (G has only one connected component so we set p = 1.) The generalized case where p > 1 is used for design complexity, see Section VII. The implication, therefore, is that there is a basis set of five independent paths that when taken in combination, will generate all paths. For example, the set of five paths shown below form a basis.

bl:	abcg
b2:	a(bc)*2g
b3:	abefg
b4:	adefg
b5:	adfg

Note: The notation (bc)*2 means iterate the (bc) loop twice.

If any arbitrary path is chosen, it should be equal to a linear combination of the basis paths bl - b5. For example, the path abcbefg is equal to b2 + b3 - bl, and path a(bc)*3g equals 2 * b2 - bl. To see this, it is necessary to number the edges in G Figure II-3 and show the basis as edge vectors Figure II-4.

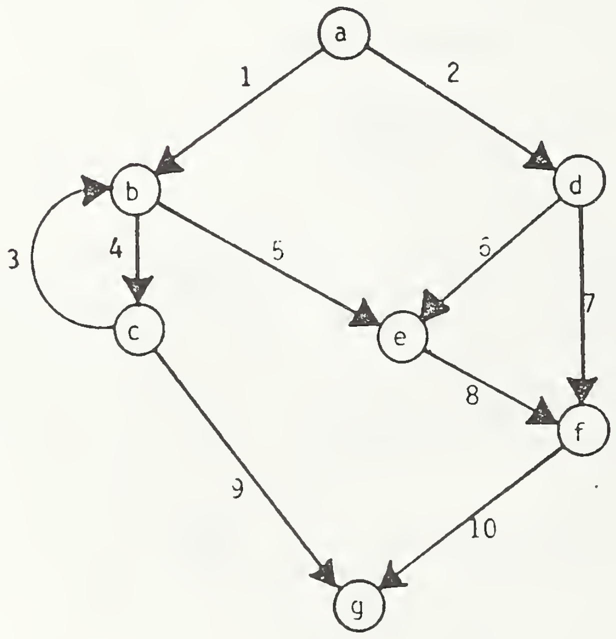


Figure II-3 Control Graph G with Numbered Paths

The path abcbefg is represented as the edge vector shown in Figure II-4, and it is equal to b2 + b3 - bl where the addition and subtraction are done component-wise. In similar fashion, the path a(bc)*3g shown in Figure II-4 is equal to 2 * b2 - bl.

Basis

	1	2	3	4	5	6	7	8	9	10
b1	1	0	0	1	0	0	0	0	1	0
b2	1	0	1	2	0	0	0	0	1	0
b3	1	0	0	0	1	0	0	1	0	1
b4	0	1	0	0	0	1	0	1	0	1
b5	0	1	0	0	0	0	1	0	0	1
abcbefg	1	0	1	1	1	0	0	1	0	1
a(bc)*3g	1	0	2	3	0	0	0	0	1	0

Figure II-4 Basis for Control Graph G

It is important to notice that Theorem 1 states that G has a basis set of size five but it does not tell us which particular set of five paths to choose. For example, the following set will also form a basis.

adfg
abefg
adefg
a(bc)*3befg
a(bc)*4g

When this is applied to testing, the actual set of five paths used will be dictated by the data conditions at the various decisions in the program. The Theorem, however, guarantees that we will always be able to find a set of five that form a basis.

It should be emphasized that the process of adding the extra edge to G was performed only to make the graph strongly connected so Theorem 1 would apply. When calculating the complexity of a program or testing the program, the extra edge is not an issue, but rather it is reflected by adding 1 to the number of edges. The complexity v, therefore, is defined as:

$$v = (e + 1) - n + 1$$

or more simply

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

III. EXAMPLES

Several actual control graphs and their complexity measures are presented in Figures III-1 through III-12, to illustrate these concepts. These graphs are from FORTRAN programs on a PDP-10. The programs were analyzed by an automated system, called FLOW, that recognizes the blocks and transitions in a FORTRAN program, computes the complexity, and draws the control graphs on a DATA DISK CRT. The straight edges represent downward flow (e.g., in Figure III-2 below, the line between (2) and (3) means that (2) branches to (3)). The curved arcs represent backward branches (e.g., in Figure III-2 (5) branches back to (2)).

The graphs in Figures III-1 through III-12 are presented in the order of increasing complexity to suggest the relationship between the complexity numbers and our intuitive notion of the complexity of the graphs.

One essential ingredient in any testing methodology is to limit the program logic during development in order that, first, the program can be understood, and second, the amount of testing required to verify the logic is not overwhelming. To illustrate what goes wrong when this principle is violated, the graph in Figure III-13 is presented. According to its author, the program below is "one of my simpler programs; it required four or five tests to verify."

The size of the program that gave rise to the graph in Figure III-13 is only 70 lines of source code. The size of several of the programs producing the 12 previous graphs exceeded 70 lines. In practice, often large programs have low complexity and small programs have very high complexity. Because of this, the common practice of attempting to limit complexity by controlling only how many pages a routine will occupy is entirely inadequate. This complexity measure has been used in production environments by <u>limiting the complexity of every module to 10</u>. Programmers have been required to calculate the complexity as they develop routines, and if it exceeded 10, they were required to recognize and modularize subfunctions or redesign the software. The only situation where the limit of 10 seemed unreasonable and an exception allowed, is in a large CASE statement where a number of independent blocks follow a selection function. (reference Figure III-8 for an example of a CASE statement graph).

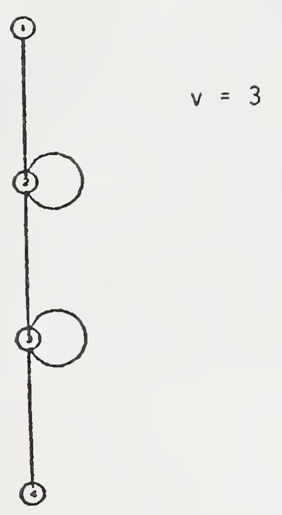


Figure III-1
Control Graph with
Complexity 3

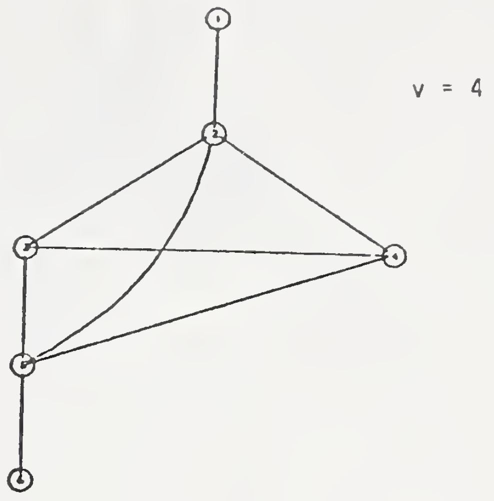


Figure III-2
Control Graph with
Complexity 4

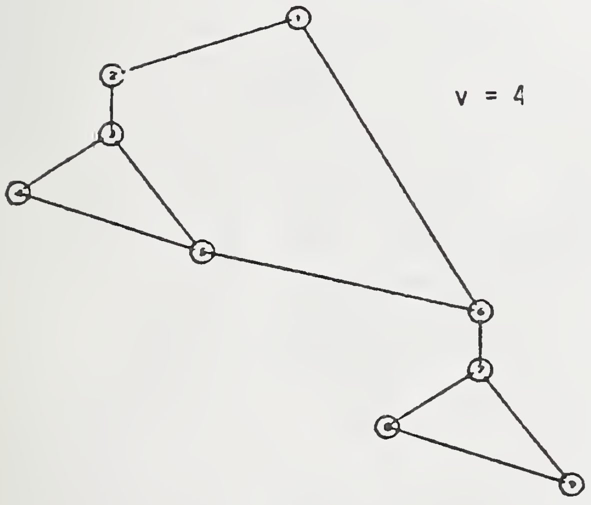


Figure III-3
Control Graph with
Complexity 4

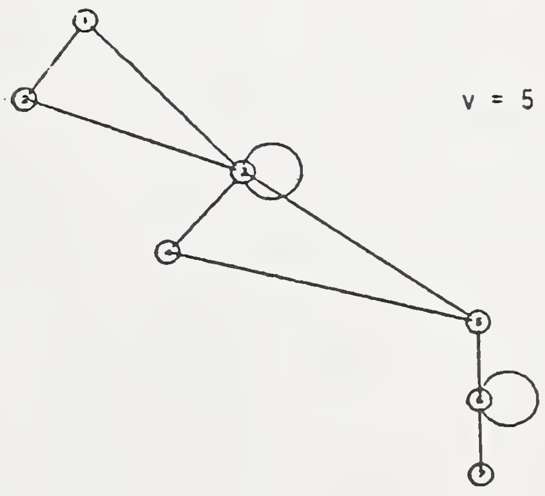


Figure III-4
Control Graph with
Complexity 5

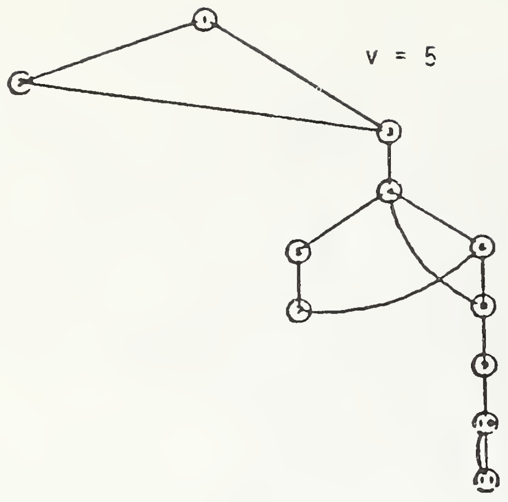


Figure III-5
Control Graph with
Complexity 5

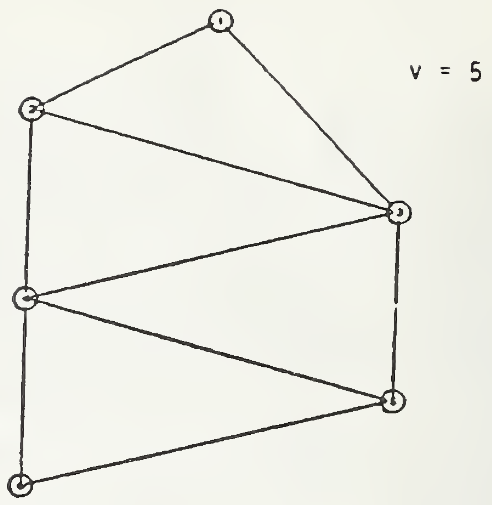


Figure III-6
Control Graph with
Complexity 5

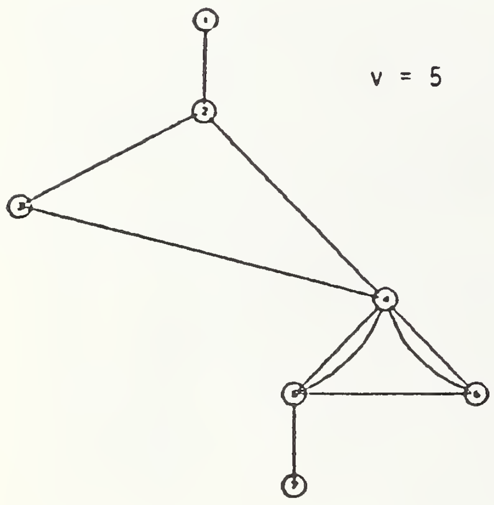


Figure III-7
Control Graph with
Complexity 5

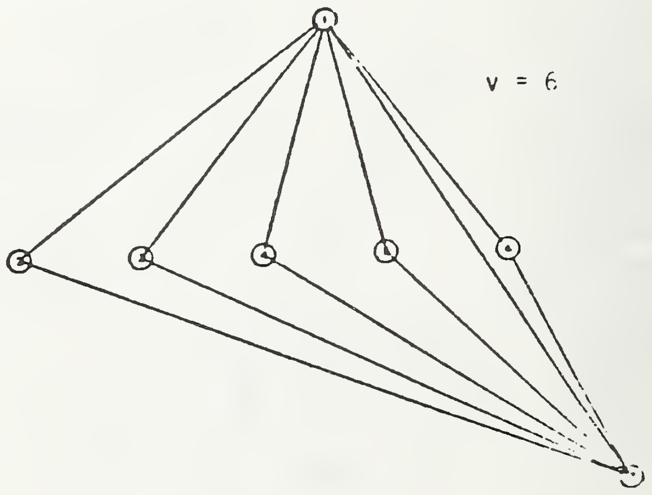


Figure III-8
Control Graph with
Complexity 6

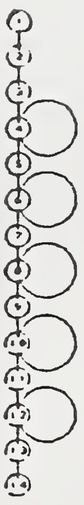


Figure III-9
Control Graph with
Complexity 6

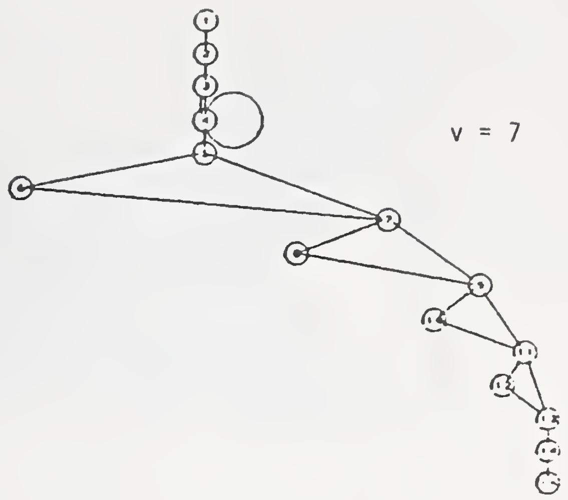


Figure III-10
Control Graph with
Complexity 7

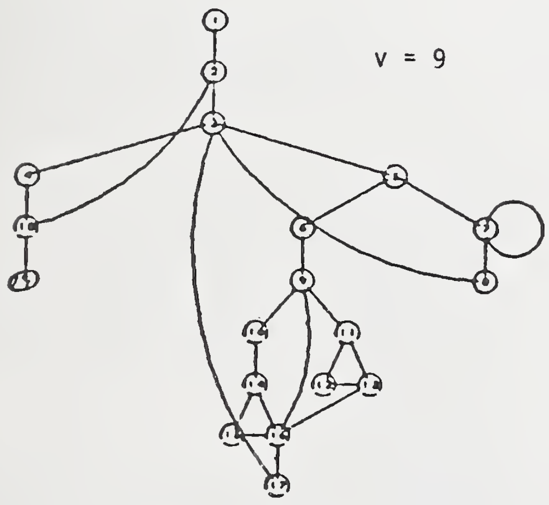


Figure III-11
Control Graph with
Complexity 9

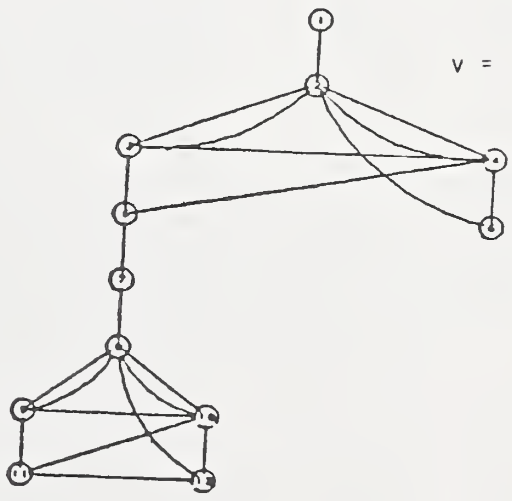


Figure III-12
Control Graph with
Complexity 12

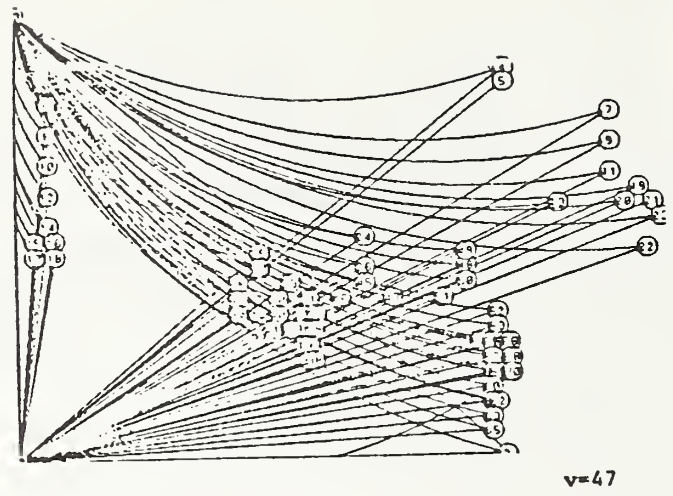


Figure III-13 Control Graph with Complexity 47

IV. SIMPLIFICATION

Since the calculation of e - n + 2 is error-prone and tedious, alternative methods of calculating complexity are presented. The results are presented without proof. The interested reader is referred to [MCCA] for proofs.

The first simplification allows the calculation of v by counting "splitting nodes" in the graph. A splitting node has more than one outcome and is associated with some conditional in the source program. A splitting node in the control graph is illustrated in Figure IV-1.

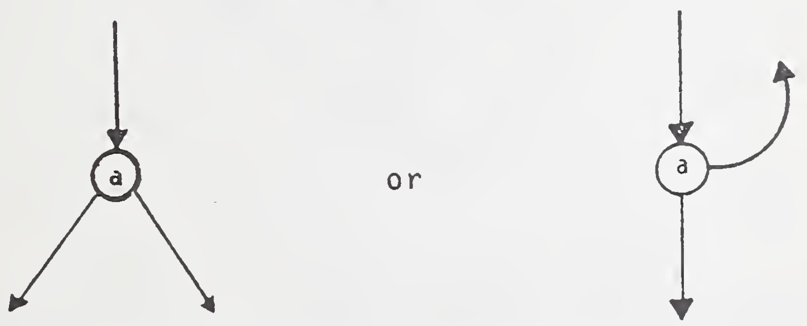


Figure IV-1 Splitting Node

In FORTRAN a splitting node would be associated with an IF, conditional GOTO, computed GOTO, or DO statement. If S is the number of splitting nodes in a graph, then v = S + 1. For example, in Figure IV-2 the splitting nodes are (a), (b), (c), and (d), so v=4+1.

Since each one of the splitting nodes in the graph is associated with some predicate or condition in the program, the expression v = S + 1 can be calculated by simply counting conditions in the source program. In fact, the number of conditions is a better complexity indicator than the number of predicates since a compound predicate can have more than one condition, e.g.:

```
IF c1 OR c2 THEN b1 ELSE b2
```

Since there are at least two ways the predicate can be true, the statement is modeled in Figure IV-3.

Notice that the complexity of the graph and the statement are both three. Notice, also, that the following statement is equivalent and also has complexity three.

```
IF c1 THEN b1
     ELSE IF c2 THEN b1
                ELSE b2
```

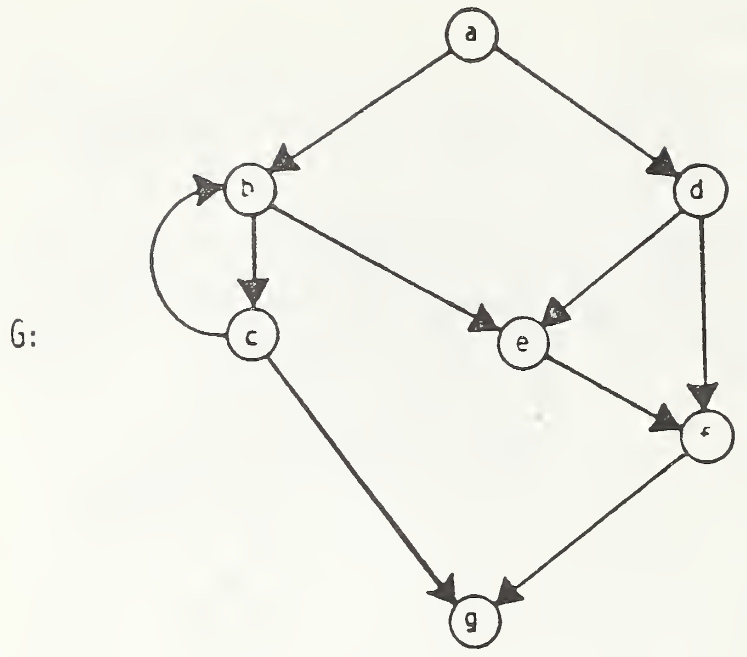


Figure IV-2 Control Graph G

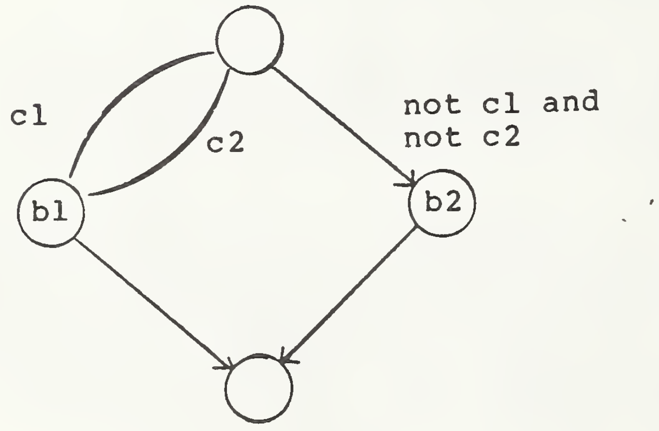


Figure IV-3 IF-THEN-ELSE Control Graph

If a program contains an n-way predicate, such as a CASE statement with n cases, the n-way predicate contributes n - 1 to the count of S. For example, in Figure IV-4 the CASE predicate (a) has three outcomes so it contributes two to S. This gives us v = 2 + 1. Notice that a CASE statement with n cases can be simulated with n - 1 nested IF-THEN-ELSE statements, which again produce the same complexity.

A second simplification allows the calculation of e - n + 2 by counting regions in the control graph. It uses Euler's formula:

> If G is a connected plane graph (a graph with no edges crossing) with n vertices, e edges, and r regions, then n - e + r = 2.

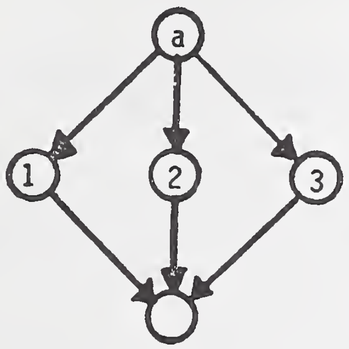


Figure IV-4 Case Control Graph

By changing the order of the terms, we get r = e - n + 2. So if the graph is planar, the calculation of complexity reduces to counting regions as illustrated in Figure IV-5.

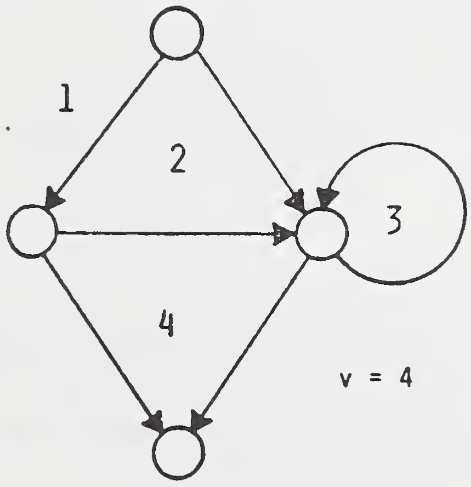


Figure IV-5 Plane graph

PART 2. TESTING METHODOLOGY

V. THE STRUCTURED TESTING CRITERIA

The criteria that must be satisfied to complete the structured testing technique for a program with complexity v is:

1. Every outcome of each decision must be executed at least once. and

2. At least v distinct paths must be executed.

It is important to understand that this is purely a criterion that measures the quality of the testing and not a procedure to identify test cases. In other words, the criteria above are a measure of the completeness of the testing that a programmer must satisfy. The criteria do not indicate how to arrive at the test data - we will discuss such a procedure in the Section VI.

For example, a program of complexity five has the property that no set of four test paths will suffice (even if there are, for example, 39 distinct tests that concentrate on only these four paths). The Theorem in Section II establishes that in the case where only four paths have been tested, there must be, independent of the programming language or the data within the program, at least one additional test path that can be executed. On the other hand, the Theorem in Section II also establishes for a program of complexity five, that a 6th, 7th, 8th ... path in a fundamental sense is redundant. That is, a combination of five basis paths will generate the additional paths.

Notice that most programs with a loop will have an arbitrarily high number of possible control paths, the testing of which is unrealizable. The power of the theorem in Section II is that it establishes a complexity number of v test paths that have two critical properties:

1. a test set of v paths can be realized (when this is violated, Section VIII will demonstrate that a program with lesser complexity will satisfy the same requirement).

2. testing beyond v independent paths is redundantly exercising linear combinations of basis paths.

Several studies have shown that the distribution of run time over the statements in the program has a peculiar shape. Typically, 50% of the run time within a program is concentrated within only 4% of the code [KNUT]. When the test data is derived from only a requirements point of view and is not sensitive to the internal structure of the program, it likewise will spend most of the run time testing a few statements over and over again. The testing criteria in this paper establishes a level of testing that is inherently related to the internal complexity of a program´s logic. One of the effects of this is to distribute the test data over a larger number of basis paths. Often the experience with the technique is that a lesser volume of testing is found to be more effective because it forces spreading the test data over more basis paths.

Operationally, the following experience with this technique has been observed. If a program´s complexity is small (range 1-5), then conventional testing techniques usually satisfy the structured testing criteria. However, as complexity increases, the experience is that conventional testing techniques will typically not execute a complete set of basis paths. Explicitly satisfying the structured testing criteria will in these cases yield a more rigorous set of test data.

The criteria is illustrated with an example. The FORTRAN program in Figure V-1 is designed to recognize strings of the form:

(i) A(B/C)*X

A string satisfying (i) has an ´A´ followed by zero or more occurrences of ´B´ or ´C´ followed by an ´X´. If the string satisfies (i), the program is supposed to return the parameter BOOL ´true´ and set the parameter COUNT to the total number of occurrences of ´B´ and ´C´. If the string does not satisfy (i), the program is to return the parameter BOOL as ´false´.

Notice that in the program, statements have been numbered to facilitate drawing the flowgraph. The style used in producing the graph is a matter of individual taste - the reader may have to try drawing a few graphs to be comfortable with this technique.

There are several techniques, such as numbering statements and highlighting labels that help. Many of the nodes in the graph in Figure V-1 have one entry and one exit and, therefore, can be deleted. Also, it helps to label edges coming out of a decision according to the conditions they represent. The flowgraph in Figure V-2 employs both these techniques and it is generally easier to work with than the original.

```
                         SUBROUTINE SEARCH(STRING,PTR,BOOL,COUNT)
                         INTEGER A,B,C,X
                         INTEGER STRING(80), COUNT, PTR
                         LOGICAL BOOL
                         DATA A,B,C,X/"101,"102,"103,"130/
                         BOOL = .FALSE.
                         BCOUNT = 0
                         COUNT = 0
1                        IF (STRING(PTR) .NE. A) GO TO 40
2             10         CONTINUE
3                        PTR = PTR + 1
4                        IF (STRING(PTR) .NE. B) GO TO 20
5                        BCOUNT = BCOUNT + 1
6                        GO TO 10
7             20         CONTINUE
8                        IF (STRING(PTR) .NE. C) GO TO 30
9                        COUNT = COUNT + BCOUNT + 1
10                       BCOUNT = 0
11                       GO TO 10
12            30         CONTINUE
13, 14                   IF (STRING(PTR) .EQ. X) BOOL = .TRUE.
15            40         RETURN
16                       END
```

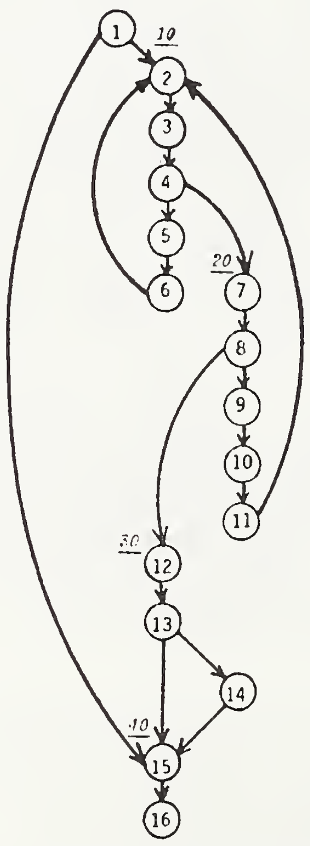


Figure V-1 FORTRAN Example Graph

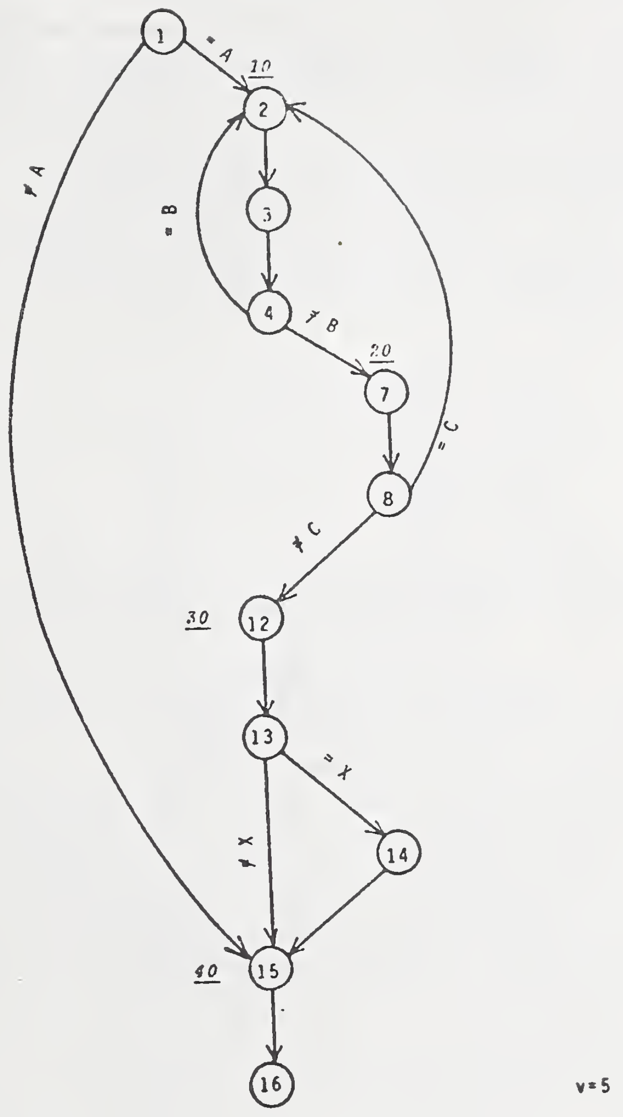


Figure V-2 Flow Graph

The error in the program in Figure V-1 is that BCOUNT should be added to COUNT after recognizing a B. We will examine some testing schemes and see how effective they are in detecting this bug.

A methodology often used [MILL] requires that

1) all statements must be executed

2) each decision must be executed both ways

A typical set of test data that fulfills these criteria is:

```
t1:  #     (# denotes any character except A)
t2:  ABC^  (^ denotes any character except X)
t3:  ABCX
```

The program SEARCH executes correctly in each test t1 through t3 so the test data fails to detect the error.

Applying the testing criterion discussed in Section V, we need at least five distinct paths that cover all the edges in order to test SEARCH. The following set of five will do.

```
b1:  ABCX
b2:  #
b3:  ACX
b4:  ABX
b5:  ABC^
```

Test b4 results in BOOL = ´true´ and COUNT = 0, so it shows that the program does not meet the specification.

A few comments about practical experience using this technique may be in order. One misconception the reader may get is that it is necessary to have an automated system in order to use this method. Although an automated system can help, particularly in seeing the control graphs on a CRT, the main application of this technique has been by hand. Control graphs are drawn by hand, and the graph, the complexity v, and the data for the v distinct paths that are tested are all included as part of the standard documentation. Experience has shown that having the graphs and the actual test data proves invaluable later in modification and maintenance phases because the programmers know exactly which cases worked previously and they do not have to guess or take it on faith. Part III of this paper elaborates on the use of the test data in maintenance.

Often more than one test is performed on a path. The validation process for a particular path often consists of more than just exercising it once; all the functional requirements the path implements should be tested. Also, the programmer should explicitly look for data values that could produce errors along the path. This process may result in a large number of distinct test cases; however, it is critical that within this set of test cases, there are v distinct paths that cover every edge.

VI. IDENTIFYING TEST PATHS: THE BASELINE METHOD

The technique described here gives a specific methodology to identify a set of control paths and test data to satisfy the structured testing criteria. The technique, when applied, results in a set of test data and control paths equal in number to the cyclomatic complexity of a program. The technique is currently called the baseline method; it strengthens the structured testing method because it gives a specific technique to identify actual test data and test paths.

THE METHOD

The first step is to pick a functional "baseline" path through the program which represents a legitimate function and not just an error exit. The selection of this first baseline path is somewhat arbitrary. The key, however, is to pick a representative function provided in the program as opposed to an error path that results in an error message or recovery procedure. To test the baseline, exercise all the functional requirements implemented on the baseline. Also look for data that would produce errors on the baseline.

It is to be realized that this functional baseline path represents a sequence of decisions taken in a particular way.

The second step is to identify the second path by locating the first decision in the baseline path and flipping its result while simultaneously holding the maximum number of the original baseline decisions the same as on the baseline path. This is likely to produce a second path which is minimally different from the baseline path. Apply the same testing process described above to this path.

The third step is to set back the first decision to the value it had on the baseline path and identify and flip the second decision in the baseline path while holding all other decisions to their baseline values. This, likewise, should produce a third path which is minimally different than the baseline path. Test this path.

This procedure continues until one has gone through every decision and has flipped it from the baseline value while holding the other decisions to their original baseline values.

Since $v = S + 1$ if, for example, $v=7$, there are 6 such decisions which one flips resulting in 6 paths that differ from the baseline path; all of which adds up to 7 distinct paths.

Since the selection of the baseline path is somewhat arbitrary, there is not necessarily "the" right set of test data for a program. That is, there may be several sets of test data that satisfy the structured testing criteria. The application of this baseline method will, however, generate a set of test data with the right properties:

- o v distinct independent paths will be generated
- o every edge in the program graph will be traversed.

An Example:

The graph G, Figure IV-1, discussed in Section II is used to illustrate the method. The reader will have to trace through G to follow the discussion.

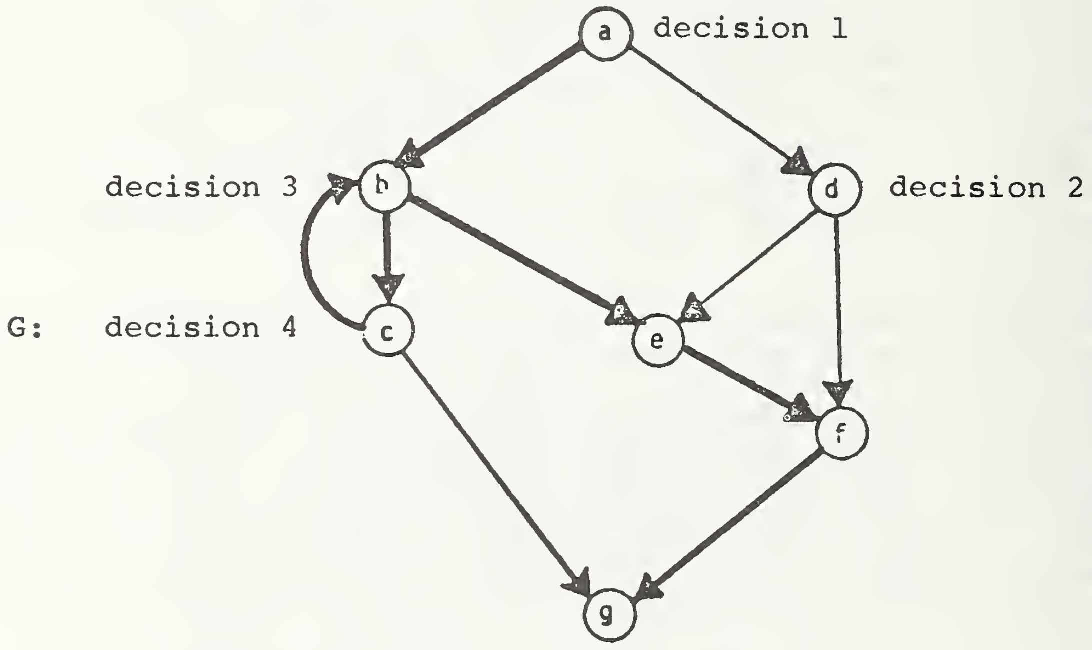


Figure VI-1

STEP 1: Choose a Baseline path, path 1. As shown in G by dark lines, the path A-B-C-B-E-F-G is chosen as the baseline. It is assumed that this path represents one of the main functions in the program as opposed to an error path. This initial choice is somewhat arbitrary; keep in mind that the baseline path ideally performs the major full function provided in the program. Try to pick a baseline path that intersects a maximal number of decisions in the graph.

STEP 2: The first decision to be flipped is A. Path 2 should be chosen to differ minimally from the baseline - this yields the path A-D-E-F-G to be tested.

STEP 3: Now flip decision D in path 2 yielding the third path: A-D-F-G.

STEP 4: Since A has been flipped from the baseline, the next decision to flip is B. This results in the path 4, A-B-E-F-G.

STEP 5: The nodes E, F, and G in the baseline path A-B-C-D-E-F-G are not decisions. Since A and B have been flipped, the only decision remaining to reverse is C. This yields the fifth and last path: A-B-C-G.

Since we have flipped every decision once, this completes the procedure. Notice that the baseline procedure satisfied the structured testing criteria. The complexity of G is 5 (v(G) = 11 - 7 + 1); we have generated 5 independent paths that have traversed each edge.

Refer back to the sample FORTRAN program in Section V, Figure V-1, and its graph which is repeated as Figure VI-2. It is interesting to apply the baseline method to this program.

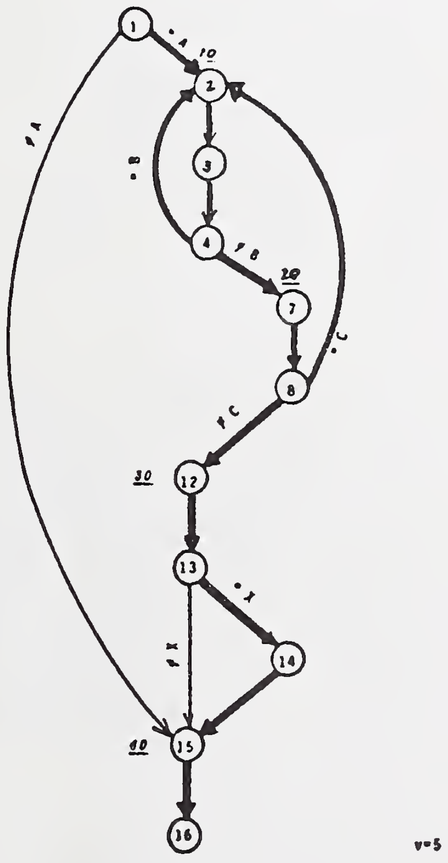


Figure VI-2 Flow Graph

Assume the path 1-2-3-4-2-3-4-7-8-2-3-4-7-8-12-13-14-15-16, as shown by dark lines, representing the test data ABCX, is chosen as the baseline. The baseline procedure then yields paths with the following test data.

b1: ABCX

b2: # (# denotes any charcter except A)

b3: ACX

b4: ABX

b5: ABC^ (^ denotes any charcter except x)

This is the same test data presented in Section V - here we have shown how to derive it.

VII. INTEGRATION TESTING

In Section II, the notion of complexity was derived from the cyclomatic number of a graph. The discussion, however, was limited to graphs with only one component. In this section we will generalize the approach to cover graphs that have several components. The application will be to measure design complexity; specifically, we will quantify the effort required to perform integration testing of several modules within a design structure.

Our focus up to this point has been on one module at a time and the testing application has been at the unit level. A module will typically be represented in a FORTRAN, PL1, or PASCAL program as a procedure, function, or the main line code. In COBOL, a module is typically expressed as a paragraph which is referenced from several distinct places within the program.

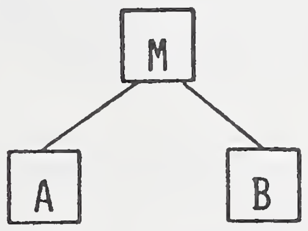


Figure VII-1 Design Hierarchy

Figure VII-1 is a standard representation of a design where M is the top level and calls module A and module B. The design in Figure VII-1 implies the following: M, A, and B are all distinct modules. They have their own internal specifications and will have their own unique test data. Modules A and B are called from M. They, however, may also be called in a different context by other modules and could be on a program library. Notice that this is quite different from a situation where A´s code and B´s code would be embedded within M.

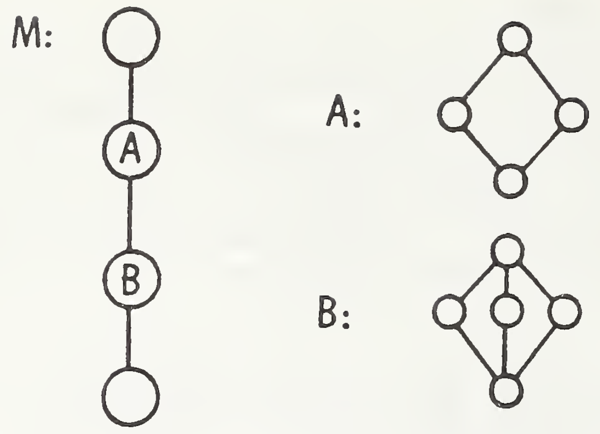


Figure VII-2 Graph of Design Hierarchy

Figure VII-2 is a graph which shows the algorithm we might find "inside" modules M, A, and B. The graph in Figure VII-2 has three components, M, A, and B - each of the graphs we have previously discussed had only one component.

We have to add an extra edge to each component in a graph to satisfy the strong connectivity condition of Theorem 1. Therefore, the more general expression is $v = e - n + 2p$; this expresses the system complexity of a design with several component graphs as opposed to the more specific $v = e - n + 2$, which applies to a single component.

When the number of components is 3 ($P=3$), the complexity calculation yields $v = 13 - 13 + 2 * 3$. This design complexity of 6 represents the testing effort required to perform a top down integration of the three modules M, A, and B. For example, using a top down integration strategy, the following tests occur:

- <u>One</u> test is required to verify the code within M. Stubs that simulate the actions of A & B are called to allow this testing of M.

- <u>Two</u> tests are required to verify A´s logic. Each of these calls on A are driven through M in order to invoke A. During A´s testing, the stub for B is still in place.

- <u>Three</u> tests are likewise required to verify B´s logic. As above, each of the three calls on B would be driven "top down" through M.

As the complexity quantification indicated, it is indeed the case as shown above that six tests are required to integrate the three modules.

Notice that the design complexity in a graph with several components is equal to the summation of the unit level complexity. With the example above, the complexity can be computed as $v = v(M) + v(A) + v(B) = 1 + 2 + 3$. For a formal proof of this, see [MCCA].

The application of the design level complexity is different in nature than the application of unit level complexity. Design level complexity is not limited in the sense that the unit level complexity is controlled. The main application of the design complexity is to quantify the integration effort of a collection of the modules.

There are occasions, however, when the design level complexity can be used to make a comparison of the relative complexity of subsystems within an overall design. This quantitative view of the subsystems complexity will give a more stable predictor of several project attributes than the more customarily used lines of code. For example, if the design level complexity of one subsystem is 2000, and a second subsystem complexity is 30, there are several implications - for example, the subsystem testing and integration are more closely correlated with this design complexity quantification than the subsystems´ physical size in terms of lines of code.

VIII. THE REDUCTION TECHNIQUE

When this methodology is applied to an ongoing project or when an existing testing practice is analyzed, the usual outcome is that the actual number of paths tested is less than the cyclomatic complexity. The concept behind this methodology is to quantify and to limit the complexity of a program and then to require the testing to be at least as thorough as the quantification.

The idea in this section is that if the actual testing does not meet the cyclomatic complexity, then either the testing can be improved or the program logic can be simplified.

Let us assume that a program has been written. Its complexity v has been calculated. The number of distinct paths traversed during the test phase is ac (actual complexity). If ac is less than v, one of the following conditions holds:

1. The program contains additional paths that can be tested.

2. The complexity of v can be reduced by v - ac (v - ac decisions can be removed from the program).

3. Portions of the program can be reduced to in-line code (constant length loops have been used in order to conserve space).

The actual paths that are tested in a program are determined by the data flow and data conditions at the various decisions. Because of the data flow, a number of paths may not be realizable in a given program. The point of this section is that when the data flow and data conditions are considered, there must be at least v distinct paths or else the program can indeed be reduced.

Several examples are shown to illustrate case 2, the reduction technique; it should be noted that, in practice, the most frequent outcome is case 1 and the existing testing can, in fact, be improved.

Consider the following program:

```
J = 1;
IF I ≤ 3 THEN I = 2
         ELSE J = 14;
IF (I + J) ≤ 6 THEN OUTPUT (I, J);
```

The complexity is 3 and the control graph is shown in Figure VIII-1.

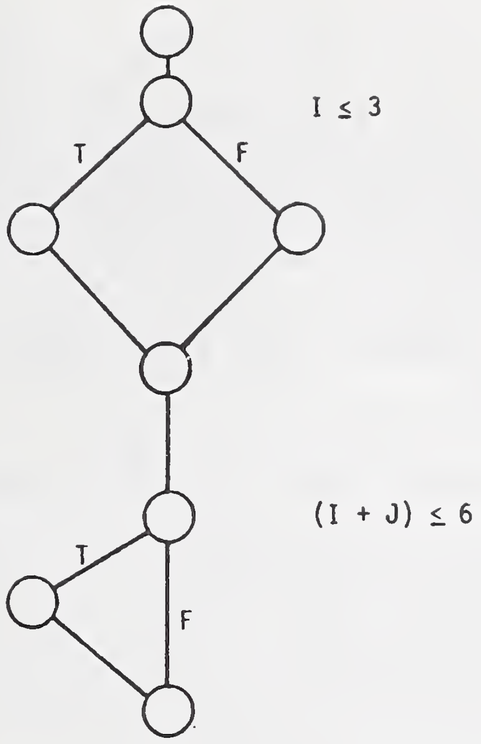


Figure VIII-1 Program Control Graph

It is clear that ac = 2 since the only realizable paths on the graph are TT, FF. That is, one path where $I \leq 3$ is true and $(I + J) \leq 6$ is also true, and a second path where both conditions are false.

Since there are no additional paths to test, and there are no constant length loops, the program can be reduced to complexity 2:

```
J = 1
IF I ≤ 3 THEN
   BEGIN
    I = 2
    OUTPUT(I,J);
   END
ELSE
   J = 14;
```

As a second example, we will use the FORTRAN program in Section V. Let us assume that the tests used are:

```
#   (# denotes any character except A)
A^  (^ denotes any character except X)
ABCBCX
```

Recall that these tests satisfy the C2 testing criterion that each decision outcome is executed at least once, but we have ac = 3 whereas v = 5. Now, if we believe that the tester really cannot find any additional tests, then the program can be reduced to complexity 3. In fact, if the tester insists that no more paths exist, a programmer must admit that the program in Figure VII-2 containing these three paths will suffice.

```
      SUBROUTINE SEARCH (STRING, PTR, BOOL, COUNT)
      INTEGER A, B, C, X
      INTEGER STRING (80), COUNT, PTR
      LOGICAL BOOL
      DATA A, B, C, X /"101", "102", "103", "130"/
      BOOL = .FALSE.
      COUNT = 0
      IF (STRING(PTR) .NE. A) GO TO 40
      PTR = PTR +1
      IF (STRING(PTR) .NE. B) GO TO 40
      COUNT = 4
      BOOL = .TRUE.
 40   RETURN
      END
```

Figure VIII-2 FORTRAN Program

The point here is not that the program in Figure VIII-2 is the desired one, but rather that the testing process can, and should, be improved.

A frequent error in testing strategy is to test only the expected data and overlook testing the error conditions. The third example illustrates this with the program SEARCH, in a case where test data checks only the expected conditions.

```
ABCBBCBBX
AX
ACCX
```

Once again, if the programmer claims that these are the only paths, the program can be reduced to the following complexity 3 code:

```
      SUBROUTINE SEARCH (STRING, PTR, BOOL, COUNT)
      INTEGER A, B, C, X
      INTEGER STRING (80), COUNT, PTR
      LOGICAL BOOL
      DATA A, B, C, X /"101", "102", "103", "130"/
      BOOL = .TRUE.
      COUNT = -1
   20 PTR = PTR + 1
      COUNT = COUNT + 1
      IF ((STRING(PTR) .EQ. B) OR (STRING (PTR) .EQ. C)) GO TO 20
      RETURN
      END
```

Figure VIII-3 FORTRAN Program Revised

The one case where it may in fact be impossible to find v distinct paths is the situation where the programmer increases complexity to conserve space. For example, if the program contains a fixed-length loop, where the increment and limits are constant and are not modified by the body, then a loop that iterates n times is equivalent to n in-line copies of the body. For instance, the following code:

```
    DO 10 I = 1,3
10  A(I) = I
```

is equivalent to

```
    A(1) = 1
    A(2) = 2
    A(3) = 3
```

which has complexity 1.

In summary, the cyclomatic complexity v of a program can be thought of as specification for testing the paths. If a given program does not have at least v distinct tested paths, then either the testing is incomplete or there is excessive logic that can be removed. The guideline is: if there is any logic that is untestable, then that logic is removable.

For large systems and certain applications, it is recognized that this objective may be very difficult and not practical to attain. Some cases in which reduction may not be possible are:

- defensive programming
- hardware mistrust
- fault tolerant programming

Nonetheless, every effort should be made to accomplish this goal.

Part 3. Maintenance Methodology

IX. ESSENTIAL COMPLEXITY

An interesting question associated with a program's complexity is quantifying how well-structured a program is. That is, how do we quantify the degree to which a program has been written using only the standard structured control flow constructs in Figure IX-1.

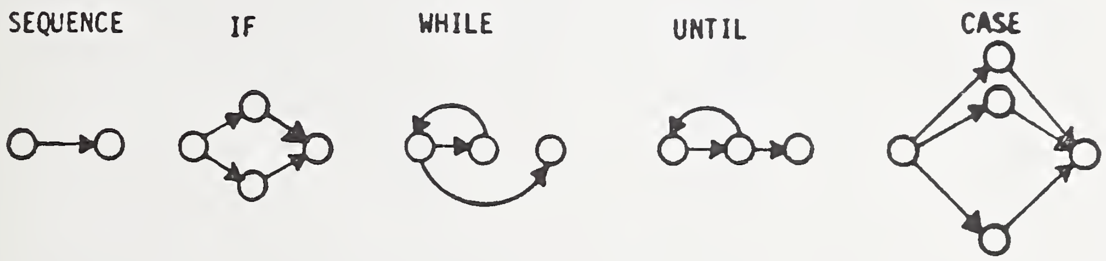


Figure IX-1 Structured Control Constructs

This is an important concern since one of the basic ways to reduce the complexity of a program where v exceeds 10 is to recognize and to remove subfunctions from the main control flow so they become separate subroutines or functions. It turns out that if a program is structured (i.e., it uses only the constructs SEQUENCE, UNTIL, WHILE, IF, CASE), its complexity can be reduced in a straightforward manner. For example, in the graphs in Figures IX-2 and IX-3, the original structured program can be reduced to a program of complexity 1 by making the one-entry-one-exit subgraphs into functions.

The reduction process, more formally, is the process of replacing proper subgraphs with single-entry and exit nodes. Essential complexity is defined below in order to reflect how well a program is structured.

Let G' denote the reduced graph that results from removing all proper, single-entry, single-exit subgraphs. Also, leading edges and trailing edges should be removed so the first node is a decision node and the last node is a collection node. The essential complexity of a graph G is defined as ev = v(G').

Although G1 through G3 has v = 5,Figures IX-3 through IX-5, each of the subgraphs of G1 could be removed whereas G3 cannot be reduced at all. If the graphs were highly complex, this would be crucial since G1 could be reduced into subroutines, each with complexity less that 10, but G3 could not be reduced. Therefore, a further modularization would require that the G3 program be thrown out and a new program be designed.

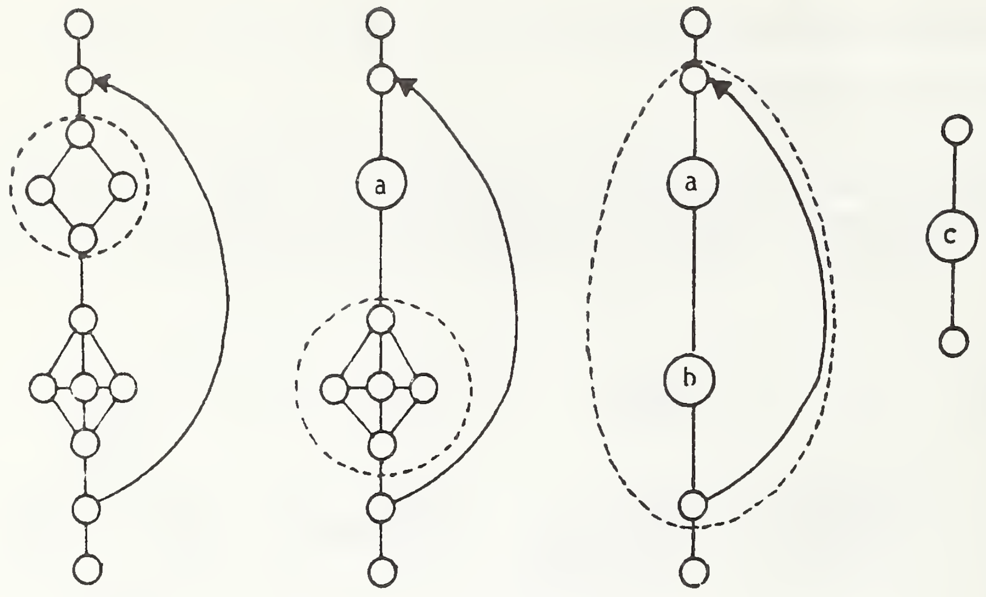


Figure IX-2 Reducing Complexity, Example 1

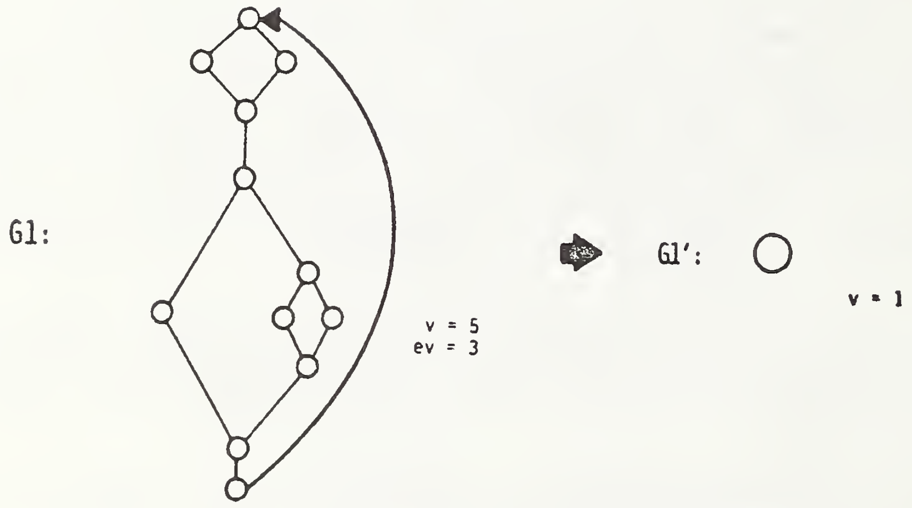


Figure IX-3 Reducing Complexity, Example 2

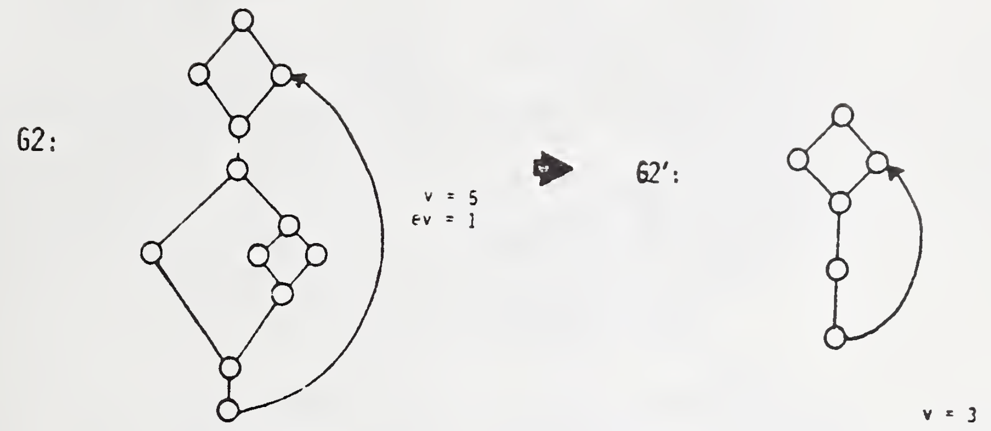


Figure IX-4 Reducing Complexity, Example 3

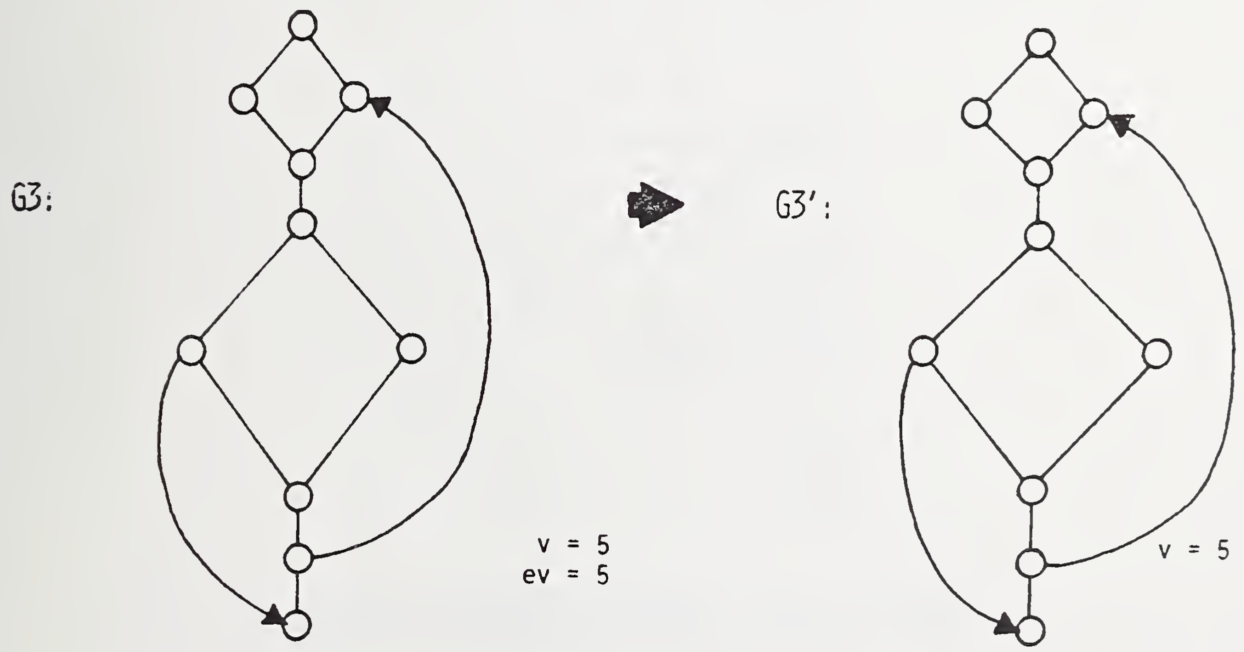
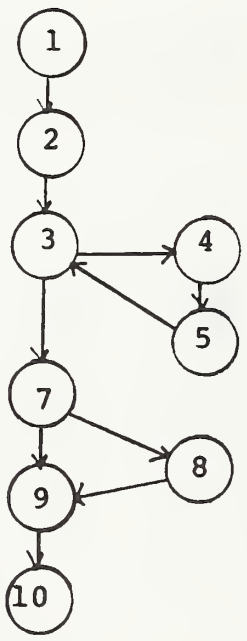


Figure IX-5 Non-Reducible Control Graph

An example of reducing complexity by adhering to standard structured control flow constructs is Figure IX-6. This is a rewrite of Figure V-1, the string recognition problem. The complexity of the original program is 6 while the complexity of the rewrite is 4. The essential complexity of the original program is 3 while the essential complexity of the rewrite is 2.

```
                  SUBROUTINE SEARCH(STRING,PTR,BOOL,COUNT)
                  INTEGER A,B,C,X
                  INTEGER STRING(80), COUNT, PTR
                  LOGICAL BOOL
                  DATA A,B,C,X/"101","102","103","130/
                  COUNT=0
                  BOOL=.FALSE.
1                 IF STR(PTR) .EQ. A
2                    THEN PTR = PTR+1
3                    WHILE STR(PTR) .EQ. B .OR. STR(PTR) .EQ. C DO
4                       COUNT=COUNT+1
5                       PTR=PTR+1
6                    END
7                    IF (STR(PTR) .EQ. X)) BOOL=.TRUE.
9                 ENDIF
10           40   RETURN
```



v=4

Figure IX-6 FORTRAN Example Rewriten

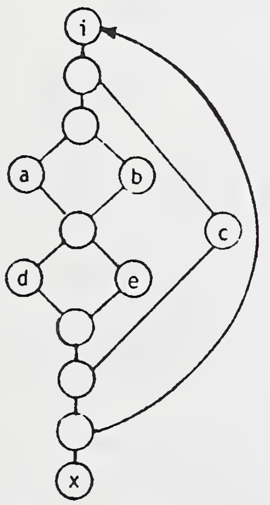
X. PROGRAM MODIFICATION

X.1 The Problem

Several studies have indicated that software maintenance and modification often takes as much as 70% of the total life-cycle cost. Much of this maintenance activity involves the modification and retesting of existing programs, for which very little methodology exists. This section introduces procedures for performing modifications and their tests in a more orderly manner.

X.2 Modifying Functional Statements

If a patch or modification to a program does not change the control flow structure, the change is typically confined to a functional node (a node with not more than one output edge). In programming terms, this type of change involves modifying functional statements such as: input, output, and statements that perform calculations. In contrast to this, a control flow change involves the modification or insertions of statements such as GOTO´s, IF´s, and DO-LOOPS which affect program control.



v = 5

Tests:

1. icx
2. iaecx
3. iadbecx
4. ibdbecx
5. iadbdcx

If, for example, node e is changed, then the minimal amount of retesting is the subset of test paths that contain e:

2. iaecx
3. iadbecx
4. ibdbecx

Figure X-1 Program Modification

The method of verifying a functional statement change follows:

1. identify all structured test paths that contain the changed node - these test paths should be contained in the Unit Development Folder or other suitable documentation

2. re-execute all such paths that contain the changed node

The example in Figure X-1 illustrates the procedure.

Notice that the retention of such unit test data forms a local test bed that can be used to regressively establish that the change does not destroy the original functions provided.

X.3 Modifying Control Flow

X.3.1 Example of Catastrophic Change

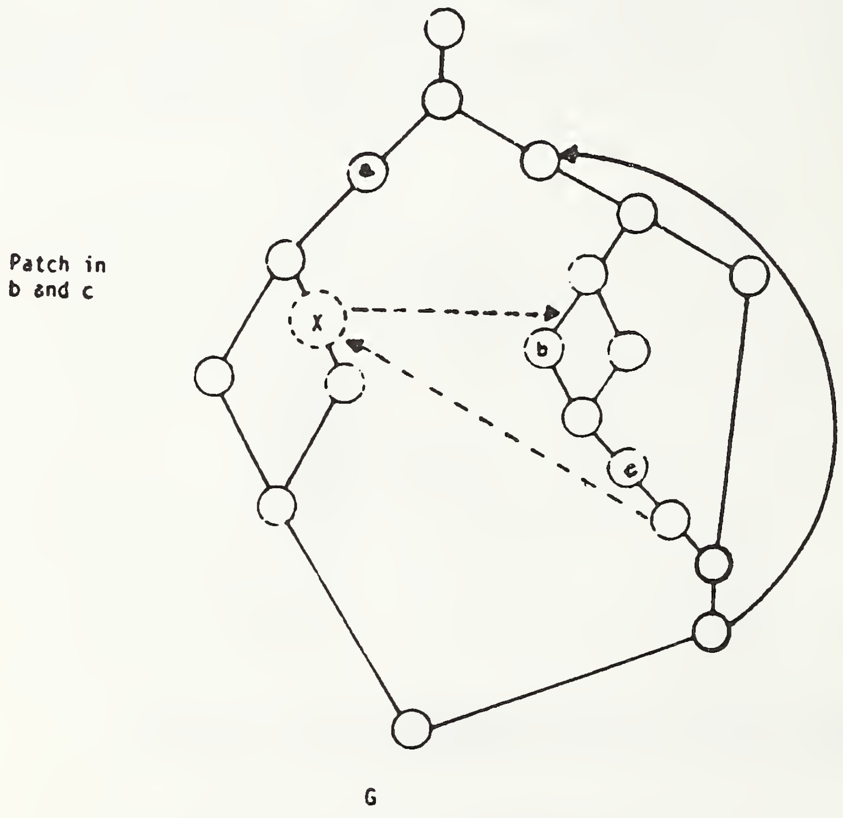


Figure X-2 Program Patch

Assume the program in Figure X-2 is being modified. Assume also that at the node X the programmer wants to have the code (b) and (c) execute.

A common way to achieve this is to patch in the conditional GOTO´s shown as dotted lines that branch to a point before (b) and then return after node C. This may seem innocent and, in fact, desirable, since the size of the blocks (b) and (c) could be large and the programmer may be enthusiastic about the space being saved. The point usually missed, however, is the structural change in the program. The two patches only had a modest effect on the cyclomatic complexity, which changed from 6 to 8. And, in general, cyclomatic complexity changes slowly with patches to control nodes (v goes up 1 per conditional GOTO, and v is insensitive to the deletion or addition of functional nodes like (a)).

The essential complexity has, on the other hand, changed substantially with the patches: from 1 in the original to 8. The original program could be decomposed into several independent functions whereas the modified program could not, and the functions that were independent are now coupled. The main point here is that our common maintenance practices often have the effect of changing a well-structured program into a completely unstructured program. This can happen by changing only one source statement without being aware of the dramatic structural change.

X.3.2. Re-Test Methodology

The next two sections give the operational steps in performing maintenance testing. The first step identifies changes that are virtually untestable; it precludes such changes from being introduced. The second step actually quantifies the number of tests to be run given a structural change.

X.3.2.1. Evaluate Essential Complexity

The previous example in Section X.3.1 illustrates the first step of the maintenance discipline:

> "Evaluate the effect of a control flow change on <u>essential complexity</u>; do not allow a well-structured program to severely degrade."

X.3.2.2. Re-test Quantification

This leads to the central procedure for maintenance testing. This procedure quantifies the number of tests required to validate such a change.

Figure X-3 depicts a situation where Y is the name of the code being branched into and X is the code being branched from.

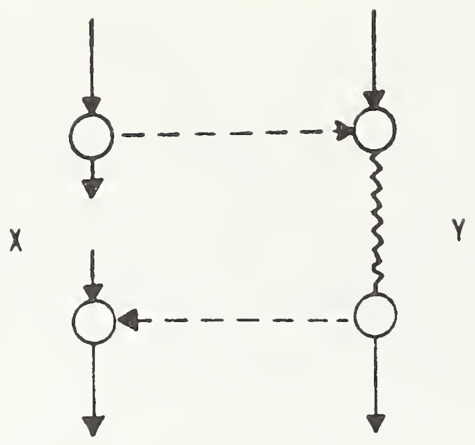


Figure X-3 Branching Graph

The three particular cases arising in practice are shown in Figure X-4.

In cases where Y is a subgraph with unique entry and exit nodes, we can compute the cyclomatic complexity v(Y). In such cases, the number of paths to verify the patch is 2*v(Y) + 1. Cases 1) and 2) satisfy this since code Y that was branched into has a single entry and exit.

For example, in Case 2, v(Y) = 3, so 7 tests are required to test the patch. Three paths should be tested through the normal entry of Y to demonstrate that the branch back into X is not taken. Three more paths that take the new branch into Y, traverse Y in three different ways and then return to X. And finally, one last test has to be made of the path that does not take the new path but falls into X instead.

In case 3) the block Y does not have a single entry and exit, so the expression 2*v(Y) + 1 does not apply. This type of modification should be avoided since it will have a disasterous effect on the program´s structure; the evaluation of essential complexity cited in X.3.2.1 would have precluded such a change.

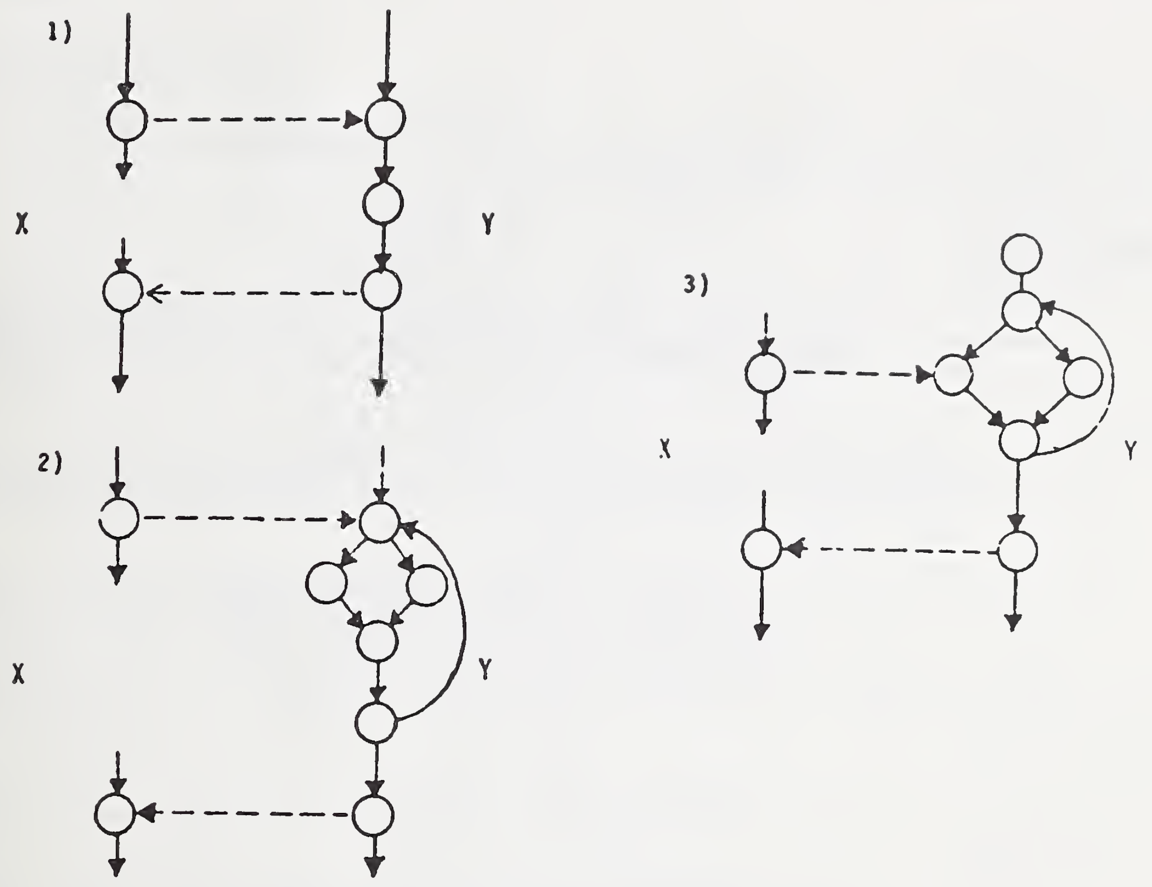


Figure X-4 Branching Graph Amplified

XI. SUMMARY

In this section the operational steps of structured testing are consolidated and listed below.

DESIGN STAGE

If the algorithm is written in a high level program design language, limit complexity to seven. Experience has shown that when the coding takes place, complexity will approach 10.

If the internal specifications of software modules include the number of conditions that must be tested internally, limit such conditions to six.

If the above information is not available at the design phase, break modules you intuitively feel will exceed complexity 10 into submodules with complexity less than or equal to 10.

CODING PHASE

Make an explicit flow graph organic to the programming process.

Calculate the cyclomatic complexity v with any of the three methods described in Section IV.

When complexity exceeds 10 go back to the design phase and refine the logic into modules, each with complexity 10 or less (the exceptions are the CASE statement or project specific contraints).

UNIT TESTING PHASE

Use the baseline method to identify test paths and data until the number of such paths satisfies the following criteria:

v independent paths are represented.
Every edge in the flowgraph is covered at least one time.

If the above criteria are not satisfied, then either:

More test paths exist that can be exercised or
The program contains redundant logic that can be removed.

Keep documentation on the paths tested available for the maintenance phase; typically in a unit development folder.

MAINTENANCE PHASE

Classify a proposed change to the code as a functional statement change or a control statement change.

In the case of functional statement change, regressively retest all original test paths in the unit development folder that intersect with changed functional statements.

In the case of a control statement change:

If the essential complexity will substantially increase, do not make the change; the program will become unmaintainable. Take a different approach to the modification.

Where the essential complexity will not increase, quantify and retest by the 2*v(T)+1 rule.

Figure XI-1 illustrates the main steps in the structured testing technique.

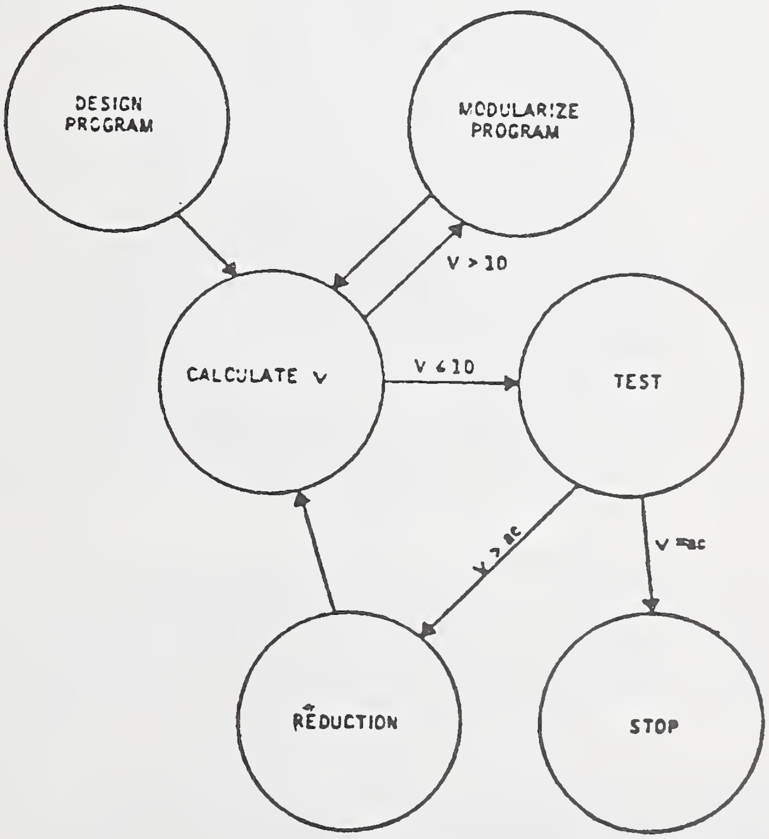


Figure XI-1 Structured Testing Technique

APPENDIX A

Empirical Evidence

We have been introducing this subject by graph theory, example, and intuition. Independent empirical data that validates this approach in the real world would be useful. In software, empirical evidence typically takes years to collect; data on the use of v, the cyclomatic complexity, is at this time sparce. However, the results are encouraging.

In a series of controlled experiments conducted at General Electric [CURTa] and [CURTb], v was found to predict the performance of programmers on comprehension, modification, and debugging tasks. For example, on the debugging task, programmers were asked to locate and correct a single error in each of three programs. A statistically significant correlation was found between the complexity of the programs, as measured by v, and the time required to locate and correct the bugs. V, in fact, was a considerably better predictor of debugging time than was the number of lines of code.

A comparison of the programs [BASI] produced by disciplined teams, conventional teams, and individuals was done in 1979. The participants in this experiment consisted of advanced undergraduate and graduate students at the University of Maryland. Their task was to design and implement a relatively simple compiler. The entire project required approximately two staff-months of effort and resulted in systems averaging about 1200 lines of code. The disciplined teams were required to use a set of state-of-the-art techniques such as top down design, walkthroughs, and chief programmer team organization. The conventional teams and individuals were given no such requirement and, in fact, had received no formal training in these techniques. The software produced by the disciplined teams was completed with less effort and with fewer errors than that produced by the conventional teams and individuals; the program modules were less complex as measured by v. Thus, the disciplined methodology in this study led to more reliable, less complex software.

Henry, Kafura, and Harris [HENR] reported empirical error data collected on the UNIX operating system. The authors obtained a list of errors from the UNIX User´s Group and performed correlations on three metrics. The cyclomatic complexity v was the most closely related to errors of the three - the correlation between v and number of errors was .96.

Walsh [WALS] 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 of less than 10 and half a v of 10 or greater. The average error rate for modules with a complexity of less than 10 was 4.6 per 100 source statements while the corresponding error rate for the more complex modules was 5.6. As Walsh pointed out, one would expect a similar pattern for undetected errors as well. It would be expected that fewer errors should appear for the less complex modules during the maintenance phase.

Myers [MYER] calculated v for the programs contained in the classic text by Kernigan and Plauger [KERN]. For every case in which an improved program was suggested, this improvement resulted in a lower value of v. Myers describes one interesting case in which Kernigan and Plauger suggested two simplified versions of a program which has a v of 16. The two improvements were done and Myers found that both had a v of 10.

In a recently-completed study [SHEP], the performance of programmers in constructing programs from various specification formats was examined. An automated data collection system recorded the complete sequence of events involved in constructing and debugging each program. An analysis of the error data revealed that the major source of difficulty was related to the control flow and not to such factors as the sheer number of statements or variables. The most difficult program had the most complex decision structure while a considerably easier program performed extremely complex arithmetic calculations but had a simpler decision structure. Thus, v can be used to measure a truly difficult aspect of programming. A similar result was also reported by Curtis, Sheppard, and Milliman [CURTa].

Not only does v have a solid foundation in mathematics, but the studies cited illustrate that it predicts the difficulty experienced by programmers in working with software, the number of errors detected in program modules, and it conforms to subjective judgments of complexity.

APPENDIX B

A FORTRAN Example

An example is given to illustrate the structured testing technique. It is virtually impossible to choose an example that would interest everybody. It seems that the best choice is a program that plays a game with which hopefully, many people are familiar, Blackjack. We will give an informal specification of the game, show a design of a Blackjack playing system, and finally concentrate on one of the modules and apply the structured testing process to it.

The rules of Blackjack are somewhat parochial. Even if you are an expert, read the following specification because the rules may differ from your experience.

BLACKJACK

The program, as the dealer, deals two cards to itself and two cards to the player. The player´s two cards are shown face up, while only one of the dealer´s cards is shown. Both the dealer and the player may draw additional cards, a hit. The player´s goal is to reach 21 or less, but be closer to 21 than the dealer´s hand - in which case, the player wins. Ties go to the dealer. If the player´s or the dealer´s hand totals greater than 21, the hand is busted. The King, Queen and the Jack all count as 10 points. All other cards, except the Ace, count according to their face value. The Ace counts as 11 unless this causes the hand to be over 21. In this case, the Ace counts as one.

If both the dealer and the player get Blackjack, which is a two-card hand totaling 21, neither wins; it is a push. Blackjack beats all other hands - if the player has Blackjack, he wins "automatically" before the dealer has a chance to take a hit. The Player can also win automatically by having five cards without busting. The player may take any number of hits - as long as the hand is not a bust. The dealer must hit while the hand is less than or equal to 16; at 17, the dealer must stand.

The program checks the player for Blackjack; ´hits´ the player; checks the dealer for Blackjack; ´hits´ the dealer. The protocol to receive a card is a hit and to stop where you are is a stand. The program periodically shuffles and asks for a cut of the deck. When queried by the program, type a "1" if the answer is yes, and "0" if the answer is no.

DESIGN

The design hierarchy for the Blackjack system we will be using is shown in the Figure B-1. The top module, BLACKJACK, first calls subroutine SETUP that initializes the deck. It then calls a module MIX which shuffles the deck; MIX asks the player to cut the deck . The third module called is HAND which contains the logic for a one-hand session of Blackjack. HAND, in turn, calls a subroutine, HIT, which determines the next card in the deck and handles the Ace which can have two values. A typical use of the system is that Blackjack is called and calls SETUP one time, calls MIX for an initial shuffle; then calls HAND for a Blackjack session. At the end of one Blackjack hand, the player has the option to play again. If so, several calls on HAND are made as shown by the inner loop on the hierarchy chart figure. The outer loop represents MIX being periodically called when the end of the deck is approaching. Our interest in this example is the module HAND. The code for the other modules is included at the end of this section for completeness but need not be focused on for this exercise. We will assume the other modules are working correctly in this exercise and focus on the testing of the logic within HAND.

The specification for Blackjack that we use follows:

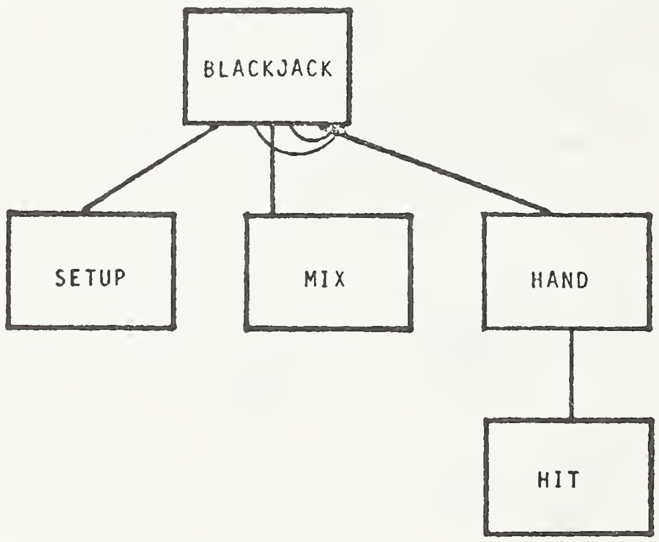


Figure B-1 Blackjack Specification

<u>SUBROUTINE "HAND"</u>

We will introduce the logic within HAND in pieces in order to make it understandable.

At several points within HAND, there are calls on subroutine HIT - the code follows.

```
175.      0 C**
176.      0 C**
177.      0            SUBROUTINE HIT(TOTAL,ACES)
178.      0            INTEGER TOTAL,ACES
179.      0            INTEGER I,CARDS(52),DEBUG
180.      0            COMMON /DECK/CARDS,I,DEBUG
181.      0
182.      0            IF ( DEBUG .EQ. 1 ) THEN
183.      9                WRITE(*,'(A$)')'                NEXT CARD?'
184.     62                READ(*,'(BN,I2)') CARDS(I)
185.    109            ENDIF
186.    109 C**
187.    109            TOTAL=TOTAL+CARDS(I)
188.    129            IF (CARDS(I) .EQ. 11)  THEN
189.    148                ACES=ACES+1
190.    154            ENDIF
191.    154            I=I+1
192.    167            IF ((TOTAL .GT. 21) .AND. (ACES .GE. 1)) THEN
193.    178               TOTAL=TOTAL-10
194.    184               ACES=ACES-1
195.    190            ENDIF
196.    190 C**
```

The parameters passed to HIT are, the player's or dealer's total and number of aces. It adds the next card to the total (line 187). If the total exceeds 21, the program changes the value of aces on lines 192 through 196.

The first section of subroutine HAND contains several declarations and initialization.

```
 88.      0 C**
 89.      0 C**
 90.      0 C**
 91.      0            SUBROUTINE HAND(WIN)
 92.      0 C**
 93.      0            INTEGER P,D,PACE,DACE,I,CARDS(52),DEBUG,COUNT,WIN
 94.      0            COMMON /DECK/CARDS,I,DEBUG
 95.      0 C**
 96.      0            P=0
 97.      3            D=0
 98.      6            PACE=0
 99.      9            DACE=0
100.     12            WIN=0
101.     15 C**      WIN WILL BE 0 IF DEALER WINS,1 IF PLAYER WINS,2 IF A PUSH
102.     15            CALL HIT(P,PACE)
103.     21            CALL HIT(D,DACE)
104.     27            CALL HIT(P,PACE)
105.     33            CALL HIT(D,DACE)
106.     39            COUNT=0
107.     42 C**
108.     42            WRITE(*,'(A,I2)')'DEALER SHOWS ---   ',CARDS(I-1)
109.    108 960        FORMAT ('PLAYER = ',I2,' NO OF ACES - ',I1)
110.    108            WRITE(*,960)P,PACE
```

The parameter ´WIN´ is an output that HAND sets: it is "1" if the player wins, "0" if the dealer wins, and "2" if it is a tie. The code above initializes P and D to "0", which represent respectively the player´s and dealer´s total. Also PACE and DACE are initialized to be "0" - they represent ,respectively, the number of player aces and the number of dealer aces. Lines 102 - 105 deal the first four cards by calling subroutine HIT. The player gets the first and third card; the dealer gets the second and fourth. Lines 108 through 110 display the player´s total and dealer´s card to the CRT.

The next section of code handles the various Blackjack outcomes. If the player has Blackjack, lines 111 and 112 will write out the message and set WIN to "1".

```
111.    137         IF (P .EQ. 21) THEN
112.    142              WRITE(*,'(A)')'PLAYER HAS BLACKJACK'
113.    188              WIN=1
128.    367         ENDIF
129.    367 C** HANDLE THE BLACKJACK SITUATIONS, CASE WHEN DEALER HAS BLACKJACK:
130.    367         IF(D .EQ. 21) THEN
131.    372             WRITE(*,'(A)')'DEALER HAS BJ'
132.    411             IF( WIN .EQ. 1 ) THEN
133.    417                 WRITE(*,'(A)')'------ PUSH'
134.    453                 WIN=2
135.    456                 GO TO 13
136.    458             ELSE
137.    460                 WRITE(*,'(A)') 'DEALER AUTOMATICALLY WINS'
138.    511                 GO TO 13
139.    513             ENDIF
140.    513          ELSE
141.    515 C** CASE WHERE DEALER DOESN'T HAVE BLACKJACK:
142.    515 C** CHECK FOR PLAYER BLACKJACK OR FIVE CARD HAND:
143.    515           IF ((P .EQ. 21) .OR. (COUNT .GE. 5)) THEN
144.    524                WRITE(*,'(A)')'PLAYER AUTOMATICALLY WINS '
145.    576                WIN=1
146.    579                GO TO 13
147.    581           ENDIF
148.    581         ENDIF
```

If the dealer has Blackjack (line 130), the program checks the variable WIN. If it is "1", the player also has Blackjack; line 133 writes out the message ´PUSH´. Otherwise, the player does not have Blackjack and line 137 writes out the message ´DEALER WINS´. Label 13 is the end of the program HAND. Line 143 deals with the situation where the dealer does not have Blackjack: if the player has Blackjack or the player´s card count is greater than or equal to five, the player wins.

Lines 150 - 161 hit the dealer. Label 13 is the end of the program.

```
150.    581         WRITE(*,970)D
151.    605 970     FORMAT('DEALER HAS ',I2)
152.    605 12      IF(D .LE. 16) THEN
153.    610             CALL HIT(D,DACE)
154.    616             WRITE(*,970)D
155.    641             IF (D .GT. 21) THEN
156.    646                WRITE(*,'(A)')'DEALER BUSTS - PLAYER WINS'
157.    698                WIN=1
158.    701                GO TO 13
159.    703             ENDIF
160.    703             GO TO 12
161.    705         ENDIF
```

The logic for hitting the player is on lines 115 - 127 below:

```
114.    191         ELSE
115.    193              COUNT=2
116.    196 11           WRITE(*,'(A$)')'HIT?'
117.    226              READ(*,'(I1)') K
118.    258              IF( K .EQ. 1) THEN
119.    263                  CALL HIT(P,PACE)
120.    269                  COUNT=COUNT+1
121.    274                  WRITE(*,960)P,PACE
122.    303                  IF(P .GT. 21) THEN
123.    308                      WRITE(*,'(A)')'   PLAYER BUSTS - DEALER WINS'
124.    363                      GO TO 13
125.    365                  ENDIF
126.    365                  GO TO 11
127.    367          ENDIF
```

Count is set to be "2" and bumped for every hit (line 120). If the player exceeds 21, line 123 writes a player bust message; the GOTO on line 124 jumps to the exit label.

The entire program is shown below. The only new code is in lines 173 through 174. This code determines the winner and contains label 13 on the END statement.

Next we will discuss testing the program flow. Its graph is shown in Figure B-2.

```
  0 C**
  0 C**
  0 C**
  0           SUBROUTINE HAND(WIN)
  0 C**
  0           INTEGER P,D,PACE,DACE,I,CARDS(52),DEBUG,COUNT,WIN
  0           COMMON /DECK/CARDS,I,DEBUG
  0 C**
  0           P=0
  3           D=0
  6           PACE=0
  9           DACE=0
 12           WIN=0
 15 C**     WIN WILL BE 0 IF DEALER WINS,1 IF PLAYER WINS,2 IF A PUSH
 15           CALL HIT(P,PACE)
 21           CALL HIT(D,DACE)
 27           CALL HIT(P,PACE)
 33           CALL HIT(D,DACE)
 39           COUNT=0
 42 C**
 42           WRITE(*,'(A,I2)')'DEALER SHOWS ---   ',CARDS(I-1)
108 960       FORMAT ('PLAYER = ',I2,' NO OF ACES - ',I1)
108           WRITE(*,960)P,PACE
137           IF (P .EQ. 21) THEN
142                WRITE(*,'(A)')'PLAYER HAS BLACKJACK'
188                WIN=1
191           ELSE
193                COUNT=2
196 11             WRITE(*,'(A$)')'HIT?'
226                READ(*,'(I1)') K
258                IF( K .EQ. 1) THEN
263                    CALL HIT(P,PACE)
269                    COUNT=COUNT+1
274                    WRITE(*,960)P,PACE
303                    IF(P .GT. 21) THEN
308                        WRITE(*,'(A)')'   PLAYER BUSTS - DEALER WINS'
363                        GO TO 13
365                    ENDIF
365                    GO TO 11
367                ENDIF
367           ENDIF
367 C** HANDLE THE BLACKJACK SITUATIONS, CASE WHEN DEALER HAS BLACKJACK:
367           IF(D .EQ. 21) THEN
372               WRITE(*,'(A)')'DEALER HAS BJ'
411               IF( WIN .EQ. 1 ) THEN
417                   WRITE(*,'(A)')'------ PUSH'
453                   WIN=2
456                   GO TO 13
458               ELSE
460                   WRITE(*,'(A)') 'DEALER AUTOMATICALLY WINS'
511                   GO TO 13
513               ENDIF
513            ELSE
515 C** CASE WHERE DEALER DOESN'T HAVE BLACKJACK:
515 C** CHECK FOR PLAYER BLACKJACK OR FIVE CARD HAND:
515             IF ((P .EQ. 21) .OR. (COUNT .GE. 5)) THEN
524                  WRITE(*,'(A)')'PLAYER AUTOMATICALLY WINS
576                  WIN=1
579                  GO TO 13
581             ENDIF
581           ENDIF
581 C**
581           WRITE(*,970)D
605 970       FORMAT('DEALER HAS ',I2)
605 12        IF(D .LE. 16) THEN
610               CALL HIT(D,DACE)
616               WRITE(*,970)D
641               IF (D .GT. 21) THEN
646                  WRITE(*,'(A)')'DEALER BUSTS - PLAYER WINS'
698                  WIN=1
701                  GO TO 13
703               ENDIF
703               GO TO 12
705           ENDIF
705 C**
705 980       FORMAT(' PLAYER = ',I2,'  DEALER = ',I2)
705           WRITE(*,980) P,D
733           IF(P .GT. D) THEN
738               WRITE(*,'(A)')'PLAYER WINS'
775               WIN=1
778           ELSE
780               WRITE(*,'(A)')'DEALER WINS'
817           ENDIF
817
817 C**
817 C**
817 13      END
```

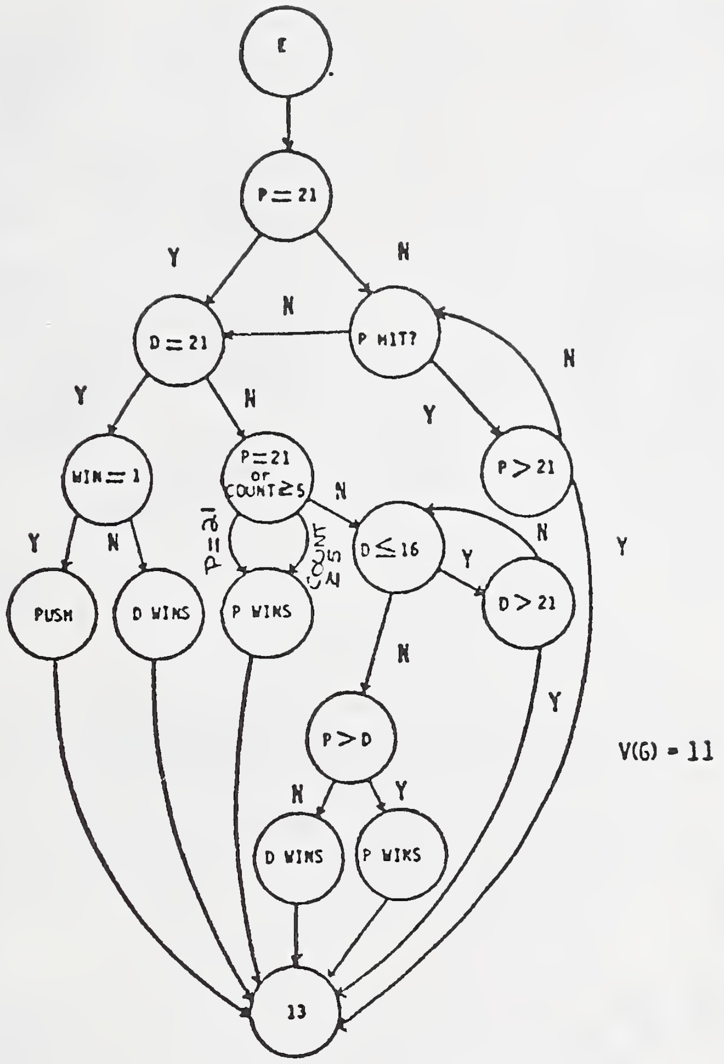
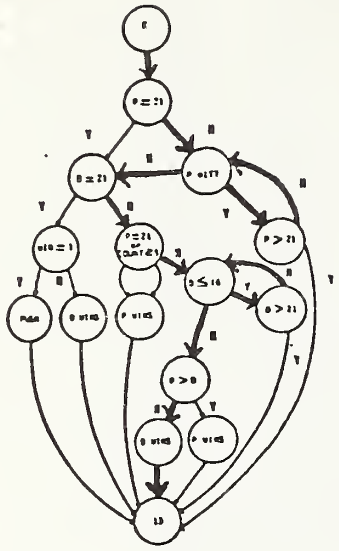


Figure B-2 Blackjack Graph

We will next step through the structured testing process test-by-test. The first step is to establish a baseline test. A representative flow through the program could be the data shown in the Figure B-3 where the player takes a hit; the dealer takes a hit; and the dealer wins. This baseline path is shown in Figure B-3 with Test 1 as a darkened path.

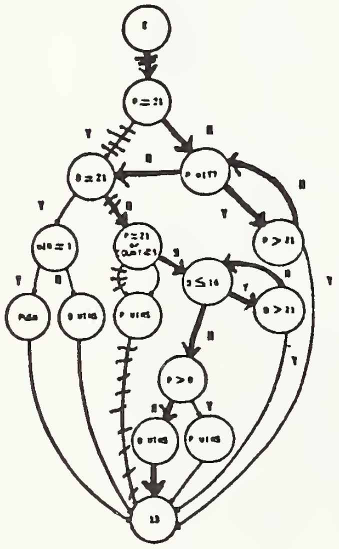
Test 1 is the actual output of the computer run. Where it says "NEXT CARD?", a debug option has been turned on that allows the testing person to supply the next card in order to select particular paths. This debugging data is indented to the right. Test 1 conforms to the blackjack specification and shows no errors.



```
                    NEXT CARD?10
                    NEXT CARD?10
                    NEXT CARD?5
                    NEXT CARD?4
DEALER SHOWS ---- 4
PLAYER = 15  NO OF ACES - Ø
HIT?1
                    NEXT CARD?4
PLAYER = 19  NO OF ACES - Ø
HIT?Ø
DEALER HAS 14
                    NEXT CARD?6
DEALER HAS 20
   PLAYER = 19  DEALER = 20
DEALER WINS
```

Figure B-3 Test 1

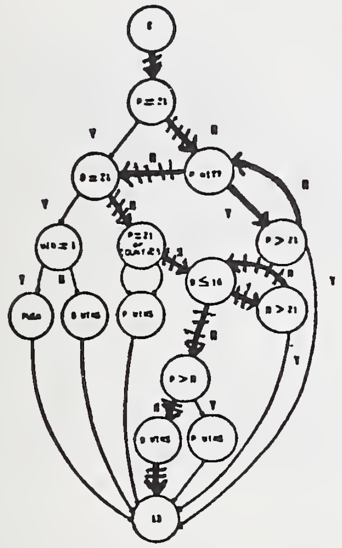
Using the baseline method to select the second path, we must flip the first decision (p = 21). This results in the data and path shown in Figure B-4 with hash marks. No error results.



```
                    NEXT CARD?10
                    NEXT CARD?10
                    NEXT CARD?11
                    NEXT CARD?6
DEALER SHOWS ---- 6
PLAYER = 21  NO OF ACES - 1
PLAYER HAS BLACKJACK
PLAYER AUTOMATICALLY WINS
```

Figure B-4 Test 2

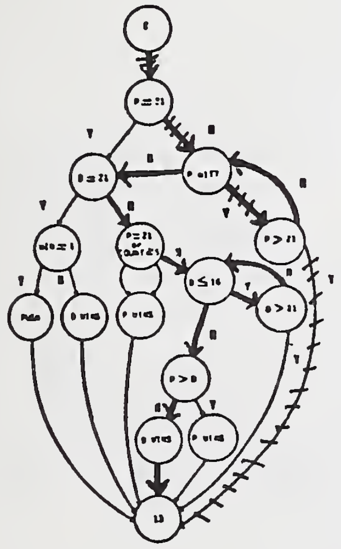
Test 3 is performed by flipping the second decision in the baseline path and comming back to the baseline. This means that the player would not take a hit. The test data in Figure B-5 realizes this path and shows no errors:



```
                         NEXT CARD?10
                         NEXT CARD?10
                         NEXT CARD?8
                         NEXT CARD?5
DEALER SHOWS ---- 10
PLAYER = 18  NO OF ACES - Ø
HIT?Ø
DEALER HAS 15
                         NEXT CARD?5
DEALER HAS 20
  PLAYER = 18  DEALER = 20
DEALER WINS
```

Figure B-5 Test 3

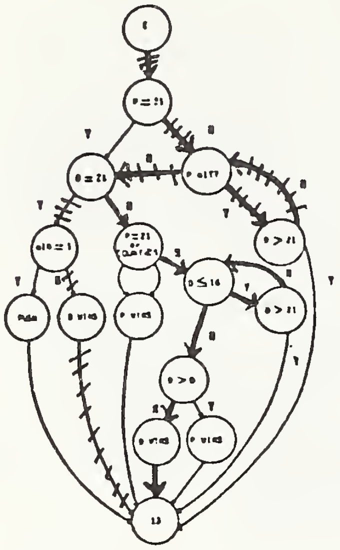
Test 4 is done by flipping the third decision in the baseline (P>21). This tests busting the player. This path and test data are shown in Figure B-6; no error is found.



```
                         NEXT CARD?10
                         NEXT CARD?9
                         NEXT CARD?7
                         NEXT CARD?10
DEALER SHOWS ---- 10
PLAYER = 17  NO OF ACES - Ø
HIT?1
                         NEXT CARD?6
PLAYER = 23  NO OF ACES - Ø
  PLAYER BUSTS - DEALER WINS
```

Figure B-6 Test 4

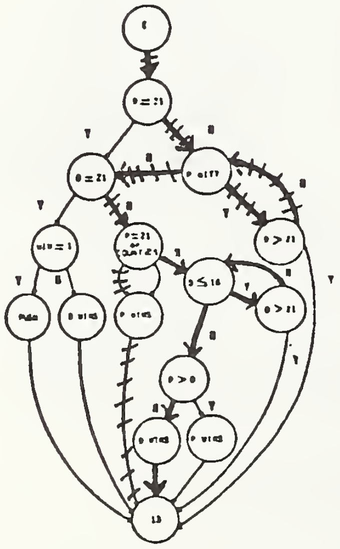
Test 5 is to flip the fourth decision (D = 21). To do this from the baseline path results in the test path and data in Figure B-7. It conforms to the specification - no error is found.



```
                     NEXT CARD?10
                     NEXT CARD?10
                     NEXT CARD?4
                     NEXT CARD?11
DEALER SHOWS ---- 11
PLAYER = 14  NO OF ACES - Ø
HIT?1
                     NEXT CARD?7
PLAYER = 21  NO OF ACES - Ø
HIT?Ø
DEALER HAS BJ
DEALER AUTOMATICALLY WINS
```

Figure B-7 Test 5

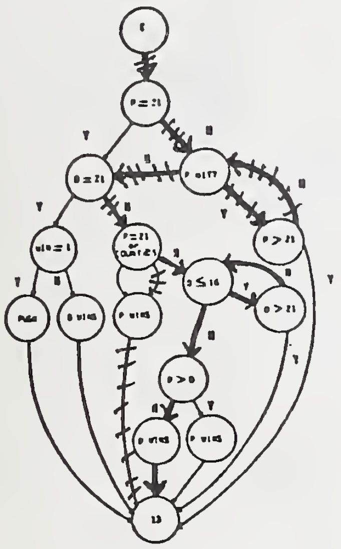
Test 6 we get by flipping the 5th decision in the baseline. We must have the player equal to 21 after a hit, the test path is shown in Figure B-8. An error is found - the player automatically wins with 3-card 21 before the dealer has a chance to hit.



```
                     NEXT CARD?10
                     NEXT CARD?7
                     NEXT CARD?6
                     NEXT CARD?10
DEALER SHOWS ---- 10
PLAYER = 16  NO OF ACES - Ø
HIT?1
                     NEXT CARD?5
PLAYER = 21  NO OF ACES - Ø
HIT?Ø
PLAYER AUTOMATICALLY WINS
```

Figure B-8 Test 6

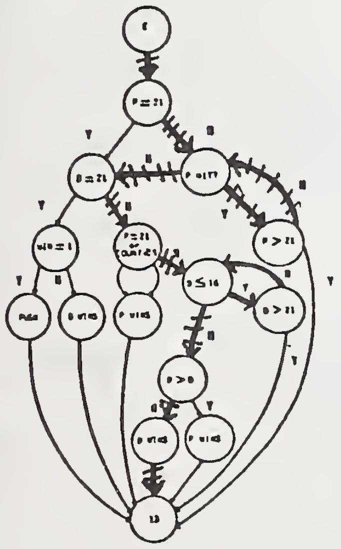
Test 7 is realized by flipping the sixth decision (COUNT >= 5) in the baseline. The path and data are shown in Figure B-9 which show no error.



```
                 NEXT CARD?5
                 NEXT CARD?10
                 NEXT CARD?4
                 NEXT CARD?10
DEALER SHOWS ---- 10
PLAYER = 9  NO OF ACES - Ø
HIT?1
                 NEXT CARD?4
PLAYER = 13 NO OF ACES - Ø
HIT?1
                 NEXT CARD?2
PLAYER = 15 NO OF ACES - Ø
HIT?1
                 NEXT CARD?11
PLAYER = 16 NO OF ACES - Ø
HIT?Ø
PLAYER AUTOMATICALLY WINS
```

Figure B-9 Test 7

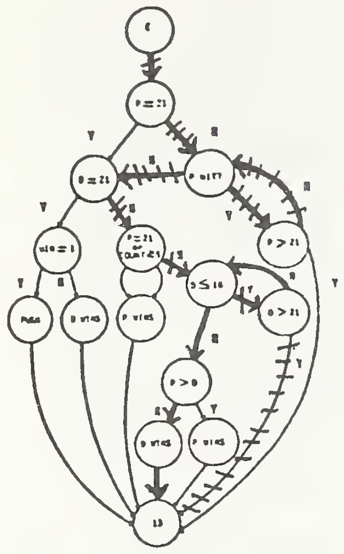
Test 8 is encountered by flipping the seventh decision which implies the dealer will not take a hit. The path and data are shown in Figure B-10 which conforms to the blackjack specification.



```
                 NEXT CARD?10
                 NEXT CARD?10
                 NEXT CARD?6
                 NEXT CARD?10
DEALER SHOWS ---- 10
PLAYER = 16  NO OF ACES - Ø
HIT?1
                 NEXT CARD?4
PLAYER = 20  NO OF ACES - Ø
HIT?Ø
DEALER HAS 20
  PLAYER = 20  DEALER = 20
DEALER WINS
```

Figure B-10 Test 8

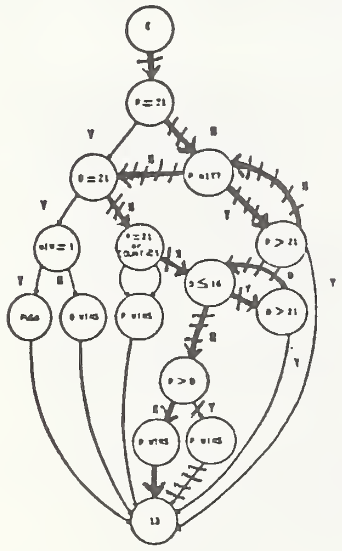
Flipping the eighth decision, we get Test 9 - the dealer busts. This test run conforms to the specification.



```
                    NEXT CARD?10
                    NEXT CARD?10
                    NEXT CARD?3
                    NEXT CARD?4
DEALER SHOWS ---- 4
PLAYER = 13  NO OF ACES - Ø
HIT?1
                    NEXT CARD?5
PLAYER = 18  NO OF ACES - Ø
HIT?Ø
DEALER HAS 14
                    NEXT CARD?8
DEALER HAS 22
DEALER BUSTS - PLAYER WINS
```

Figure B-11 Test 9

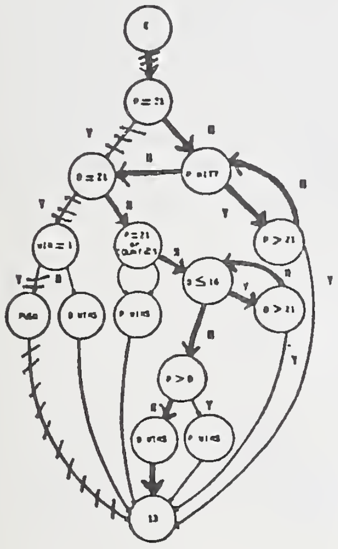
Test 10 is arrived at by flipping the ninth decision in the baseline which means we finally let the player win. The run in Figure B-12 conforms to the specification.



```
                    NEXT CARD?10
                    NEXT CARD?10
                    NEXT CARD?3
                    NEXT CARD?6
DEALER SHOWS ---- 6
PLAYER = 13  NO OF ACES - Ø
HIT?1
                    NEXT CARD?7
PLAYER = 20  NO OF ACES - Ø
HIT?Ø
DEALER HAS 16
                    NEXT CARD?3
DEALER HAS 19
  PLAYER = 20  DEALER = 19
PLAYER WINS
```

Figure B-12 Test 10

We have flipped every decision on the baseline and have 10 tests. Since the complexity is 11, there must be another decision to flip (WIN = 1). We do so and generate the 11th and final test shown in Figure B-13:



```
                              NEXT CARD?10
                              NEXT CARD?10
                              NEXT CARD?11
                              NEXT CARD?11
DEALER SHOWS ---- 11
PLAYER = 21  NO OF ACES - 1
PLAYER HAS BLACKJACK
DEALER HAS BJ
-------------PUSH
```

Figure B-13 Test 11

Observations

A few comments about this error and the difficulty of finding it are in order. One common approach to testing is to use, as a testing criterion, the rule that every statement be exercised at least one time. If we use this testing criterion, the following tests would satisfy it: Test 1, Test 2, Test 4, Test 5, Test 9, Test 10, and Test 11. Notice that the above sets of tests do, indeed, exercise every statement in the FORTRAN program at least one time and unfortunately do not detect the error.

Another more rigorous testing criterion sometimes used is to require that every edge of the flowgraph be traversed at least one time. The following set of tests satisfy this criteria: Test 1, Test 2, Test 4, Test 5, Test 7, Test 9, Test 10, Test 11. Notice that the above set of tests do indeed cover every edge of the graph one time but unfortunately the error would not be detected.

With this example, the structured testing criteria imposed additional tests beyond every statement and every edge; this is typical. The structured testing criteria will guarantee satisfying the weaker criteria and typically produces several additional tests. Notice in this particular example, the structured testing technique forced a combination of edges that worked properly by themselves but not when taken in combination.

It is interesting to hypothesize how effective functional testing would turn out in this case compared to the structured testing technique we have described. This particular system is operational and has been field tested for several weeks. Several people were asked to use the system; they were introduced to the Blackjack game as an experimental system that is undergoing a field-type test. They were given a copy of the specification and told how to interact with the computerized game. At this point, 139 games have been played (the computer won 82; the player won 57). Several comments have been made about the readability of the messages and some human factor considerations. Nobody, as of this date, has reported the error found in Test 6.

REFERENCES

[BASI] Basli, V.R. and Reiter, R.W. "Evaluating Automatable Measures of Software Development", Workshop on Quantitative Software Models, New York:IEEE, 1979.

[BERG] Berge, C. Graphs and Hypergraphs. Amsterdam, The Netherlands: North-Holland, 1973.

[CHAP] Chapin, N., "A Measure of Software Complexity," AFIPS Conference Proceedings, NCC, Vol. 48, 1979, pp. 995-1002.

[CHEN] Chen, E.T., "Program Complexity and Programmer Productivity", Proceedings of Computer Software and Applications Conference, 1977, pp. 142-148.

[CURTa] Curtis, B., Sheppard, S.B., & Milliman, P. "Third time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proceedings of the Fourth International Conference on Software Engineering, New York: IEEE, 1979.

[CURTb] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. "Predicting Performance on Software Maintenance Tasks with the Halstead and McCabe metrics",IEEE Transactions on Software Engineering, 1979, 5, pp. 95-104.

[GILB] Gilb, T., Software Metrics, Winthrop Publishers, Inc., Cambridge, MA, 1977.

[HALS] Halstead, M.H., Elements of Software Science, Elsevier North Holland, New York, 1977.

[HENR] Henry, S., Kafura, D., & Harris, K. "On the Relationships Among Three Software Metrics", 1981 ACM Workshop/ Symposium on Measurement and Evaluation of Software Quality, March 1981.

[JELI] Jelinski, Z., & Moranda, P.B., Metrics of Software Quality, McDonnell Douglas Astronautics Co., Report MDC-G7517, Ad-A077896, August 1979.

[KOLE] Kolence, K.W., "Software Physics and Computer Performance Measurement," Proceedings of the ACM 1972 Annual Confreence, pp. 1024-1040.

[KERN] Kernighan, B.W, & Plauger, P.J., The Elements of Programming Style, New Jersey: Bell Telephone Laboratories, 1974.

[KNUT] Knuth, D.E. "An Empirical Study of FORTRAN Programs", Software Practice and Experience, April-June 1971, pp. 105-133.

[LEGA] Legard, H,& Marcotty, M. "A Geneology of Control Structures", CACM,18, pp. 629-639, Nov. 1975.

[MCAL] McCall, J.A., et al., Factors in Software Quality, RADC-TR-77-369, Vol. I, II, III (AD-A049-014, -015, -055), General Electric Co., Sunnyvale, CA, July 1977.

[MCCA] McCabe, T.J. "A Complexity Measure", IEEE Trans. on Software Engineering, SE-2 No. 4, pp. 308-320, Dec. 1976.

[MILL] Miller, E.F. "Program Testing: Art Meets Theory," Computer,10, No. 7, pp. 42-51, July 1977.

[MOHA] Mohanty, S.N. & Adamowicz, "Proposed Measures for the Evaluation of Software," Proceedings of Symposium on Computer Software Engineering, N.Y., April 1976, pp. 485-497.

[MYER] Myers, G.J. "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, 1977.

[PAIG] Paige, M. "An Analytical Approach to Software Testing," Proceedings COMPSAC 78, Chicago, 1978, IEEE Computer Society, New York, pp. 527-532.

[SHEP] Sheppard, S.B. and Kruesi, E. "The Effects of the Symbology and Spatial Arrangement of Software Specifications in a Coding Task", Technical Report TR-81-388200-3. Arlington, VA: General Electric Company, 1981.

[WALS] Walsh, T.J. "A Software Reliability Study Using a Complexity Measure", In Proceedings of the National Computer Conference. New York: AFIPS, 1979.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS SP 500-99	2. Performing Organ. Report No.	3. Publication Date December 1982

4. TITLE AND SUBTITLE

Computer Science and Technology:
Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric

5. AUTHOR(S)

Thomas J. McCabe

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

McCabe & Associates, Inc.
5550 Sterrett Place
Columbia, MD 21044

7. Contract/Grant No.

AE8892

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Patricia Powell
National Bureau of Standards
Room B267, Bldg. 225
Washington, DC 20234

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 82-600651

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Various applications of the Structured Testing methodology are presented. The philosophy of the technique is to avoid programs that are inherently untestable by first measuring and limiting program complexity. Part 1 defines and develops a program complexity measure. Part 2 discusses the complexity measure in the second phase of the methodology which is used to quantify and proceduralize the testing process. Part 3 illustrates how to apply the techniques during maintenance to identify the code that must be retested after making a modification.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

measures; metric; program complexity; software testing; structured testing

13. AVAILABILITY

[XX] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
[XX] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

72

15. Price

$5.00

USCOMM-DC 6043-P80

☆ U.S. GOVERNMENT PRINTING OFFICE : 1982 O—380-997 (2365)