1976-06

Analysis of program structure and error characteristics as applied to NTDS programs.

Kirchgaessner, Michael

Monterey, California. Naval Postgraduate School

ANALYSIS OF PROGRAM STRUCTURE AND
ERROR CHARACTERISTICS AS APPLIED
TO NTDS PROGRAMS

Michael Kirchgaessner

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

ANALYSIS OF PROGRAM STRUCTURE AND
ERROR CHARACTERISTICS AS APPLIED
TO NTDS PROGRAMS

by

Michael Kirchgaessner

June 1976

Thesis Advisor: N. F. Schneidewind

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Analysis of Program Structure and Error Characteristics as Applied to NTDS Programs	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1976
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael Kirchgaessner	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June 1976
	13. NUMBER OF PAGES 105
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program structure
Program complexity
NTDS programs
Program testing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A simulation model for the evaluation of program structure and error detection has been applied to the analysis of selected parts of NTDS programs. The simulation results were used to establish the relationship between program structure and measures of program complexity. This information would be used for the design and testing of software.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DD Form 1473
1 Jan 73
S/N 0102-014-6601

ANALYSIS CF PRCGRAM STRUCTURE AND ERROR CHARACTERISTICS AS APPLIED TC NTDS PROGRAMS

by

Michael Kirchgaessner
Lieutenant-Commander
Federal German Navy

Submitted in partial fulfillment of the requirements for the degree cf

MASTER CF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOCL

June 1976

ABSTRACT

A simulation model for the evaluation of program structure and error detection has been applied to the analysis of selected parts of NTDS programs. The simulation results were used to establish the relationship between program structure and measures of program complexity. This information would be used for the design and testing of software.

TABLE OF CONTENTS

I. INTRODUCTION

When is a program considered to be trivial? One answer to this question heard very often is "When it contains no bugs". Although this statement might be questionable, the converse is true, as there are few nontrivial programs that do not contain bugs. As the author of a critical and fundamental study of program design states: "...These bugs can never be completely exorcised in any program over some critical degree of complexity. Six months or even seven years after 'final debugging' errors crop up inevitably in the best of programs."[4]. This is a fact one has to live with, and there are only two things one can do about it: First to reduce the possibilities for bugs by careful design and use of modern programming techniques, second to devise careful testing techniques to detect and locate the bugs still remaining in the program.

Fig. 1 shows the relationship between hardware and software cost in the U.S. during the period from 1955 to 1985. Due to the fact that the software cost continues to rise and that about 50% of this cost is for testing and integration of a system [7], it is important to obtain a realistic assessment of how much effort has to be spent to test the newly designed program based on its size, structure and characteristics. If one is able to determine in the design stage the best possible structure with respect to the error detection capabilities, then bugs can be avoided and testing will be reduced. Also early in the development of a project a realistic allocation of coding and testing resources could be made.

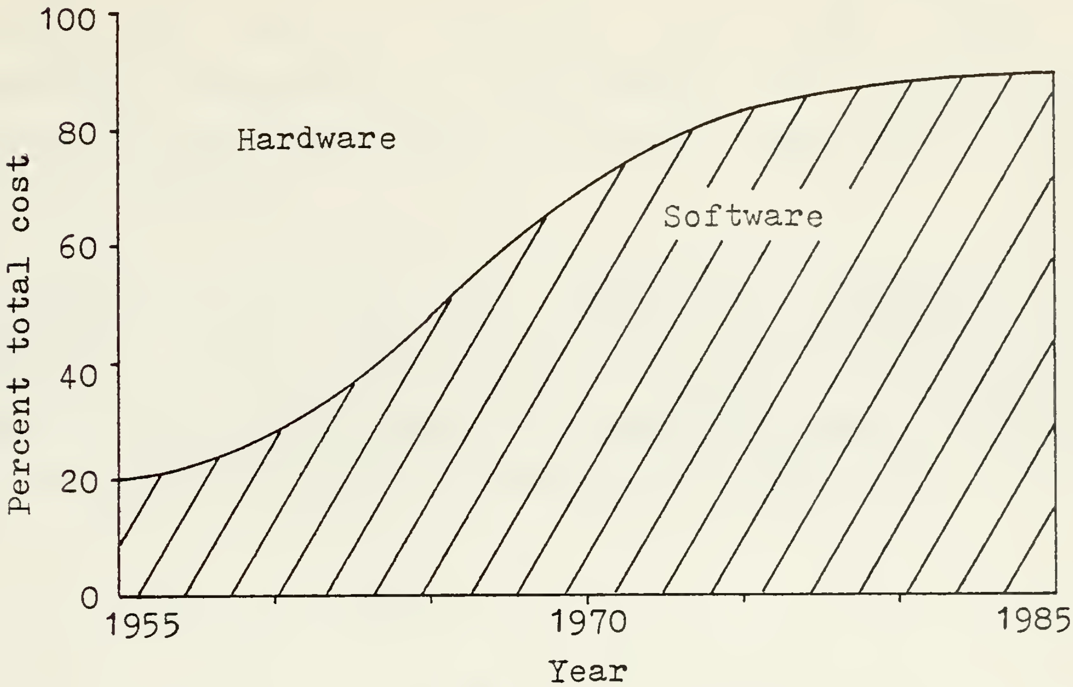


Figure 1 - SOFTWARE COST TREND IN THE U.S.
[Datamation, Sept. 1974, pg. 75]

In order to address these problems, a Software Error Detection Simulation Model has been developed [7,10]. This model was was used to identify program complexity measures which were correlated with error detection. Naval Tactical Data System programs were used for this purpose.

The structures of these NTDS-programs have been analyzed (see Chapter VI) and put into the form of directed graphs.

The data gained from the directed graph representation were used as inputs for the Error Detection Simulation Model. The results gained and the conclusions and recommendations drawn from these results are shown in Chapter VII. For reasons of security, the programs or the parts of them are not identified by names. Instead, a sequential number scheme for identifying the programs has been employed.

This work is part of a research effort sponsored by the NADC to get software evaluation aids which provide an economical assessment of the design and testing effort needed for the development of avionics and other complex software projects.

Because it is felt that efforts in testing and in debugging can be more successful if one employs modern techniques in the production of programs, an introductory chapter shows the relevance of modern programming techniques to the problem of program testing and maintenance.

II. DEFINITIONS

There was originally a lack of commonly used definitions for program testing. Only recently has a "definitional framework" emerged and very good program testing definitions are found in Ref. 8, pg. 7 - 14. In order to be consistent and to specify the meaning of keywords within this thesis, the following definitions have been adopted:

1. Program Structure

The structure of a program is a description of the underlying logic and data flow as represented in the form of a directed graph with its set of nodes and edges (arcs).

2. Reachability Index

Reachability index is a measurement of the possibilities to get to a specified node, computed over all nodes of the directed graph. It is computed with the formula:

$$R = \sum \text{path to node } (i).$$

3. Debugging

Debugging is the action one takes to locate and correct a known or detected error in a program.

4. <u>Testing</u>

Testing is the action to check whether a program meets its specifications and to establish the presence of errors in it.

5. <u>Life cycle of a program</u>

The life cycle of a program consists of the following phases:

- design
- Coding
- Debugging
- Testing
- Production and maintenance.

III. MODERN PROGRAMMING TECHNIQUES

Two recent developments in the theory and practice of software development are addressed here as important because they are relevant not only for the actual writing of the code of the program, but also to debugging, testing, and integrating software systems as well, namely the advent of modular and structured programming. The advantages of these techniques are obvious for the programmer when he develops his program. Programs written using these techniques are easier to read and to understand as far as the flow of the logic is concerned. Also, the tester can better understand the logic of a program when these techniques are employed. Furthermore, it has been proposed for structured programs to eliminate flowcharts as media of communication [13], so it is necessary to understand how much testing, integration and maintenance of software are influenced by this development.

A. MODULAR PROGRAMMING

Modular programming is a system to develop programs as a set of interrelated individual units (called modules) which later can be linked together to form a complete program [9]. Thus modular programming is not simply splitting up a program into several parts (subroutines), but rather dividing the software according to the functions to be performed. The designer faces the one crucial problem which will determine success or failure, namely to specify completely and carefully the interface between the individual modules.

Modules as individual program units should have the following properties:

(1) One module should perform only one basic function
(2) The size of a module should be such that it is easily understood and contains only a moderate amount of code
(3) A module should be designed in such a way that it has only a few control or data paths
(4) One module should process only a small amount of data.

The design of programs in this way leads not only to cleaner and more productive coding but also to easier and more flexible testing. The advantages with respect to debugging and testing show up in several ways. Single modules can be debugged and tested independently from the other modules or the main (driver) program. Furthermore, if the modules are small enough, extensive testing generally assumed as impossible with the exception of very trivial programs, can become manageable. This in turn leads to more reliable programs. If all modules of a software project can be tested extensively, a highly reliable program can be produced. Even if one falls short of this goal - and this happens in most cases due to the very large number of possible inputs and program paths - the final program will be more reliable and more thoroughly tested than a non-modular program. The possibility of testing modules individually provides for better (more economical) allocation of testing resources, because one does not have to wait until the whole program has been completed. However, to test individual modules, special test-routines are needed as drivers and if other modules must interact, dummy modules must be created if the real modules are not yet available or not yet tested.

One final point in favour of modular programming has to be made: Normally, no production program is completed until the day when it is no longer used, i.e. every running production program has to be maintained and adapted to new considerations and situations. Because of the simplicity of the overall organization of modular programs this software maintenance is alleviated since interactions between modules are more easily understood; hence, the effect of program changes is easier to identify. Also only the modules affected by the change have to be tested (together with the main program and interacting modules).

B. STRUCTURED PROGRAMMING

Having coded a program in the above described modularized fashion, there is still room for improvement. Since Dijkstra's famous letter to the editor of the Communications of the ACM in which he proposed to eliminate GO-TO statements [5], the concept of Structured Programming has evolved and led to further simplification of the coding process.

Simplification means here not that the actual code is easier to write - although this might be the case too for a programmer who is familiar with the concept and can think in these terms - but the code produced and the control sequence of the finished program is simpler than in a nonstructured program. This simplification has been theoretically demonstrated by Boehm and Jacopini as early as 1966 [3].

Although there are as many interpretations of what Structured Programming is as there are authors on this topic, the following features are essential and common to this concept:

(1) TOP-DOWN Design, i.e. the design starts at a very general level and proceeds stepwise to the specific and detailed tasks
(2) Modular Design
(3) Limited possibilities to control the logic flow of the program, namely only

    * sequential
    * conditional: IF - THEN - ELSE
    * iterative: DO - WHILE

    statements are allowed.

Whereas the so called block-structured languages like ALGOL or PL/I lend themselves to this form of coding (although GO-TO statements are provided by the language), even in FORTRAN the implementation of some of the basic principles of Structured Programming is possible if the programmer concerned with a structural flow of his program chooses the branching caused by unavoidable GO-TO statements carefully.

Baker [1] shows that the application of Structured Programming combined with the "Chief Programmer Team Method" of organizing a software project [2] can bring measurable improvements in software development, in the coding as well as in the debugging and in the testing stage. Due to the fact that Structured Programming implies Modular Programming the same advantages hold here too, i.e. the software is easier to test and to maintain after release.

IV. THE PROBLEM OF PROGRAM COMPLEXITY

The impact of the programming techniques described above on the economic development of reliable and maintainable software is directly related to the problem of program complexity. There is so far no generally adopted definition of what program complexity really means. The definition is dependent on the context in which one wants to examine program complexity. Here complexity is defined as structural properties of a program that affect the ability to detect errors.

Under the condition that the structure of a program is described by a directed graph, the following criteria can be used to measure its complexity:

1. Number of nodes
2. Number of arcs
3. Number of possible paths through the program
4. Number of source statements
5. Average path length (source statements per path, arcs per paths)
6. Reachability index
7. Fullness index (ratio of actual to maximum number of arcs).

Although Mills in his contribution to Ref. 8 generates the idea of equating program complexity with the difficulty of understanding a program and justifies this approach with "...the frustration of concocting and demolishing more simple minded direct ideas, such as counts of branches, data references, etc.", his approach does not help to get a real measurement of complexity such that one is able to make a quantitative statement how complex a program is. It seems that the important point is to relate program complexity to the problem area one pursues. The analysis of NTDS-Pograms has given insight in methods to measure complexity with respect to problems of program design and testing.

V. ERROR DETECTION SIMULATION MODEL

A. GENERAL

A Software Error Detection Simulation Model was originally developed by T.F. Green in his M.S. Thesis [7] and subsequently modified by professor G.T. Howard of the Naval Postgraduate School. Written in FORTRAN it was designed to run on the IBM 360/67 computer of the Naval Postgraduate School. Originally it had been tested against hypothetical and actual programs. It was shown that simulation of error detection was feasible and that information could be obtained on the relationship between error detection and program complexity. However, it was necessary to perform additional model feasibility tests by using the model on a large number of actual programs. In the process of testing some of the original features had to be removed and provisions had to be made for cases of program behaviour which were unexpected at the time of the simulation program design. A detailed description of the model with its specific assumptions and capabilities is found in Ref. 10, pg. IV-5 - IV-39.

B. PROGRAM REPRESENTATION

The prerequisite for the use of the simulation model is to get the structure of a program that has to be tested in the form of a directed graph. A directed graph is a convenient means to show the structure of programs. It is suitable for showing the control flow in a program, measures of complexity can be derived from this kind of representation. In addition, the "control flow graph" as this composition of structures is sometimes called, is also very useful for determining the execution time of a structure on a machine. This representation of program structures also simplifies the representation of large and complex programs because these programs can be broken up in logical segments (modules, procedures, subroutines etc.), and the segments can be tested separately from the other parts of the program.

C. CURRENT STATUS OF THE SIMULATION PROGRAM

1. Input Variables

The following input variables have to be used for the simulation:

a. MINPUT designates the number of inputs within each replication.

b. NUMOUT is the number of replications (number of paths) within every repetition.

c. NREPET is the number of reseedings with errors (repetitions).

d. MEANLN designates the mean arc length if the arc lengths are selected at random by the program and are not read in.

e. MEANER designates the mean number of instructions between errors.

f. N is the number of nodes within the structure.

g. Input for the Adjacency Matrix is done in a shorthand notation:

For every node with the exception of the last nodes there is one data card which contains information about this node in the following sequence: Identification of the node, number of arcs emanating from this node, identification numbers of the nodes to which the arcs go.

h. Input for the matrix of arc lengths (optional) similar to that for the adjacency matrix: Instead, only as the identifiers for receiving nodes the pair (identifier, number of statements on this arc) has to be provided.

i. Input to plant errors in arcs instead of letting the program seed them at random: Input as for matrix of arc length, but the number of errors on this arc has to be specified instead of the number of statements.

j. MOUT specifies the desired output:

   0 = Summary output

   1 = Extensive output (NUMOUT * NREPET ≤ 25)

2. <u>Input Formats</u>

The input formats are as follows:

First data card: (6I5) MINPUT, NUMOUT, NREPET, MEANLN, MEANER, N.

Second and following cards: adjacency matrix, (16I5); followed by delimiter-card: 99 in columns 4 and 5.

Input cards for matrix of arc length (optional): 2I5, 7(I5,F5.0); followed by delimiter-card: 99 in columns 4 and 5.

Input to seed errors manually (optional) : 16I5; delimiter-card: 99 in columns 4 and 5.

Last data card (output specification): I5.

Note that all delimiter cards are _not_ optional.

3. Limitations

This simulation program is currently restricted to accomodate a maximum number of 30 nodes. The execution time for simpler structures (about 10 - 15 nodes) is within a five minute time limit. Larger and more complex structures with more nodes and possible paths through the structure require a 30 minute time frame for the execution of one simulated input in 100 replications and 100 repetitions.

An extension of the limits of the program to accomodate larger structures seems to be impractical because of the fast rise of memory space and execution time required.

4. Program Listing

A listing of the current error detection simulation program as it was used for the analysis of the NTDS-programs is found in Appendix A.

VI. ANALYSIS OF NTDS PROGRAMS

A. GENERAL

In order to demonstrate the practicality of program analysis using the Error Detection Simulation Model, Naval Tactical Data Systems Programs have been analyzed by

1. describing the structure by converting the programs into the form of directed graphs
2. running these structures on the error detection simulation model and
3. evaluating the simulation results with respect to measures of program complexity.

B. DESIGN OF NTDS PROGRAMS

1. Modular Design

The design of NTDS programs is characterized by Modular Programming, both in general and in detail, and the modular design is a characteristic of the hardware as well. Also the actual implementation of every NTDS installation consists of hardware and software building blocks that are composed to fit exactly the need of each installation.

Although NTDS programs are really programmed in a modular fashion, the term "module" does not have the same meaning as usual. Module usually refers to basic building blocks that are parts of the program, whereas NTDS programs are composed of subsystems. The NTDS-"Modules" in turn are divided up in parts which correspond to the "module"-definition of Modular Programming. In NTDS terminology these parts are called procedures. NTDS modules perform complex tasks such as tracking, display etc. They contain a medium to large number of dependent procedures. These procedures perform the basic functions intended in Modular Programming such as checking track properties. Throughout this discussion, "module" is used as in the NTDS system, namely as a complete subsystem for performing complex tasks.

The modular approach is imbedded in a stringent hierarchical system which is controlled by the priorities of the tasks to be performed. The levels of hierarchy are applied to the modules in such a way that only major subprograms which are designed to execute distinctive tasks can communicate with each other, whereas the procedures within the modules can only communicate according to the level of hierarchy they belong to, with the exception of calls to certain system routines.

2. CS-1 Language

The NTDS programs are written using the CS-1 high level language compiler [6]. This language has the advantage that it is well suited to the application area, namely tactical programs which run under severe constraints regarding time and memory space availability. Tables are searched in a very effective way, and another interesting feature is that assembly code can be interspersed within the high level code of the program. This fact gives the programmer a powerful means for controlling the hardware which in turn facilitates the production of effective code.

C. DIRECTED GRAPH CONSTRUCTION

In order to obtain the desired statistics and to analyze the data- and control flow of a single NTDS-program, the following method has been developed and used:

1. One complete module from an existing and currently operating NTDS program has been put into the form of a directed graph. The module has been decomposed into the procedures it contains, and every procedure is treated separately. Due to the modular design thoughout the program, no logical difficulties arise here, because every procedure has only one entrance and one exit point, i.e. the interface for interacting procedures within the module is uniquely defined. For each procedure the directed graph and the adjacency matrix have been constructed. As quantitative measurements the number of nodes, arcs, paths, loops, source statements, machine instructions, source statements per arc, and machine instructions per arc have been compiled.

2. The same work was done for randomly selected procedures from one other important module of the same program in order to obtain comparative results and to relate the reported number of errors to the different modules.

3. For the construction of the directed graphs and the gathering of the several statistics the following assumptions have been made:

   a. Nodes are associated with

      (1) Procedure entrance and exits
      (2) IF-statements (decision points)
      (3) Points where paths merge
      (4) Procedure calls within the module
      (5) Beginning and ending of loops

   b. All nodes within the module are distinct. They have individually assigned numbers (some nodes are indicated as "dummy" nodes), and they are counted only once, namely in the procedure they belong to.

   c. Entrance and exit nodes of a called procedure are regarded as "transient" nodes within the calling procedure, and one "transient arc" connects both transient nodes. This transient arc represents all the arcs inside the called procedure. The transient arcs are indicated in the drawings by a dashed line. Transient nodes have either the number of the entrance node of the called procedure or they are denoted by letters to distinguish them from the original nodes of the corresponding procedure.

   d. The length of every arc is indicated as the number of source statements or the number of machine instructions respectively. In the analysis the number of source statements has been used because programs are normally written in a high level language and this is the point where errors are introduced into the program.

4. Normally, the numbers of both source statements and machine instructions have been counted in the arc where the statements appear. However, because IF-statements and procedure calls result in branching, they have been counted in the arc leading to the corresponding node. Whereas for the counting of machine instructions, it would be possible in the case of an IF-statement to split the instruction sequence according to the arcs emanating from the decision point, this is not feasible for the source statement which contains the elements of both arcs emanating from it; it cannot be split.

The structures obtained from both modules and the compiled statistics are found in Appendix B. The following figure shows how to read the structure diagrams:

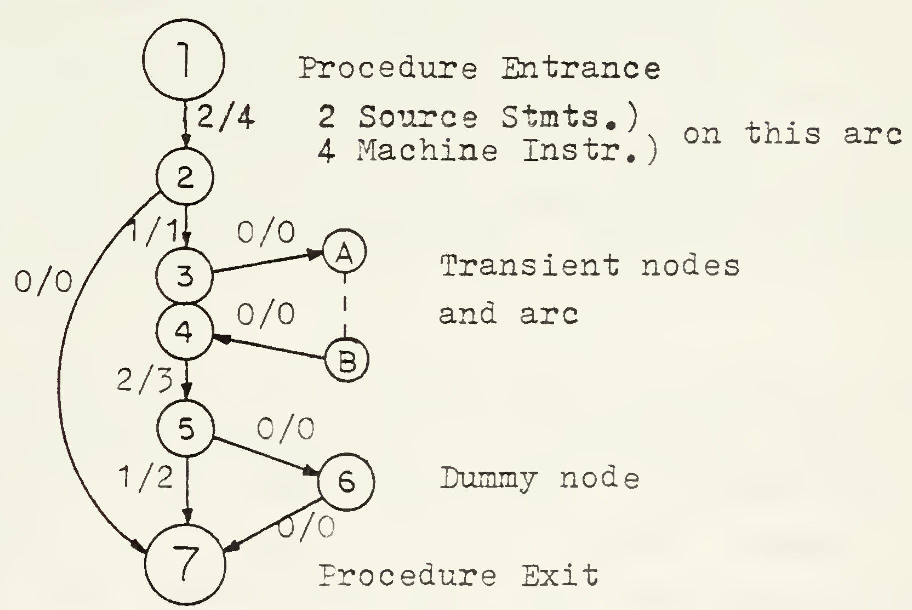


Figure 2 - EXAMPLE OF PROGRAM STRUCTURE

D. ERROR DETECTION SIMULATION ON THESE STRUCTURES

The structures which were converted into directed graphs for Module One were screened to determine their suitability for error detection simulation. It would have been desirable to select a random sample of the structures. However, it was necessary to choose structures which would not require excessive amounts of memory space and CPU time during the simulation. In addition, the structures were to have at least two or more paths. In the case of Module Two it was feasible to use a random sample because a high percentage of the structures fell within the memory space and the CPU time limitations of the model.

The input data for the simulation were taken from the actual programs, including the number of source statements for every arc. The recorded number of errors per module was used to calculate the mean number of instructions between errors, which is used for seeding errors in the simulation model. Seeding the errors was done randomly by the simulation program. However, it was provided that no errors were seeded at arcs containing zero instructions (control arcs).

The simulation was run with one input, 100 replications and 100 repetitions (reseedings), and the average number of errors found by one input was obtained. Although some of the structures were small, and a higher number of repetitions and replications could have been run, the same simulation parameters were used for each structure in order to obtain comparable results.

E. RESULTS OF THE ANALYSIS

From the average of errors found by one input in each procedure the average percentage of errors found against the errors expected within the procedure was obtained. These results were plotted against various complexity measures, e.g. the number of paths. Although the results varied somewhat between the modules, it was possible to establish relationships between structural properties and error detection capabilities.

The differences in results between modules can be traced to several factors:

1. Different sample sizes:
   From Module One 32 procedures were used, 16 procedures were randomly selected from Module Two.

2. The different size of the modules:
   Module One had 97, and Module Two had 155 procedures.

3. Differences in program design and programming style:
   Module Two was modularized to a much larger extent than Module One. It was hard to find a sufficient number of paths within randomly selected procedures of Module Two.

4. Different number of reported errors:
   Although Module Two was 1.6 times larger than Module One, it had only about two-thirds the number of errors.

The following diagrams show the percentage of average errors found against the expected number of errors for the structures of both modules.

1. Module One

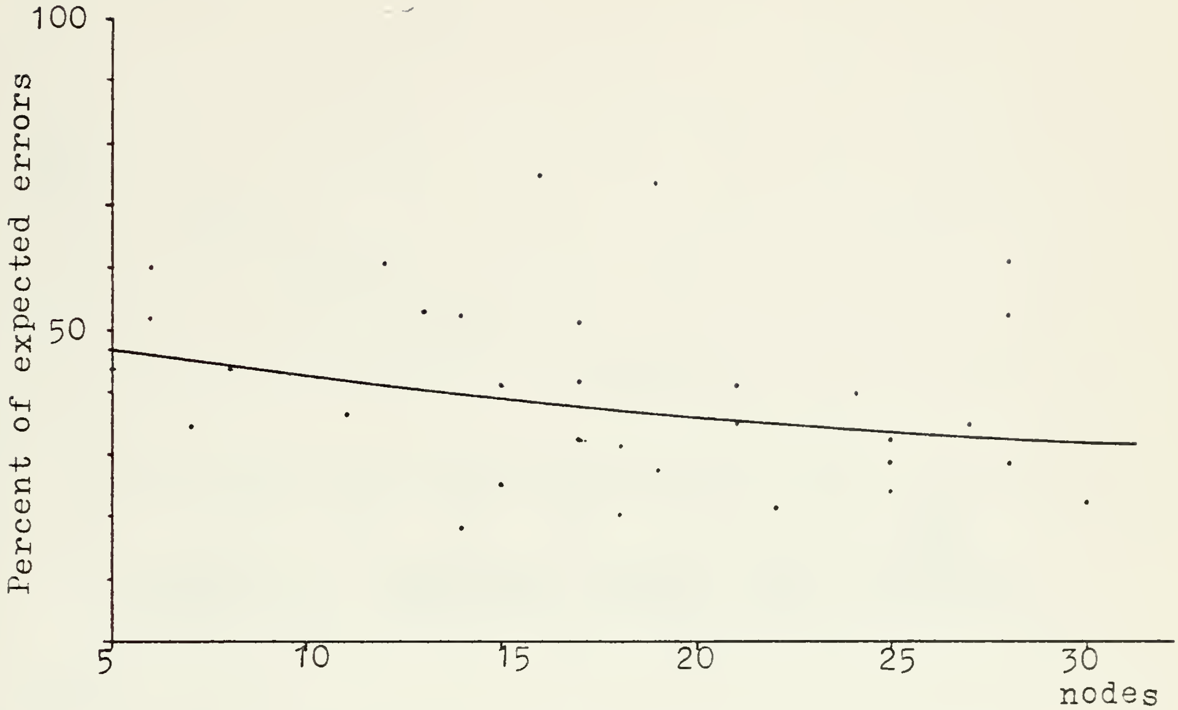


Figure 3 - PERCENTAGE ERRORS FOUND VS. NODES

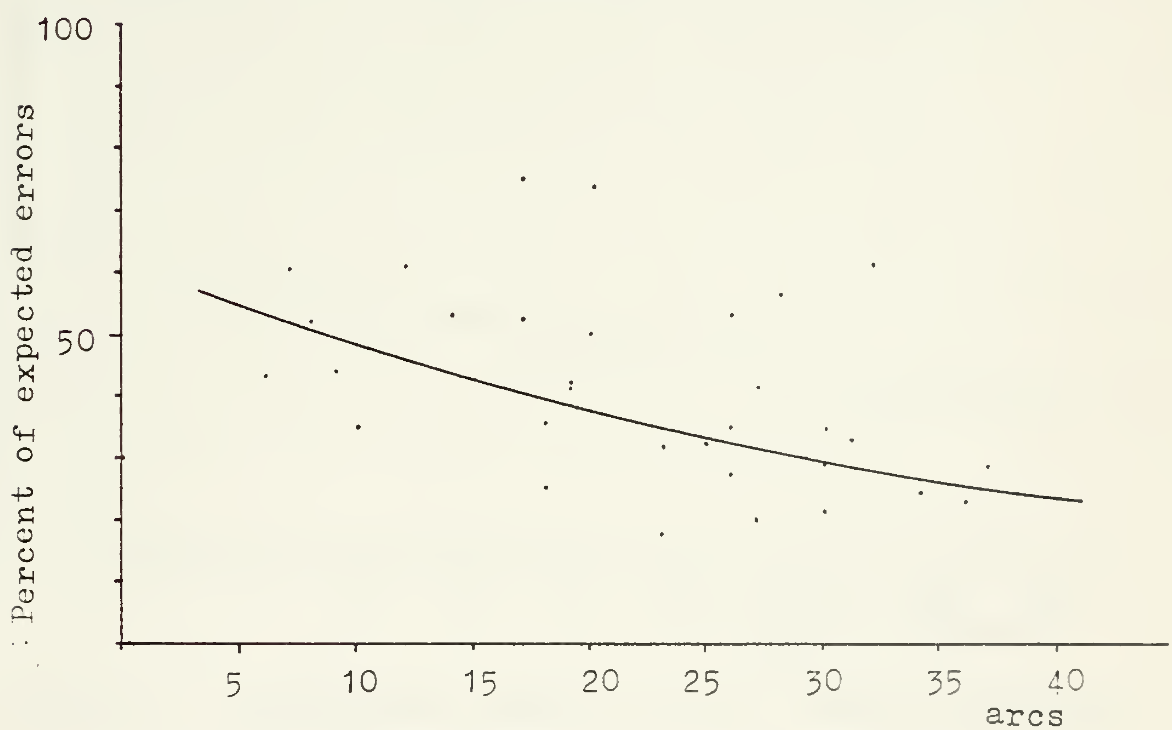


Figure 4 - PERCENTAGE ERORS FOUND VS. ARCS

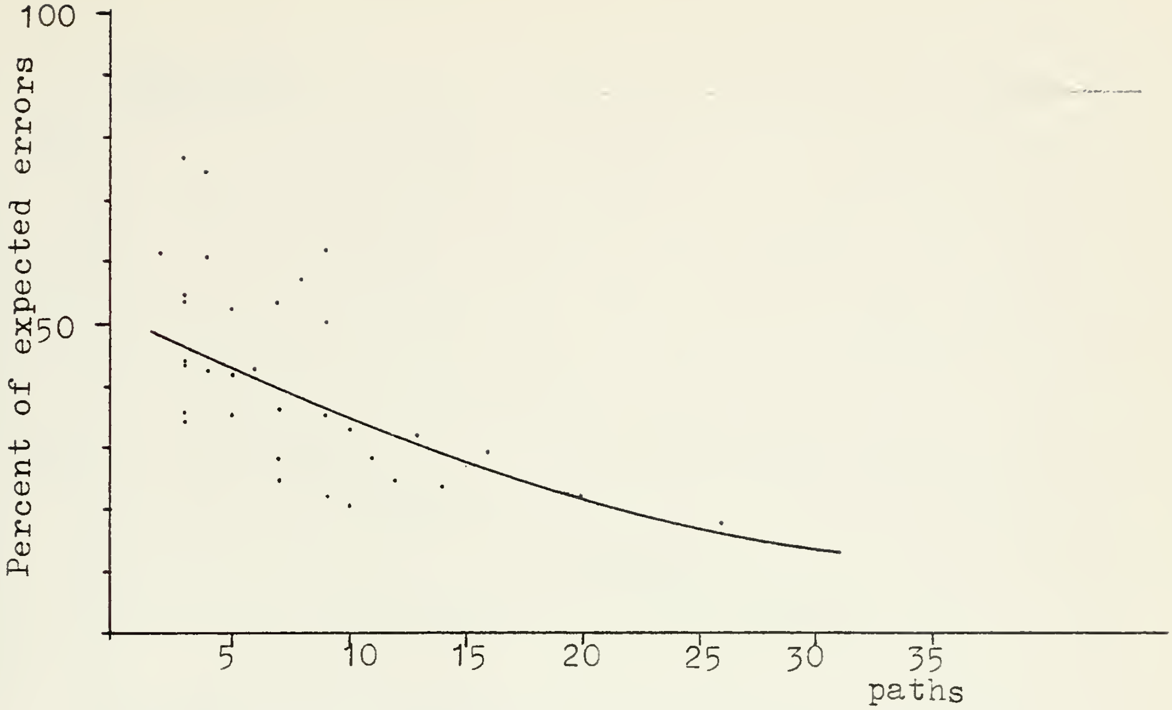


Figure 5 - PERCENTAGE ERRORS FOUND VS. PATHS

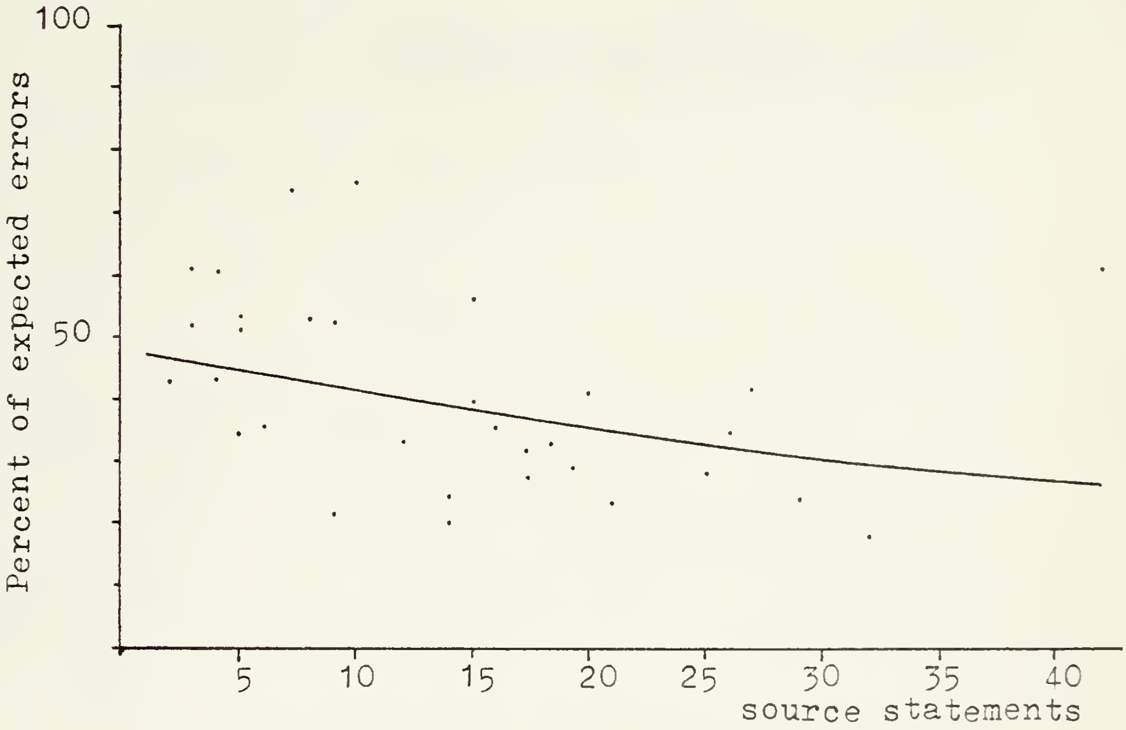


Figure 6 - PERCENTAGE ERRORS FOUND VS. SOURCE STATEMENTS

2. Module Two

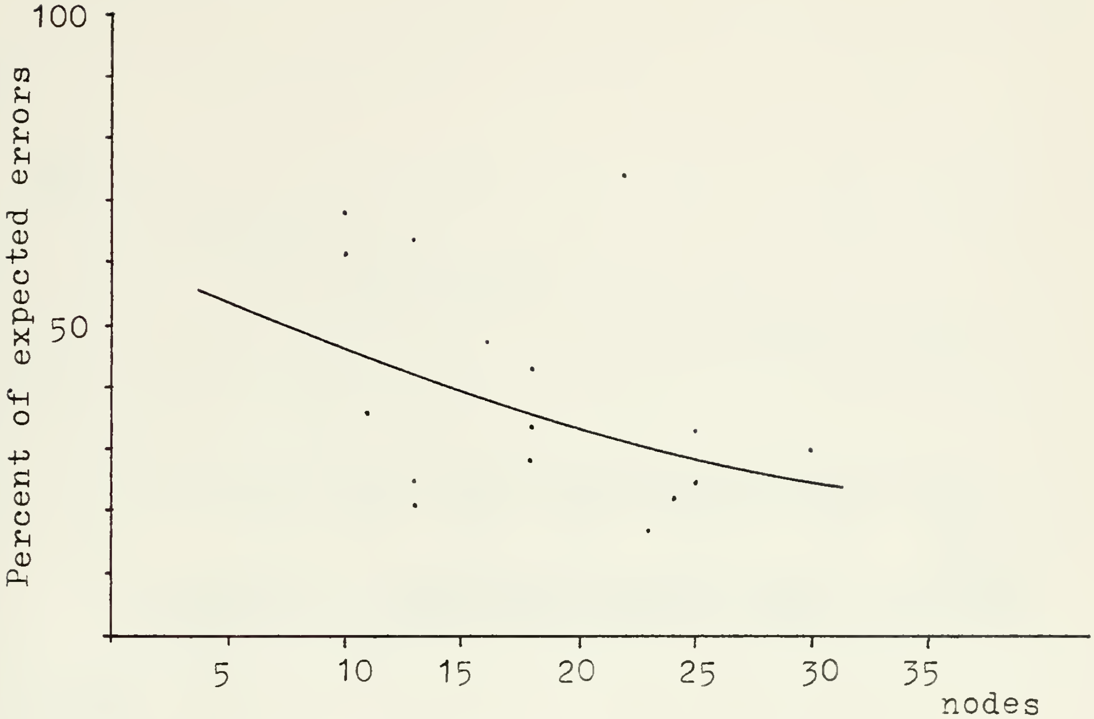


Figure 7 - PERCENTAGE ERRORS FOUND VS. NODES

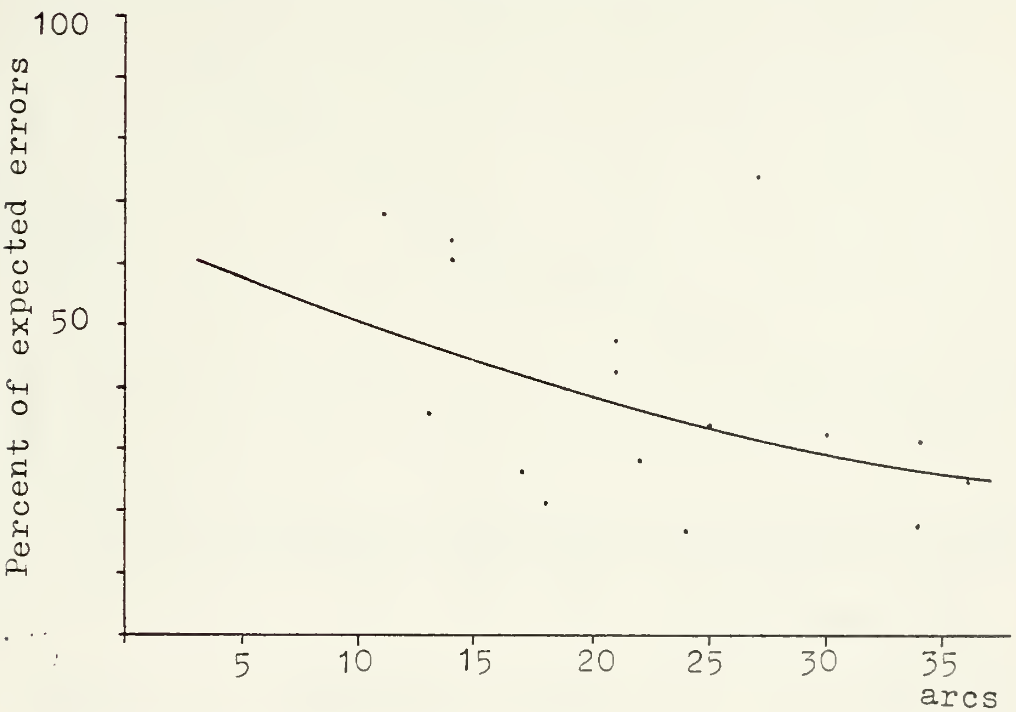


Figure 8 - PERCENTAGE ERRORS FOUND VS. ARCS

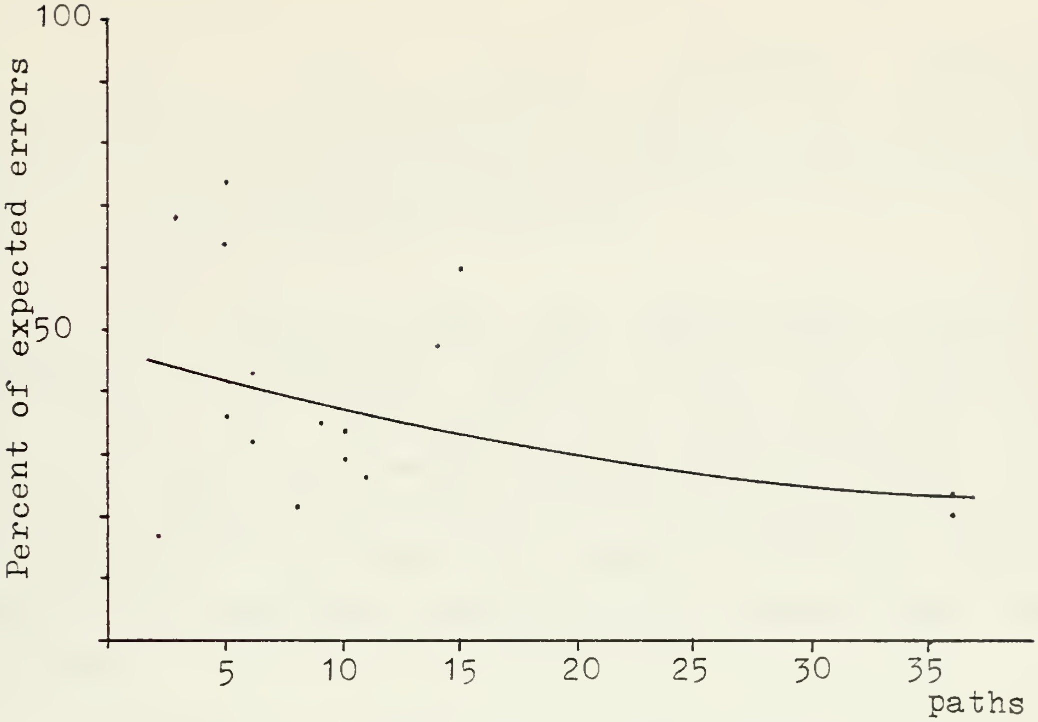


Figure 9 - PERCENTAGE ERRORS FOUND VS. PATHS

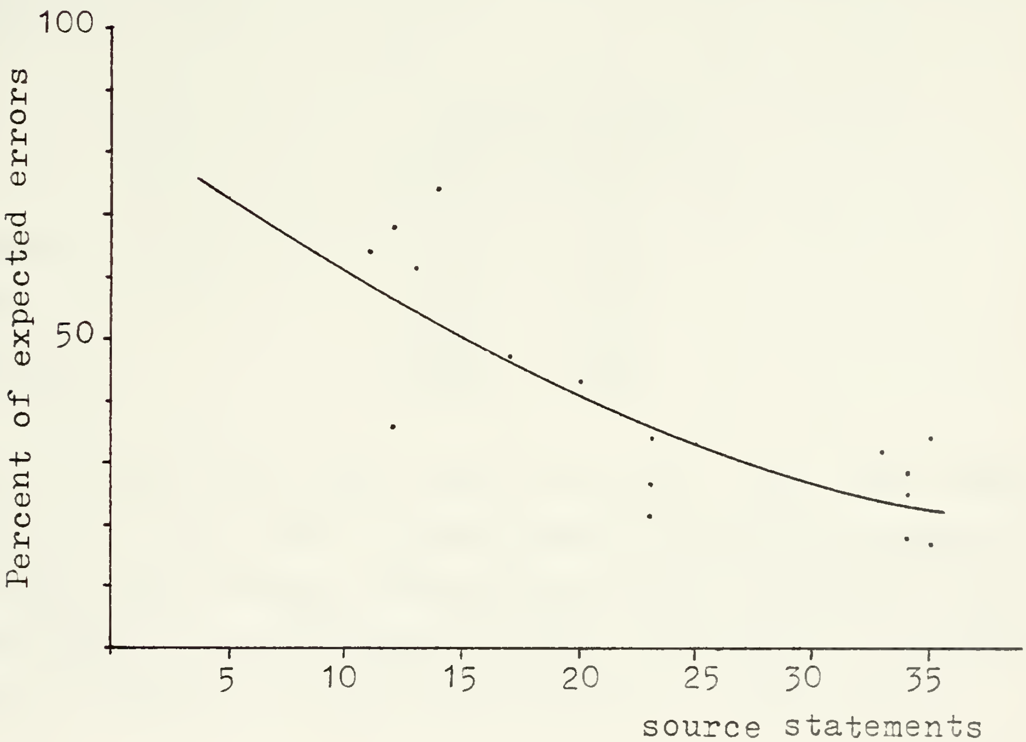


Figure 10 - PERCENTAGE ERRORS FOUND VS. SOURCE STATEMENTS

The curves shown represent exponential approximations to the datapoints according to the formula y=a*e**(-b*x) which was found to represent the relationship best. A Least Square fit was used.

All diagrams show some relationship between error detection and complexity. Module One with its larger sample size shows this relationship more than Module Two for the number of paths. This seems logical because a large number of paths reduces the ability to detect errors in a program. It appears that the number of paths could be used as a measure of program complexity for design and testing purposes.

In order to rank the approximations, a squared error factor has been computed for every complexity measure as follows:

	Error Factor	
	Mod. 1	Mod. 2
Nodes	7337	4430
Arcs	6841	3933
Paths	4995	4666
S.stmts.	6575	1808

This computation shows that for Module One the number of paths is the best approximated complexity measure by the method used. Another interesting aspect found was the well approximated relationship between percentage of errors found and the number of source statements in module Two.

VII. USE OF THE RESULTS

A. AIDS FOR SOFTWARE DEVELOPMENT

This method of program analysis provides the software manager with information for selecting structures easily in the design process. He can choose the least complex structure which will satisfy project requirements. Furthermore, after a project has been coded and is due for testing, he can make realistic assessments concerning the effort which will be needed for program testing by considering factors such as

1. expected complexity of the project
2. choice of the programming techniques used
3. organization and experience of the programming team
4. available manpower and computer time for testing purposes.

B. FUTURE WORK

The analysis done on the NIDS programs and the results obtained for the measurement of program complexity represents a modest contribution to the field of software engineering. But being far from complete or exhaustive the following steps should be taken in order to obtain additional validation of the analysis process.

1. Further evaluation of NTDS-Modules

Additional NTDS modules should be evaluated in order to obtain larger sample sizes. It is realized that the evaluation process for the important modules is very time consuming. However, the more important modules are used more frequently and will, in most cases, have a longer error history, which will provide valuable data for comparison with simulation results.

2. Evaluation of structured programs

It would be of interest in this respect to compare the evaluation of the NTDS-procedures with procedures that perform the same functions but are rewritten and converted into a structured programmed form. It is expected that the structured programs would perform better with respect to error detection.

VIII. SUMMARY AND CONCLUSIONS

A method to define and analyze program structures has been presented. All measurements obtained were based on the description of the program structure in the form of a directed graph and the use of the error detection simulation model. This method has been used to analyze the procedures from two NTDS modules. It was beyond the scope of this effort to obtain comparative results between this experiment and the actual error history of the programs. However, it was possible to obtain an initial quantitative assessment of measures of complexity.

By using this method to check program structures in the design phases it should be possible to produce programs with structures that are less complex and therefore easier and more economical to test and maintain. Also the method could be used during the test phase as a means of assigning test resources.

IX. ACKNOWLEDGEMENTS

The critical help and support by my thesis advisor, Prof. N.F. Schneidewind, the contribution of steady improvements on the simulation program by Prof. G.T. Howard, and the patience of my wife Elisabeth during this research period are gratefully acknowledged.

APPENDIX A

ERROR DETECTION SIMULATION PROGRAM

The program listed on the following pages shows the Software Error Detection Simulation Program as used in the analysis of the NTDS procedures in the version current in May 1976. Although carefully tested the program should not be regarded as a final release.

```
      COMMON MOUT
      COMMON MINST
      COMMON/ALLL/N, NODES(30,30)
      COMMON/ARRAY/X(30,30), NUMPTS(30), ASTOUT(30), AVCHNG
      COMMON/ERROR/ISEED(30,30), ITER(30,30), MEANER, INST, MEANIT, IZ
      COMMON/CUT/NSEED, INPUT, SVSEED(30,30)
      COMMON/GEN/MEANLN, IX
      COMMON/NEW/IW
      COMMON/GRAF/INREAD
      DIMENSION SFIND(50), SFOUND(50)
      DIMENSION SVAVE(50), SVVAR(50)
      DIMENSION TTIME(50), TREP(50), TLINK(50), TKOUNT(50), TINPUT(50)
      DIMENSION PSEED(50), F(50), FNA(50), FLA(50)
      DIMENSION TSEED(50), TINST(50), PTEST(50)
      DIMENSION IFIND(50),IFOUND(50)
      DIMENSION SVSQR(50)
      DIMENSION NTRAV(30,30)
      DIMENSION CUMSQR(50)
      DIMENSION NAME(3)
      DIMENSION MODULE(3)
      INTEGER*4 SVSEED
      REAL*4 MTTR
  901 FORMAT ('1  RESULTS FOR ROUTINE NR:',I3,'  MODULE:',3A4,'  NAME:',
     C3A4//5X,'NOTE: THIS IS THE VERSION OF 18 MAY 76 WITH CORRECTED VAR
     CIANCE CALCULATION'///)
  902 FORMAT (I3,3A4)
  903 FORMAT (3A4)
  108 FORMAT (I10)
  140 FORMAT (16I5)
  152 FORMAT ('1',13X,'THIS RUN WILL USE MINPUT INPUTS.')
  156 FORMAT ('0', 13X, 'THE NUMBER OF INSTRUCTIONS IN EACH ARC HAS BEEN
     1 READ IN AND IS NOT RANDOM.')
  180 FORMAT ('0', 13X, 'ADJACENCY MATRIX'//7X, 30(I4))
  203 FORMAT ('0', 13X, 'NOTE: NONEXISTENCE OF AN ARC IS INDICATED BY A
     1 ZERO'//)
  205 FORMAT (7(I5, 5X))
  250 FORMAT(' ', 13X, I4, ' SEEDED ERRORS REMAINING'//13X, 'MATRIX OF
     1SEEDED ERRORS REMAINING'//7X, 30(I4))
  255 FORMAT(' ', 7X, 30(A4))
  260 FORMAT('0',      I5, ' *', 30(I4))
  265 FORMAT('1', 11X, I3, ' INPUTS'//13X, I4, ' INSTRUCTIONS'//13X,
     1I3, ' ERRORS SEEDED'//13X, I3, ' RESIDUAL ERRORS'//13X, F6.2,
     2 ' PERCENT RESIDUAL ERRORS'//13X, F6.2, ' PERCENT OF ARCS TESTED'
     3 //)
C
C
      READ (5,902) NR, NAME
      READ (5,903) MODULE
```

```
      WRITE (6,901) NR, MODULE, NAME
      READ (5,100) MINPUT ,NUMOUT, NREPET,  MEANLN, MEANER, N
  305 FORMAT('0',13X,'THE PATH SEED IZ IS NOW',I11)
      WRITE (6,99)
      WRITE (6,101) MINPUT, NUMOUT, NREPET, MEANLN, MEANER, N
   99 FORMAT (' INPUTS  REPLICATIONS  REPETITIONS  MEANLN  MEANER  N')
  100 FORMAT (6I5)
  101 FORMAT (I6,2I13,2I8,I7)
C     READ IN THE ADJACENCY MATRIX (NODES)
      DO 30 I=1,N
      NUMPTS(I)=I
      DO 30 J=1,N
      X(I,J)=0.0
      NTRAV(I,J)=0
      ISEED(I,J) = 0
   30 NODES(I,J)=0
C
C  DATA INPUT    THE GRAPH
      NPI=N+1
      DO 20 I=1,NPI
      READ (5,140) IPN,IPC,(IFIND(J),J=1,IPC)
      WRITE(6,140) IPN,IPC,(IFIND(J),J=1,IPC)
      IF (IPN.EQ.99) GO TO 22
      DO 21 J=1,IPC
      ICC=IFIND(J)
      NODES(IPN,ICC)=1
   21 CONTINUE
   20 CONTINUE
   22 CONTINUE
C
C  DATA INPUT   THE ARC LENGTHS
      ISW1=0
      DO 24 I=1,NPI
      READ(5,141) IPN, IPC, (IFIND(J),SFIND(J), J=1,IPC)
      WRITE(6,141) IPN, IPC, (IFIND(J),SFIND(J), J=1,IPC)
  141 FORMAT (2I5,7(I5,F5.0))
      IF((I.EQ.1).AND.(IPN.NE.99)) ISW1=1
      IF(IPN.EQ.99) GO TO 23
      DO 24 J=1,IPC
      ICC=IFIND(J)
      X(IPN,ICC)=SFIND(J)
   24 CONTINUE
   23 CONTINUE
C
      DO 1000 I=1,N
      DO 1000 J=1,N
      ISEED(I,J)=0
      SNSEED(I,J)=0
```

```
 1000 CONTINUE
C   DATA INPUT    PLANT THE ERRORS
      ISW2=0
      DO 25 I=1,NPI
      READ (5,140)IPN,IPC,  (IFIND(J),IFOUND(J), J=1,IPC)
      WRITE(6,140)IPN,IPC,  (IFIND(J),IFOUND(J), J=1,IPC)
      IF ((I.EQ.1).AND.(IPN.NE.99)) ISW2=1
      IF (IPN.EQ.99) GO TO 26
      DO 25 J=1,IPC
      ICC= IFIND(J)
      ISEED(IPN,ICC)=IFOUND(J)
      SVSEED(IPN,ICC)=IFOUND(J)
   25 CONTINUE
   26 CONTINUE
      READ (5,500) MOUT
      NIX=NUMCUT*NREPET
      IF(NIX.GT.25) MOUT=0
  500 FORMAT (I5)
      CALL CVFLOW
      IF (ISW1.EQ.1) GO TO 11
      DO 10 I=1,N
      DO 10 J=1,N
      IF(NODES(I,J).EQ.0) GO TO 9
      X(I,J)=10.
      GO TO 10
    9 X(I,J)=0.0
   10 CONTINUE
      WRITE (6,144)
  144 FORMAT (' ALL ARC LENGTHS SET BY PROGRAM TO 10')
   11 CONTINUE
      IX = 71286223
      IW = IX
      IZ=1
      IREPET=0
      NTEST=0
      MTEST=0
      DO 1111 I=1,MINPUT
 1111 CUMSQR(I)=0.
C
C   BEGIN THE RESEEDING LOOP
C
   77 CONTINUE
      IREPET=IREPET+1
      IF((ISW2.EQ.1).AND.(NREPET.NE.1)) GO TO 776
      GO TO 774
  776 CONTINUE
      NREPET=1
      WRITE (6,143)
```

```
  143 FORMAT(' PROGRAM SETS NREPET TO 1 WHEN ERRORS ARE PLANTED')
  774 IF(ISW2.EQ.1) GO TO 775
C       SEED THE PROGRAM WITH ERRORS
      CALL SEED
  775 CONTINUE
C
C  BEGIN THE REPLICATION LOOP WITHIN A GIVEN SEEDING
      IREP=0
   78 CONTINUE
      IREP=IREP+1
C
C  BEGIN THE LOOP WHICH CONTROLS THE NUMBER OF INPUTS WITHIN A REPLICATION
      IRUN=1
  785 CONTINUE
      NFIND=0
      NODE = 1
      IF (MOUT.EQ.0) GO TO 800
      WRITE(6,402) IREPET
  402 FORMAT(' SEEDING NUMBER', I5)
      WRITE(6,400) IREP
  400 FORMAT(' REPLICATION=',I5)
      WRITE(6,401) IRUN
  401 FORMAT(' INPUT=',I5)
      WRITE (6,227) NODE
  227 FORMAT (' ',I3)
  800 CONTINUE
C     NUMSUC IS THE NUMBER OF SUCCESSORS
   79 NUMSUC = 0
      DO 80 J=1,N
      IF (NODES(NODE,J).EQ.1) NUMSUC = NUMSUC + 1
   80 CONTINUE
      IF (NUMSUC.EQ.0) GO TO 96
      IF (NUMSUC.NE.1) GO TO 82
      K = 1
      GO TO 83
C     THE SUCCESSORS ARE CHOSEN RANDOMLY WITH A UNIFORM DISTRIBUTION
   82 CALL RANDCM(IZ, U, 1)
      K = 1 + NUMSUC*U
   83 KK = 0
      DO 85 J = 1,N
      MMM=J
      IF (NODES(NODE,J).EQ.1) KK = KK + 1
      IF (KK.EQ.K) GO TO 86
   85 CONTINUE
   86 L=MMM
      IF (MOUT.EQ.0) GO TO 801
   89 WRITE (6,230) L
  801 CONTINUE
```

```
  230 FORMAT (' ',I3)
      CONTINUE
C     COUNT TRAVERSALS
      NTRAV(NODE,L)=NTRAV(NODE,L)+1
C     IS THERE AN ERROR IN THIS PATH
      IF(ISEED(NODE,L).EQ.0) GO TO 95
C     ERROR FOUND
      IF (MOUT.EQ.0) GO TO 802
      WRITE (6,245) ISEED(NODE,L)
  802 CONTINUE
  245 FORMAT (' NUMBER ERRORS FOUND IN PREVIOUS ARC=',I5)
      NFIND=NFIND+ISEED(NODE,L)
   90 ISEED(NODE,L) =0
   95 NODE = L
      GO TO 79
C
   96 CONTINUE
      IFIND(IRUN)=NFIND
      SFIND(IRUN)=NFIND*NFIND
      IRUN=IRUN+1
C     MINPUT IS THE MAXIMUM NUMBER OF INPUT PATHS TO BE CHECKED
      IF (IRUN.LE.MINPUT) GO TO 785
      DO 556 IRAN=1,MINPUT
      IF(IREP.EQ.1) GO TO 555
      IFOUND(IRAN)=IFOUND(IRAN)+IFIND(IRAN)
      SFOUND(IRAN)=SFOUND(IRAN)+SFIND(IRAN)
      CUMSQR(IRAN)=CUMSQR(IRAN)+SFIND(IRAN)
      GO TO 556
  555 IFOUND(IRAN)=IFIND(IRAN)
      SFOUND(IRAN)=SFIND(IRAN)
      CUMSQR(IRAN)=CUMSQR(IRAN)+SFIND(IRAN)
  556 CONTINUE
  112 CONTINUE
      IF(IREP.GE.NUMOUT) GO TO 118
      DO 113  I = 1,N
      DO 113  J = 1,N
  113 ISEED(I,J) = SVSEED(I,J)
      GO TO 78
C
  118 CONTINUE
      DO 666 NNN=1,MINPUT
      XXX=IFOUND(NNN)
      YYY=IREP
      AVE=XXX/YYY
      IF(IREP.EQ.1) GO TO 124
      DIV=IREP-1
      VAR=(((SFOUND(NNN)-(DIV+1.)*AVE*AVE))/DIV)
      GO TO 123
```

```
  124 CONTINUE
      IF (MOUT.EQ.0) GO TO 803
      WRITE (6,900)
  803 CONTINUE
  900 FORMAT (' STD DEV NOT COMPUTED')
      VAR=11108889.
  123 CONTINUE
      SD=VAR**.5
      IF (MOUT.EQ.0) GO TO 804
      WRITE(6,667) NNN
      WRITE(6,668) AVE
      WRITE(6,669) SD
  804 CONTINUE
  667 FORMAT(' INPUT NUMBER=',I5)
  668 FORMAT(' AVE NUMBER ERRORS FOUND=',F10.2)
  669 FORMAT(' STD DEV =',F5.2)
      IF(IREPET.EQ.1) GO TO 116
      SVAVE(NNN)=SVAVE(NNN)+AVE
      SVVAR(NNN)=SVVAR(NNN)+VAR
      SVSQR(NNN)=SVSQR(NNN)+AVE*AVE
      GO TO 119
  116 SVAVE(NNN)=AVE
      SVVAR(NNN)=VAR
      SVSQR(NNN)=AVE*AVE
  119 CONTINUE
  666 CONTINUE
  199 IF (IREPET.LT.NREPET) GO TO 77
C
      IF (MOUT.EQ.0) GO TO 805
      WRITE (6,305) IZ
  805 CONTINUE
      DO 120 III=1, MINPUT
      Z=NREPET
      TAVE=SVAVE(III)/Z
      Y=NUMOUT
      TVAROK=(CUMSQR(III)-Y*Z*TAVE*TAVE)/(Y*Z-1.)
      TSDOK=TVAROK**.5
      TSD=SVVAR(III)/Z
      TSD=TSD**.5
      WRITE (6,777) III
      WRITE (6,778) TAVE
      WRITE (6,779) TSDOK
      IF (NREPET.EQ.1) GO TO 7766
      IF (NUMOUT.NE.1)GO TO 7766
      VAR1=(SVSQR(III)-(NREPET)*TAVE*TAVE)/(NREPET-1)
      SD1=VAR1**.5
      WRITE (6,780) SD1
 7766 CONTINUE
```

```
  777 FORMAT ('0 SUMMARY FOR INPUT', I4)
  778 FORMAT (' AVE ERRORS FOUND=',F10.4)
  779 FORMAT (' STANDARD DEVIATION=',F5.2)
  780 FORMAT (' STD DEV OVER SEEDING WITH ONE REPLIC=',F10.4)
  120 CONTINUE
  121 CONTINUE
  125 CONTINUE
      WRITE (6,132)
      DO 131 I=1,N
      DO 131 J=1,N
      IF(NODES(I,J).EQ.0) GO TO 131
      WRITE(6,133) I,J,NTRAV(I,J)
  131 CONTINUE
  132 FORMAT ('1',4X,'I',4X,'J',4X,'NBR OF TRAVERSALS')
  133 FORMAT (2I5,I10)
  127 STOP
C
      END
C
C
C
      SUBROUTINE SEED
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                     C
C     SEED THE GRAPH WITH ERRORS                                      C
C                                                                     C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      COMMON MOUT
      COMMON NINST
      COMMON/ALLL/N, NODES(30,30)
      COMMON/ARRAY/X(30,30), NUMPTS(30), ASTOUT(30), AVCHNG
      COMMON/ERROR/ISEED(30,30), ITER(30,30), MEANER, INST, MEANIT, IZ
      COMMON/OUT/NSEED, INPUT, SVSEED(30,30)
      COMMON/GEN/MEANLN, IX
      INTEGER*4 CHANGE, AVCHNG, SVSEED
      DIMENSION ER(50),XIER(50)
  205 FORMAT ('0', 13X, 'SEED ERRORS AT INSTRUCTION INTERVALS'// 10X,
     120F5.1)
  210 FORMAT ('1'// 13X, 'TOTAL NUMBER OF INSTRUCTIONS IS', I5)
  220 FORMAT ('0'// 13X, 'TOTAL NUMBER OF ERRORS SEEDED IS',I4// 13X,
     1'THE ERROR MATRIX'//7X, 30(I4))
  225 FORMAT ('0',      I5, ' *', 30(I4))
C     NSEED  IS THE NUMBER OF ERRORS SEEDED
      NSEED = 0
      XNUMER=0.
      JUMP=0
      INTERV=0
      XINST=0.
```

```
      XMEAN=MEANER
      DO 62 I=1,N
      DO 62 J=1,N
   62 ISEED(I,J)=0
      DO 70 I = 1,N
      DO 65 J = 1,N
C     CALCULATE THE NUMBER OF INSTRUCTIONS
      XINST=XINST+X(I,J)
      NINST=XINST
      IF (NODES(I,J).EQ.0)  GO TO 65
      IF (JUMP.EQ.1) GO TO 635
   63 CALL EXPCN (IX,ER1,1)
      XER1=XMEAN*ER1
      INTERV=INTERV+1
CCC   XIER(INTERV)=XER1
      XNUMER=XNUMER+XER1
  635 JUMP=0
      IF (XINST.LT.XNUMER) JUMP=1
      IF (XINST.LT.XNUMER) GO TO 64
      ISEED(I,J)=ISEED(I,J)+1
      NSEED=NSEED+1
      GO TO 63
   64 CONTINUE
   65 CONTINUE
   70 CONTINUE
      DO 71 I = 1,N
      DO 71 J = 1,N
C     SVSEED IS THE SAVE COPY OF THE ORIGINAL SEEDED ERRORS
   71 SVSEED(I,J) = ISEED(I,J)
C     NSEED  IS THE NUMBER OF ERRORS SEEDED
      IF (MOUT.EQ.0) GO TO 806
      WRITE(6,210) NINST
      INTERV=INTERV-1
CCC   WRITE (6,205) (XIER(I),I=1,INTERV)
      WRITE (6,220) NSEED, (NUMPTS(I), I = 1,N)
      DO 75 I = 1,N
      WRITE (6,225) NUMPTS(I), (ISEED(I,J), J=1,N)
   75 CONTINUE
      WRITE (6,226) IX
  226 FORMAT ('0',13X,'THE ERROR SEED IX IS NOW',I11)
  806 CONTINUE
C
      RETURN
C
      END
```

APPENDIX B

LIST OF EVALUATED PROGRAM STRUCTURES

This list gives all the statistical data gathered from the conversion of the procedures of the NTDS modules into the form of directed graphs. The abbreviations read as follows:

PNR	Procedure number within the module
N	Number of nodes (including transient nodes)
A	Number of arcs (including transient arcs)
P	Number of paths
L	Number of loops
Ss	Number of source statements
Mi	Number of machine instructions
SA	Source stmts./arc
MA	Machine instr./arc

1. Module One

ENR	N	A	P	L	SS	MI	S/A	M/A
1	2	1	1	0	2	1	2.00	1.00
2	14	23	22	0	37	134	1.61	5.83
3	4	3	2	0	5	15	1.67	5.00
4	3	2	1	0	18	18	9.00	9.00
5	4	4	2	0	4	17	1.00	4.25
6	34	45	64	0	60	302	1.33	6.71
7	4	4	2	0	8	17	2.00	4.25
8	13	14	3	0	10	25	0.71	1.79
9	4	4	2	0	7	15	1.75	3.75
10	5	5	2	0	4	23	0.80	4.60
11	6	8	5	0	8	15	1.00	1.88
12	2	1	1	0	4	5	4.00	5.00
13	2	1	1	0	5	6	5.00	6.00
14	6	7	4	0	9	24	1.29	3.43
15	4	4	2	0	6	25	1.50	6.25
16	14	13	1	0	13	22	1.00	1.69
17	14	13	1	0	13	23	1.00	1.77
18	21	23	2	0	21	35	0.91	1.52
19	19	26	7	0	22	45	1.16	2.37
20	45	66	11	0	82	134	1.24	2.03
21	35	49	88	0	33	100	0.67	2.04
22	25	30	11	0	30	53	1.00	1.77
23	7	7	2	0	6	8	0.86	1.14
24	6	5	1	0	8	17	1.60	3.40
25	12	12	2	1	8	26	0.67	2.17
26	6	5	1	0	5	6	1.00	1.20
27	8	8	2	1	10	20	1.25	2.50
28	17	19	4	0	32	99	1.68	5.21

ENR	N	A	P	L	SS	MI	S/A	M/A
29	28	32	5	0	47	150	1.47	4.69
30	7	10	5	0	10	43	1.00	4.30
31	4	4	2	0	4	12	1.00	3.00
32	4	4	2	1	6	13	1.50	3.25
33	10	9	1	0	7	7	0.78	0.78
34	16	17	3	0	15	23	0.88	1.35
35	14	17	3	0	14	20	0.82	1.18
36	21	26	3	0	31	57	1.19	2.19
37	54	64	13	0	56	111	3.88	1.73
38	8	10	8	3	19	40	1.90	4.00
39	17	25	10	0	17	59	0.68	2.36
40	83	120	3704	10	78	271	0.65	2.26
41	33	38	7	0	31	63	0.82	1.66
42	12	14	2	0	11	17	0.79	1.21
43	13	14	83	0	8	12	0.57	0.86
44	27	30	7	0	21	38	0.70	1.27
45	12	12	2	0	8	13	0.67	1.08
46	9	9	2	0	10	25	1.11	2.78
47	19	20	4	0	12	22	0.60	1.10
48	23	26	7	0	13	34	0.50	1.31
49	15	18	7	0	19	47	1.06	2.61
50	2	1	1	0	7	38	7.00	38.0
51	9	9	2	0	11	36	1.22	4.00
52	125	150	1645	0	102	260	0.68	1.73
53	11	18	9	0	11	33	0.61	1.83
54	34	45	5	0	63	85	1.40	1.89
55	6	5	1	0	7	11	1.40	1.89
56	46	58	13	0	51	86	0.88	1.48

ENR	N	A	P	L	SS	MI	S/A	M/A
57	30	36	14	0	26	59	0.72	1.64
58	40	60	216	0	49	117	0.82	1.95
59	11	12	3	0	9	15	0.75	1.25
63	28	37	16	0	24	52	0.65	1.41
61	2	1	1	0	3	5	3.00	5.00
62	43	62	24	1	50	96	0.81	1.55
63	89	140	451	7	95	214	0.68	1.53
64	4	4	2	0	7	14	1.75	3.50
65	47	56	773	3	44	129	0.79	2.30
66	12	12	2	1	10	16	0.67	1.33
67	26	27	1	0	25	44	0.93	1.63
68	8	8	2	1	8	30	1.00	3.75
69	49	61	12	0	53	90	0.87	1.48
70	6	7	4	0	8	22	1.14	3.14
71	6	7	4	1	9	23	1.29	3.29
72	7	7	2	0	7	16	1.00	2.29
73	8	8	2	0	7	13	0.88	1.63
74	5	6	3	0	9	19	1.50	3.17
75	24	28	8	0	20	47	0.71	1.68
76	15	19	8	0	20	45	1.05	2.37
77	17	20	9	0	10	37	0.50	1.85
78	10	9	1	0	5	7	0.56	0.78
79	25	31	3	0	23	30	0.74	0.97
80	44	57	11	0	55	127	3.96	2.23
81	8	9	3	0	7	16	0.78	1.78
82	6	5	1	0	8	18	1.60	3.60
83	13	14	3	0	8	20	0.57	1.43
84	91	120	25	0	93	191	0.78	1.59

BNR	N	A	P	L	SS	MI	S/A	M/A
85	33	43	219	0	32	93	0.74	2.16
86	18	23	13	0	22	56	0.96	2.43
87	21	22	6	0	25	81	0.93	1.37
89	51	65	14	0	54	107	0.83	1.65
90	7	10	5	0	8	22	0.80	2.20
91	22	30	9	0	14	47	0.47	1.57
92	5	6	3	0	9	28	1.50	4.67
93	25	34	12	0	34	132	1.00	3.88
94	7	10	5	2	12	37	1.20	3.70
95	18	27	10	0	19	58	0.70	2.15
96	45	52	35	1	40	93	0.77	1.79
97	136	211	3972	0	99	162	0.47	0.77

2. Module Two

ENR	N	A	P	L	SS	MI	S/A	M/A
3	2	1	1	0	7	7	7.00	7.00
5	6	5	1	0	3	4	0.60	0.80
7	2	1	1	0	3	4	3.00	4.00
8	2	1	1	0	9	23	9.00	23.0
15	11	13	5	1	12	31	0.92	2.38
23	10	11	3	0	12	33	1.09	3.00
40	22	27	5	0	14	30	0.52	1.11
41	10	14	12	1	13	42	0.93	3.00
46	25	37	36	0	34	95	0.92	2.57
47	24	34	36	0	34	85	1.00	2.50
48	16	21	14	0	17	55	0.81	2.62
54	6	5	1	0	10	15	2.00	3.00
55	8	8	2	0	5	13	0.63	1.63
59	6	5	1	0	5	10	1.00	2.00
65	6	5	1	0	4	12	0.80	2.40
69	4	4	2	0	7	15	1.75	3.75
73	18	22	10	0	34	97	1.55	4.41
79	13	14	5	2	11	34	0.79	2.43
82	23	24	2	0	35	64	1.46	2.67
86	30	34	6	2	33	86	0.97	2.53
90	13	18	8	2	23	71	1.28	3.94
99	25	30	10	1	23	50	0.77	1.67
113	6	5	1	0	5	7	1.00	1.40
114	4	4	2	0	3	6	0.75	1.50
121	6	5	1	0	7	12	1.40	2.40
122	18	21	6	0	20	38	0.95	1.81
125	37	46	13	2	37	94	0.80	2.04
129	9	9	2	0	9	21	1.00	2.33
137	13	17	11	0	323	53	1.55	3.12
149	18	25	9	4	35	88	1.40	3.52

APPENDIX C

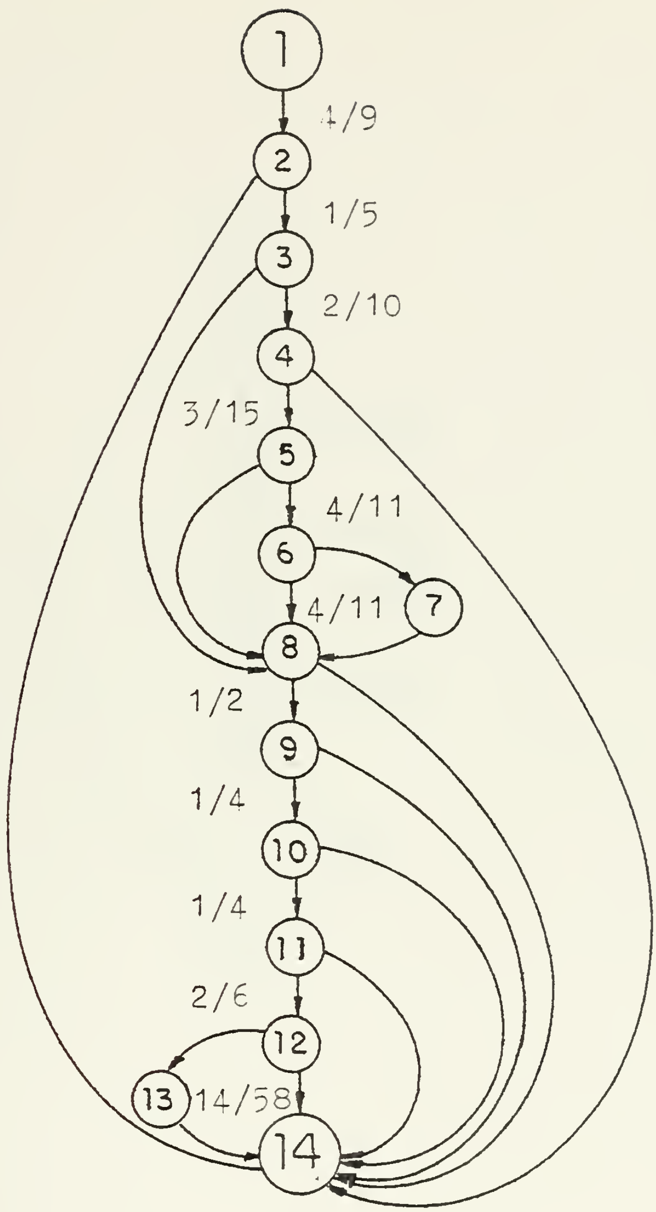
DIRECTED GRAPHS

On the following pages the structures of all the procedures are listed that were used as input data for the Error Detection Simulation Model. In addition to the complexity measures used also listed are the results obtained from the simulation, the average number of errors found with 1 input, 100 replications and 100 repetitions, and the percentage of expected errors detected.

Differently to the sample structure shown in Fig. 2, the number of statements is indicated in the following graphs only for arcs with nonzero instructions.

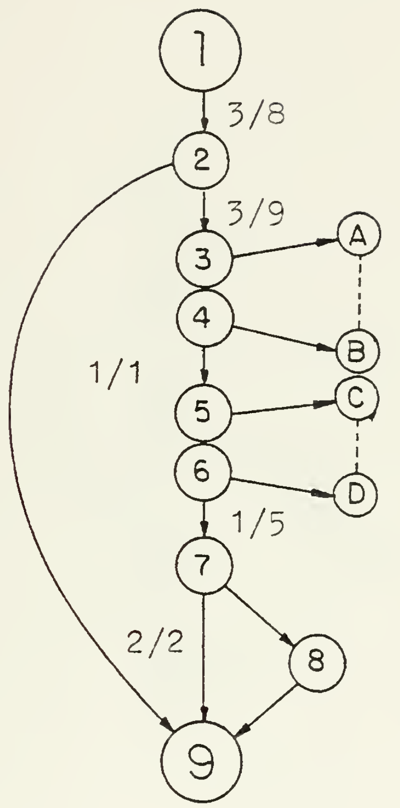
The count for the number of nodes and the number of arcs includes the transient nodes (designated by letters) and the transient arcs (dashed lines) because they must be included into the inputs for the Error Detection Simulation Program.

Module: 1 Procedure No.: 2



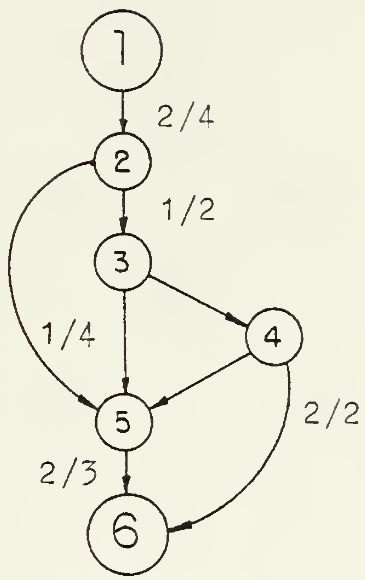
Number of nodes:	14
Number of arcs:	23
Number of paths:	26
Number of source stmts.:	37
Average error found:	0.3144
Percentage errors found:	17.84

Module: 1 Procedure No.: 8



Number of nodes:	13
Number of arcs:	14
Number of paths:	3
Number of source stmts.:	10
Average error found:	0.2523
Percentage errors found:	52.98

Module: 1 Procedure No.: 11



Number of nodes: 6

Number of arcs: 8

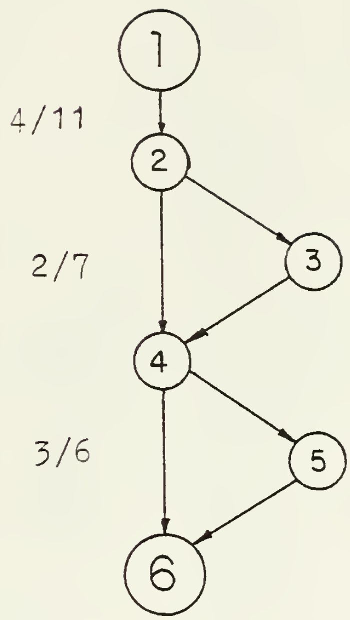
Number of paths: 5

Number of source stmts.: 8

Average error found: 0.1974

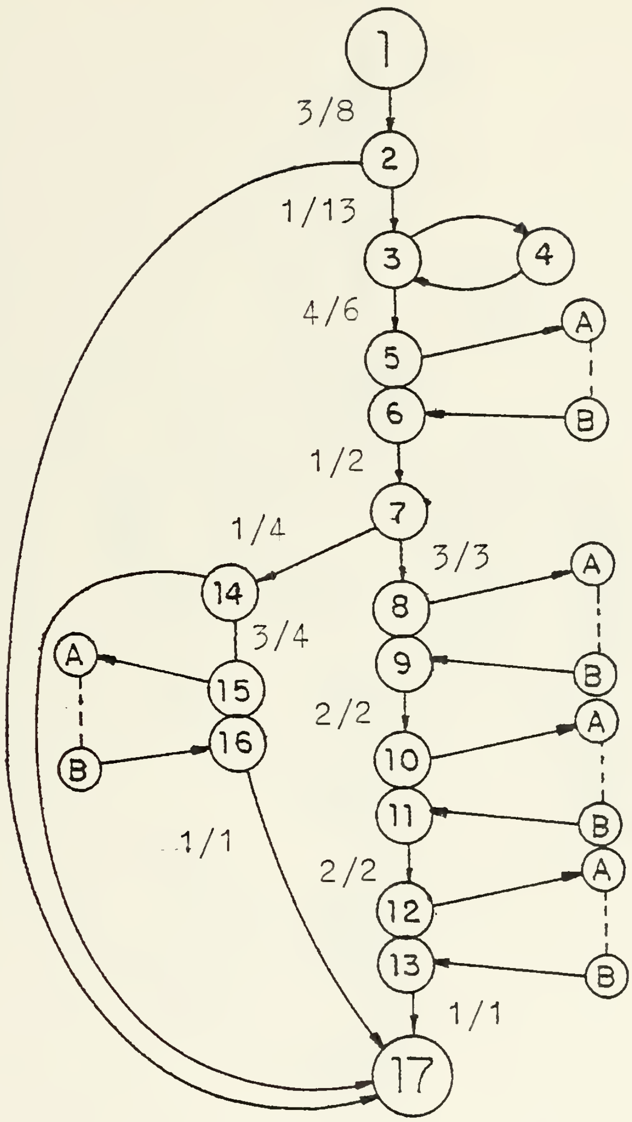
Percentage errors found: 51.82

Module: 1 Procedure No.: 14



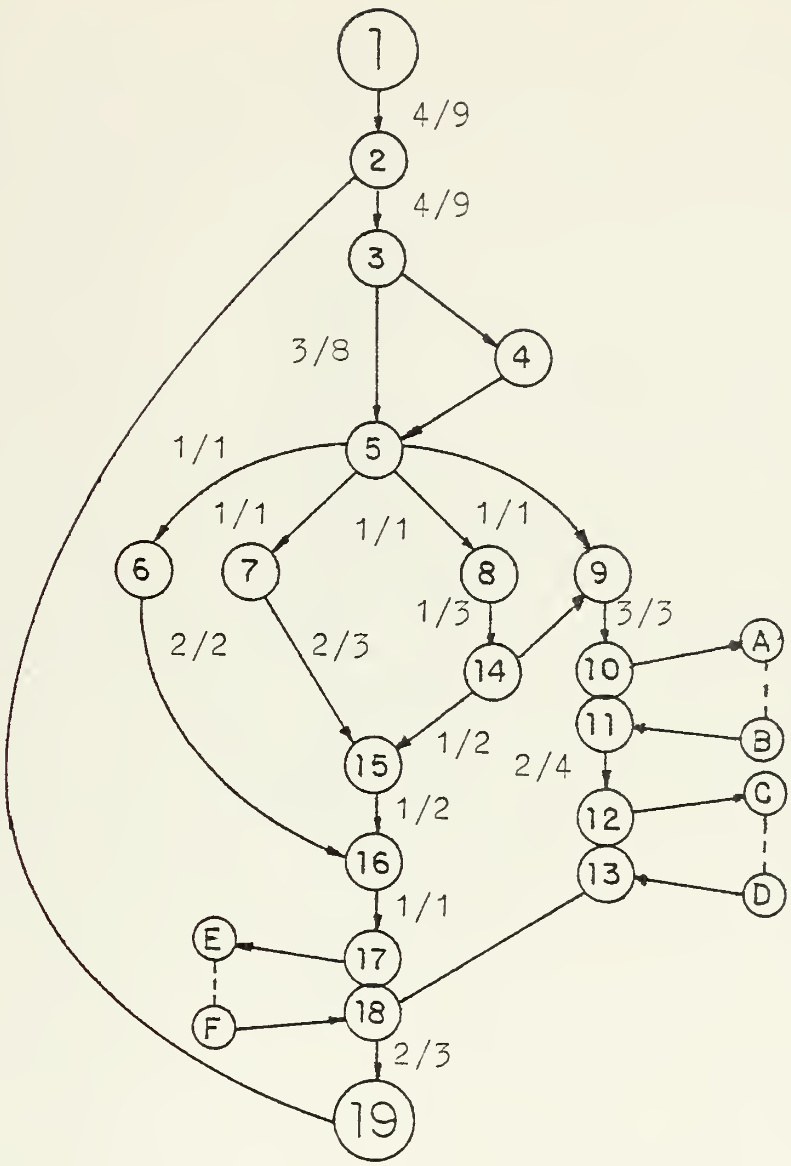
Number of nodes:	6
Number of arcs:	7
Number of paths:	4
Number of source stmts.:	9
Average error found:	0.2586
Percentage errors found:	60.34

Module: 1 Procedure No.: 19



Number of nodes:	19
Number of arcs:	26
Number of paths:	7
Number of source stmts.:	45
Average error found:	0.2885
Percentage errors found:	27.54

Module: 1 Procedure No.: 22



Number of nodes: 25

Number of arcs: 30

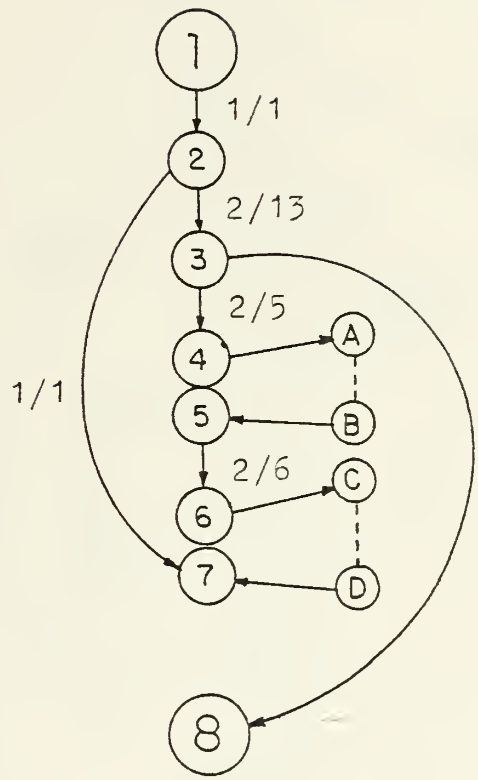
Number of paths: 11

Number of source stmts.: 30

Average error found: 0.4105

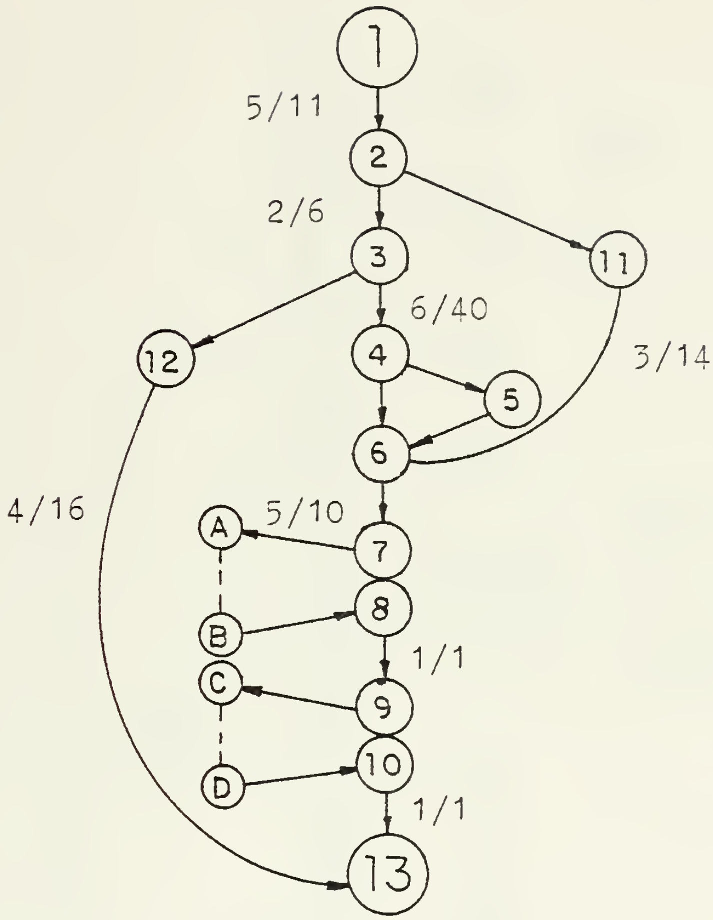
Percentage errors found: 28.74

Module: 1 Procedure No.: 25



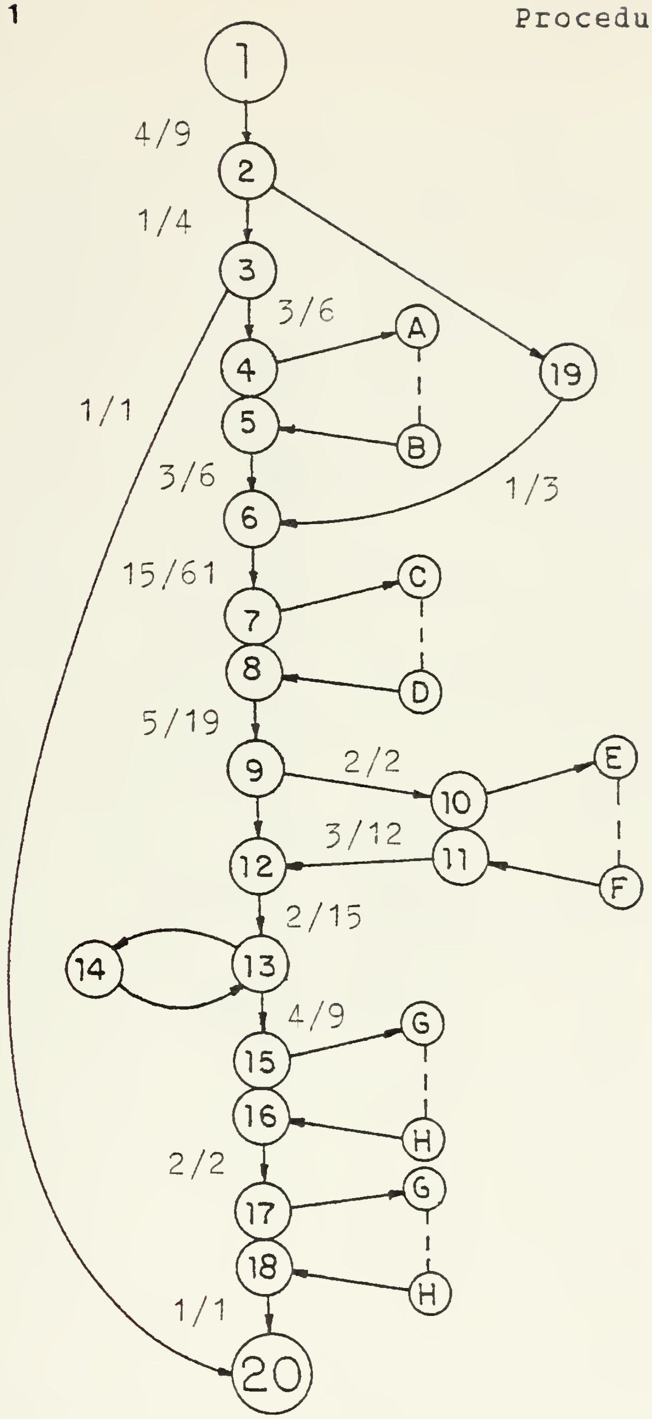
Number of nodes:	12
Number of arcs:	12
Number of paths:	2
Number of source stmts.:	8
Average error found:	0.2324
Percentage errors found:	61.01

Module: 1 Procedure No.: 28



Number of nodes:	17
Number of arcs:	19
Number of paths:	4
Number of source stmts.:	32
Average error found:	0.6400
Percentage errors found:	42.00

Module: 1 Procedure No.: 29



Number of nodes: 28

Number of arcs: 32

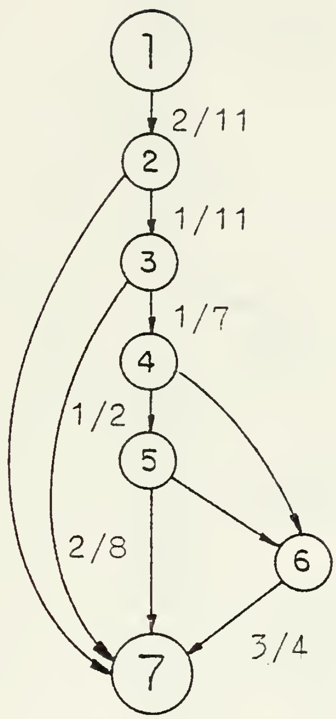
Number of paths: 9

Number of source stmts.: 47

Average error found: 1.3946

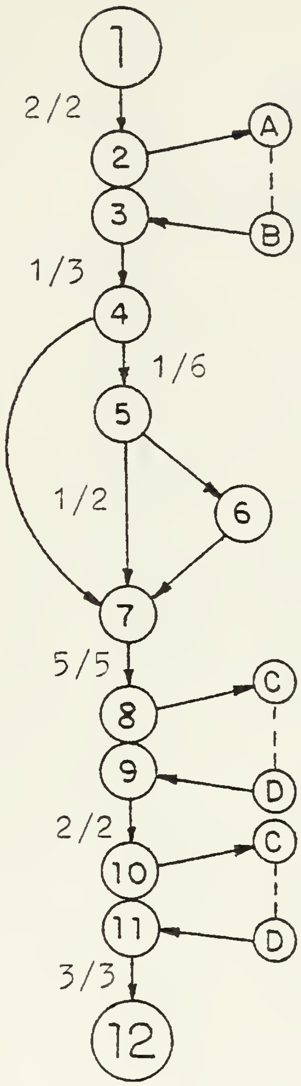
Percentage errors found: 62.31

Module: 1 Procedure No.: 30



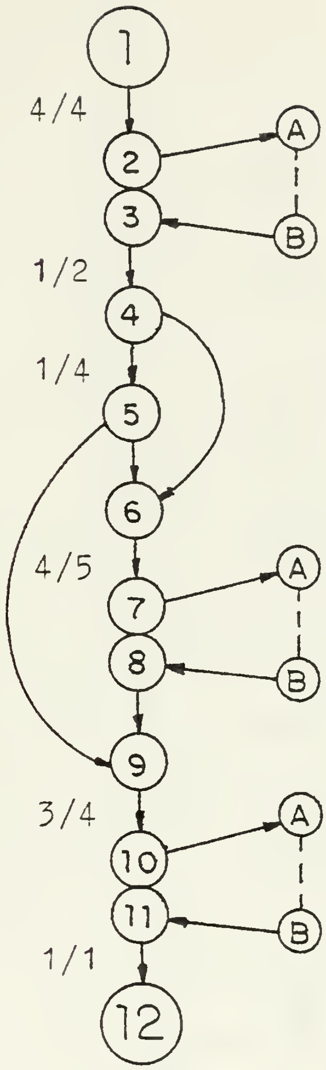
Number of nodes: 7
Number of arcs: 10
Number of paths: 5
Number of source stmts.: 10
Average error found: 0.1649
Percentage errors found: 34.63

Module: 1 Procedure No.: 34



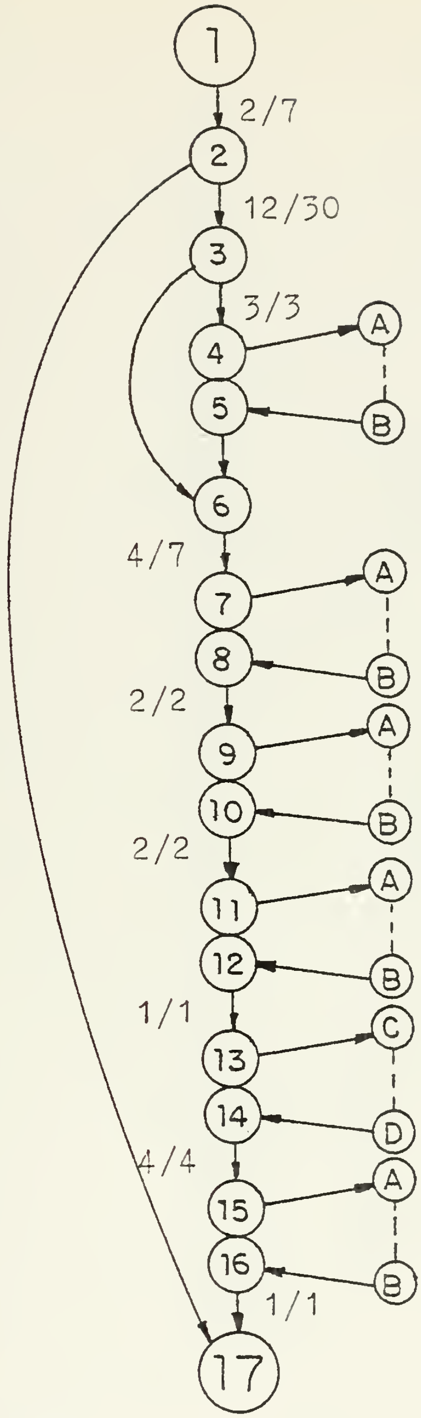
Number of nodes:	16
Number of arcs:	17
Number of paths:	3
Number of source stmts.:	5
Average error found:	0.5465
Percentage errors found:	76.51

Module: 1 Procedure No.: 35



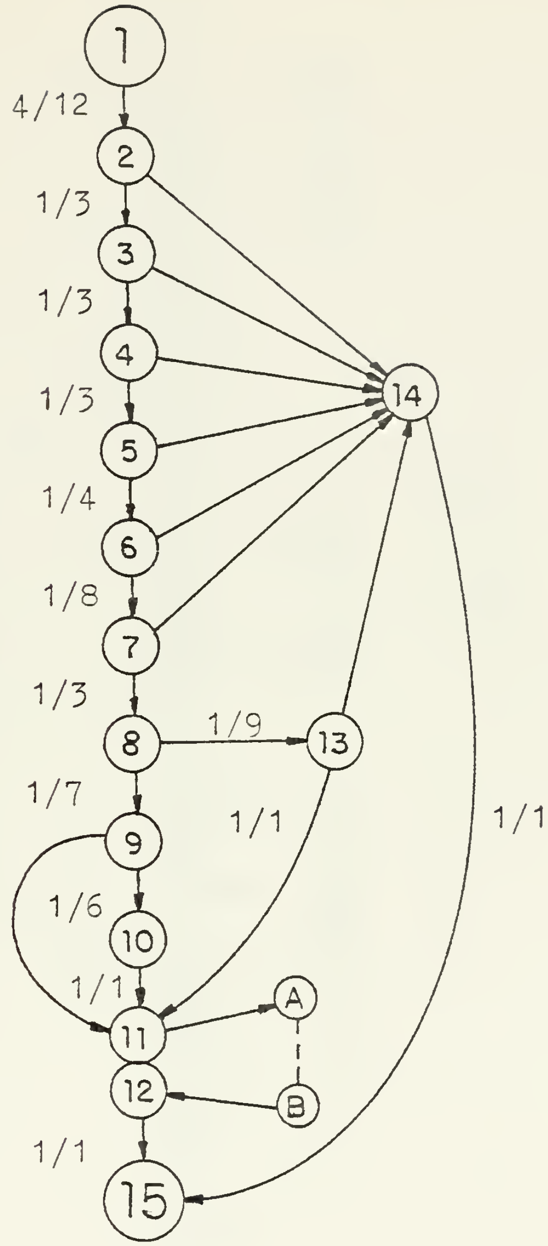
Number of nodes:	14
Number of arcs:	17
Number of paths:	3
Number of source stmts.:	14
Average error found:	0.3576
Percentage errors found:	53.64

Module: 1 Procedure No.: 36



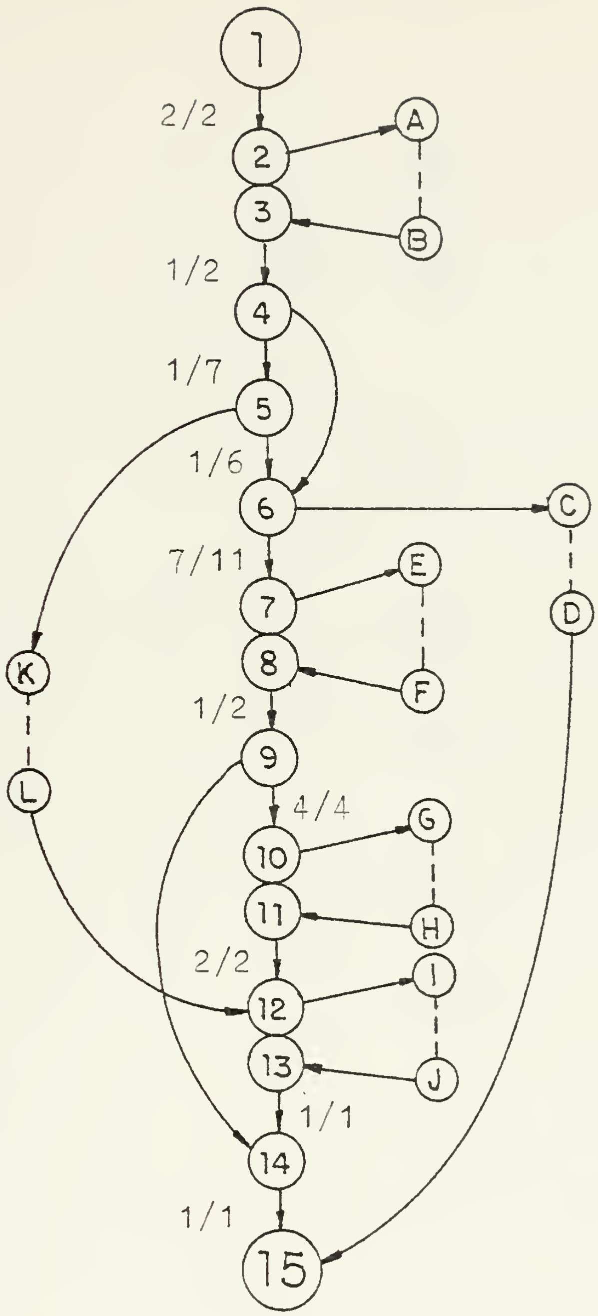
Number of nodes:	21
Number of arcs:	26
Number of paths:	3
Number of source stmts.:	31
Average error found:	0.5203
Percentage errors found:	35.25

Module: 1 Procedure No.: 39



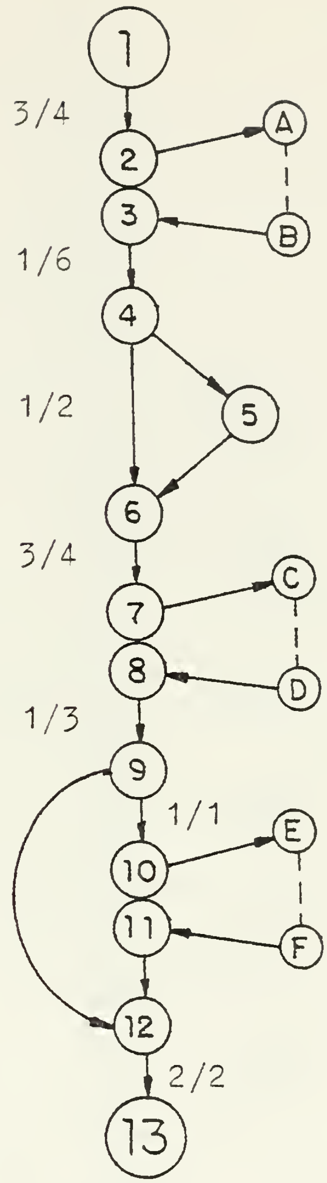
Number of nodes:	17
Number of arcs:	25
Number of paths:	10
Number of source stmts.:	17
Average error found:	0.2637
Percentage errors found:	32.57

Module: 1   Procedure No.: 44



Number of nodes:	27
Number of arcs:	30
Number of paths:	7
Number of source stmts.:	21
Average error found:	0.3554
Percentage errors found:	35.54

Module: 1 Procedure No.: 47



Number of nodes: 19

Number of arcs: 20

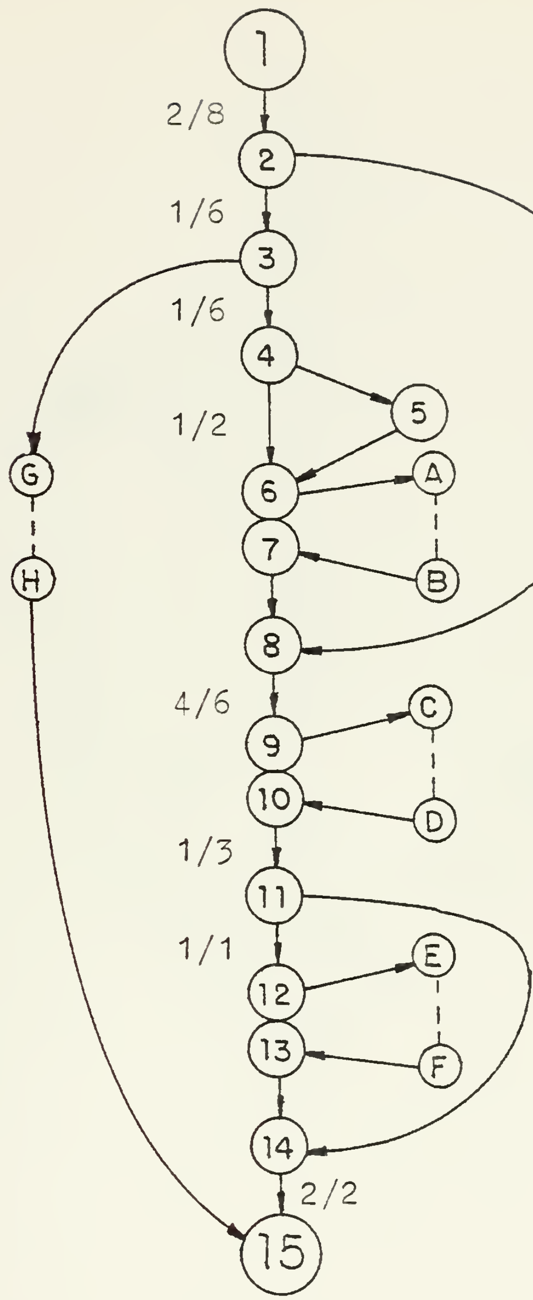
Number of paths: 4

Number of source stmts.: 12

Average error found: 0.4231

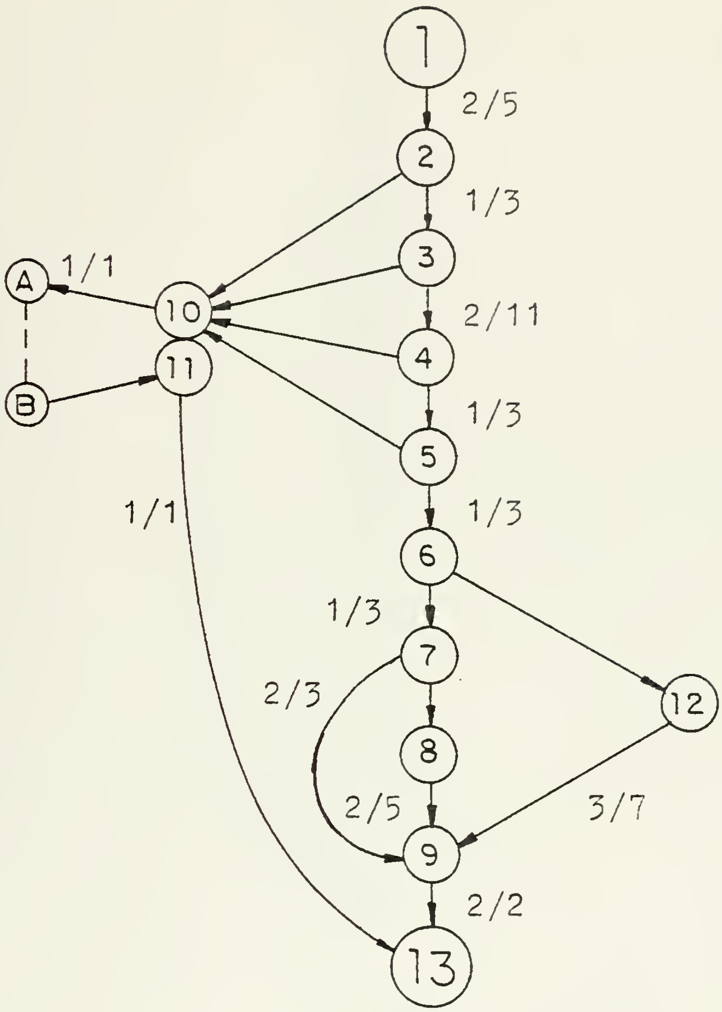
Percentage errors found: 74.04

Module: 1 Procedure No.: 48



Number of nodes:	23
Number of arcs:	26
Number of paths:	7
Number of source stmts.:	13
Average error found:	0.3287
Percentage errors found:	53.10

Module: 1 Procedure No.: 49



Number of nodes: 15

Number of arcs: 18

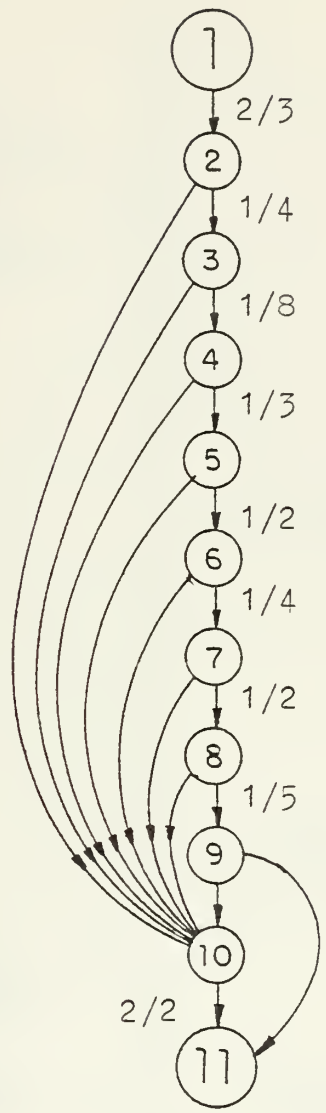
Number of paths: 7

Number of source stmts.: 19

Average error found: 0.2217

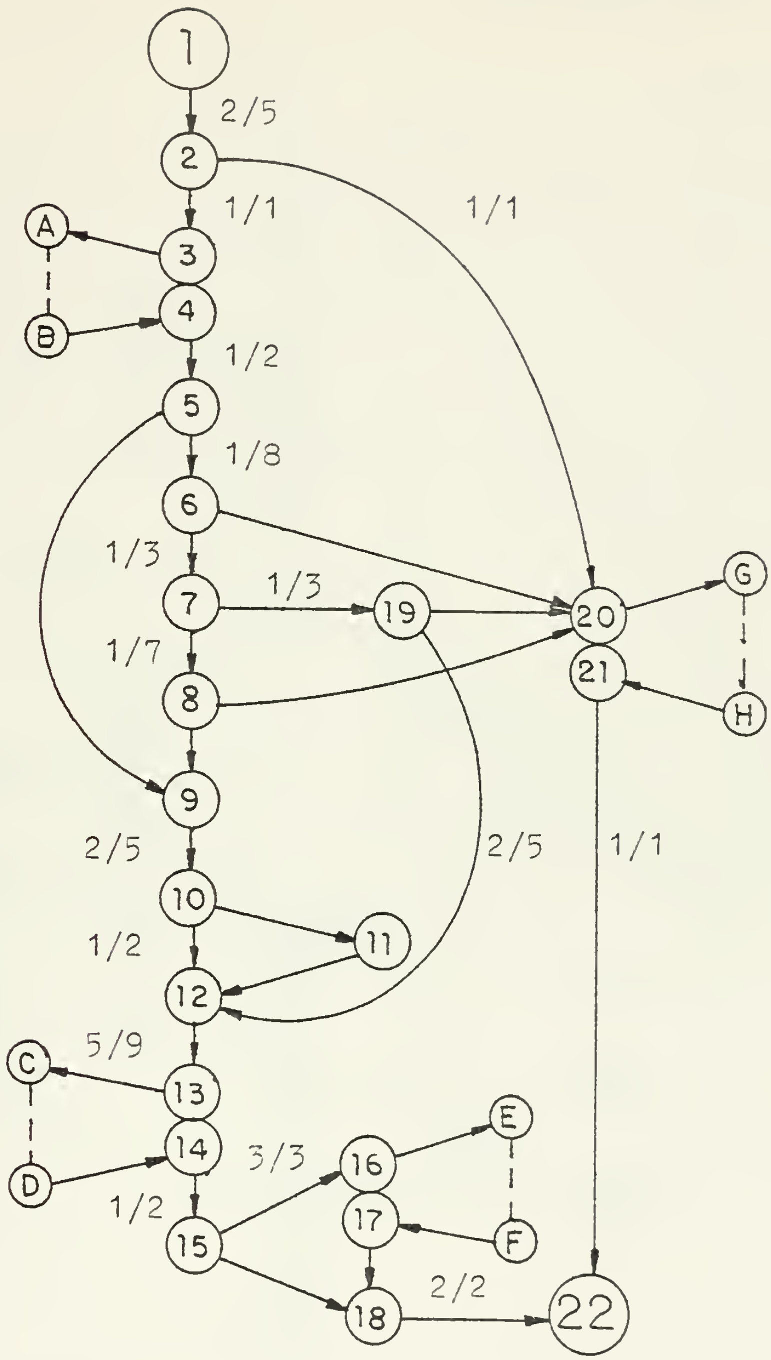
Percentage errors found: 24.50

Module: 1 Procedure No.: 53



Number of nodes:	11
Number of arcs:	18
Number of paths:	9
Number of source stmts.:	11
Average error found:	0.1876
Percentage errors found:	35.81

Module: 1 Procedure No.: 57



Number of nodes: 30

Number of arcs: 36

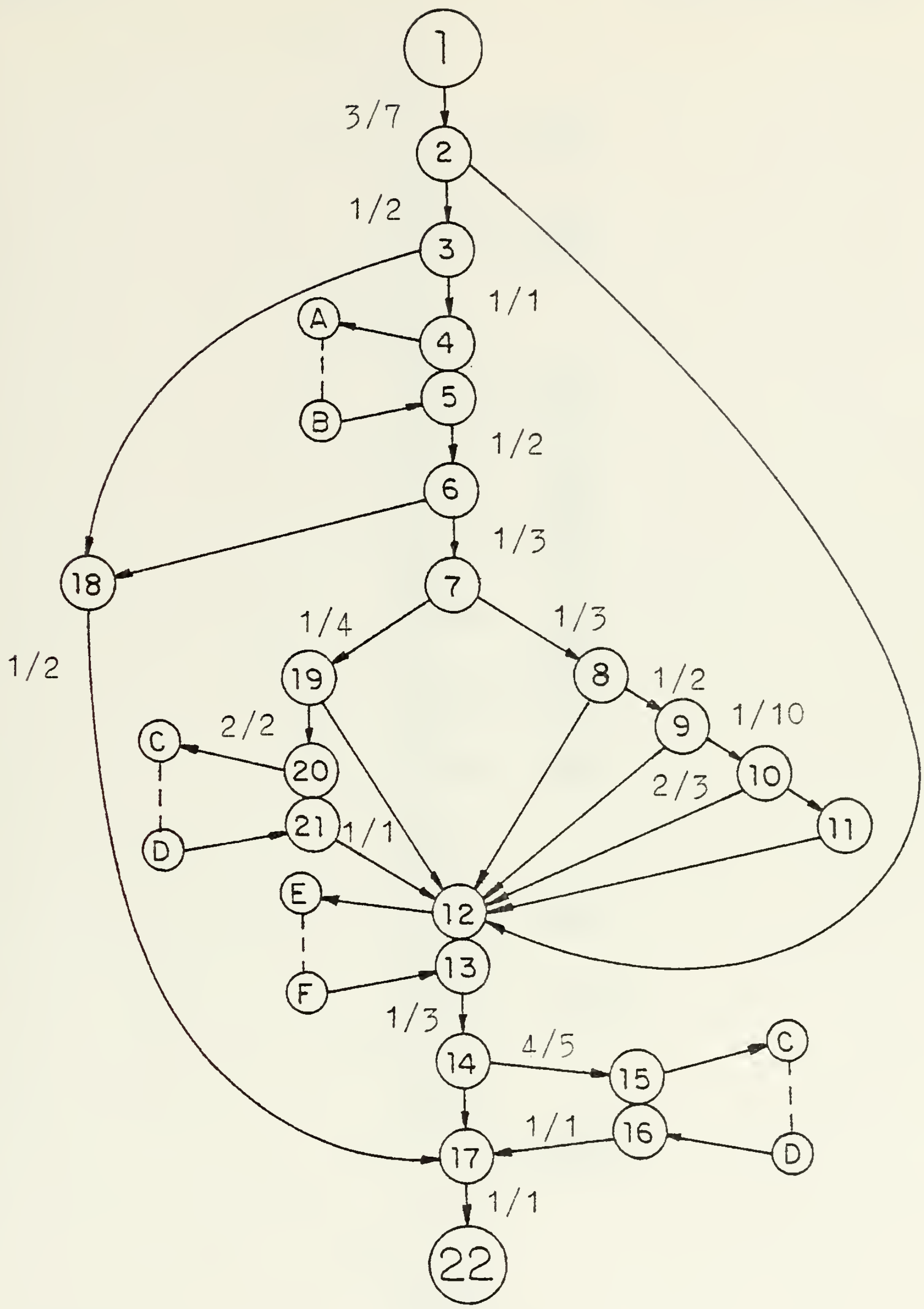
Number of paths: 14

Number of source stmts.: 26

Average error found: 0.2910

Percentage errors found: 23.50

Module: 1 Procedure No.: 60



Number of nodes: 28

Number of arcs: 37

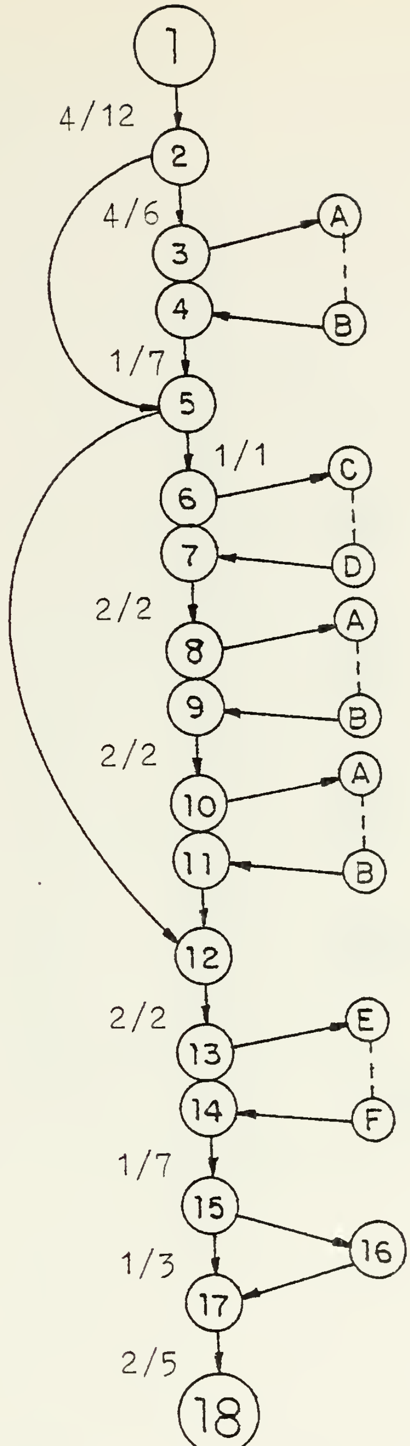
Number of paths: 18

Number of source stmts.: 24

Average error found: 0.3336

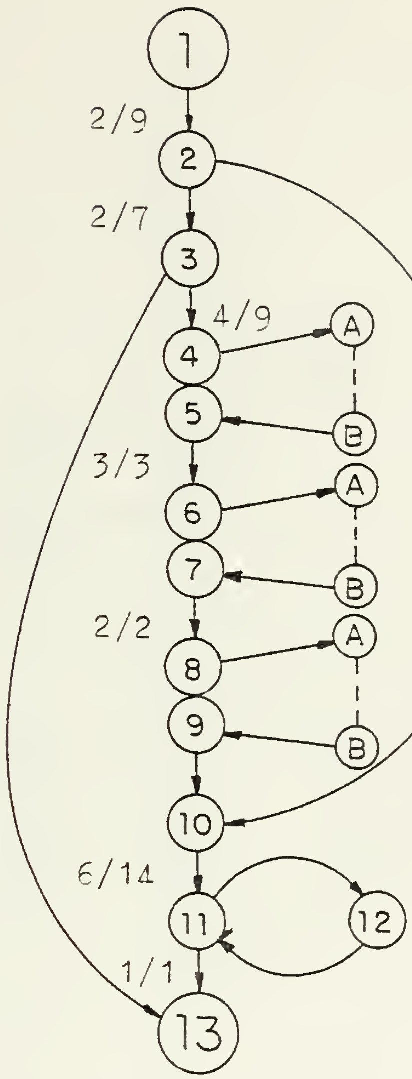
Percentage errors found: 29.19

Module: 1 Procedure No.: 75



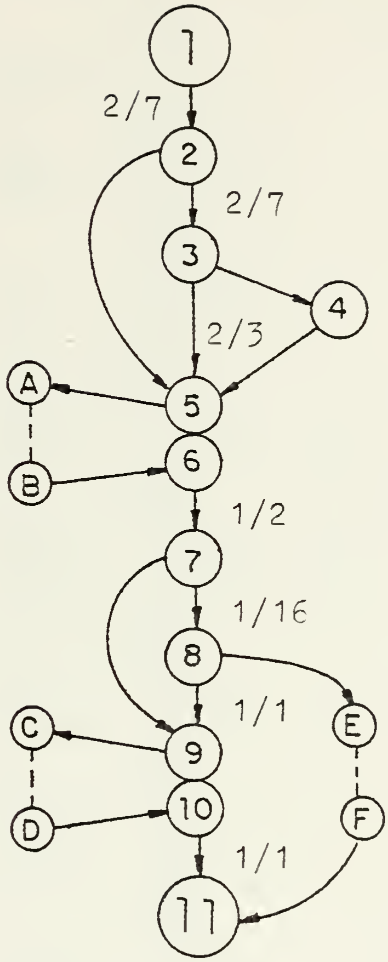
Number of nodes: 24
Number of arcs: 28
Number of paths: 8
Number of source stmts.: 20
Average error found: 0.5433
Percentage errors found: 57.05

Module: 1 Procedure No.: 76



Number of nodes: 15
Number of arcs: 19
Number of paths: 5
Number of source stmts.: 20
Average error found: 0.3893
Percentage errors found: 40.88

Module: 1 Procedure No.: 77



Number of nodes: 17

Number of arcs: 20

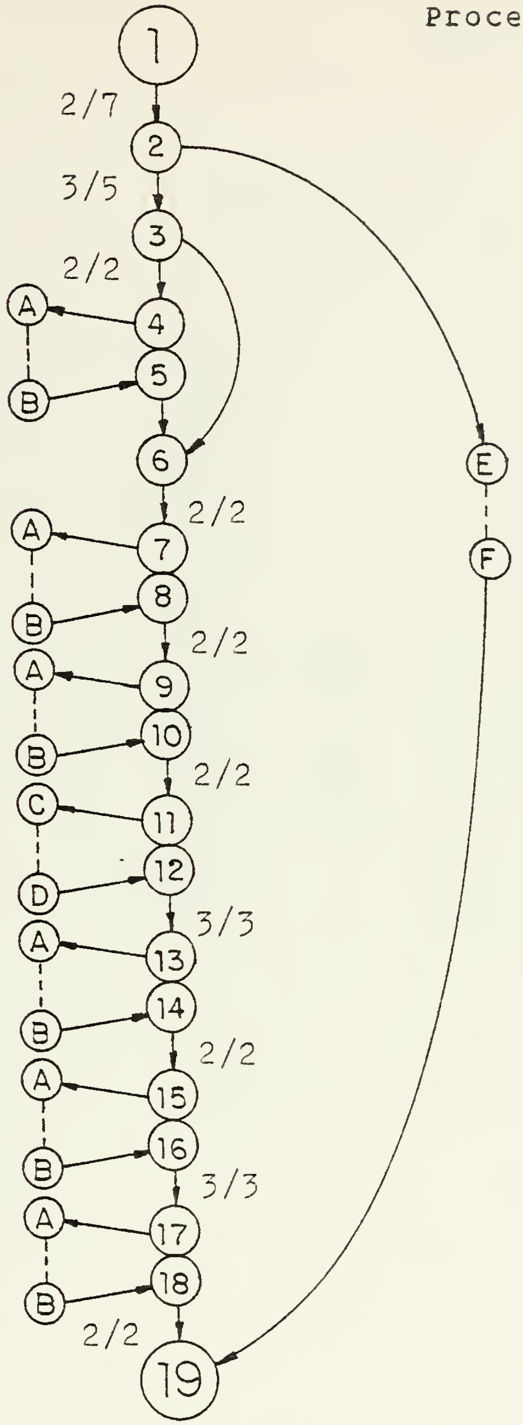
Number of paths: 9

Number of source stmts.: 10

Average error found: 0.2425

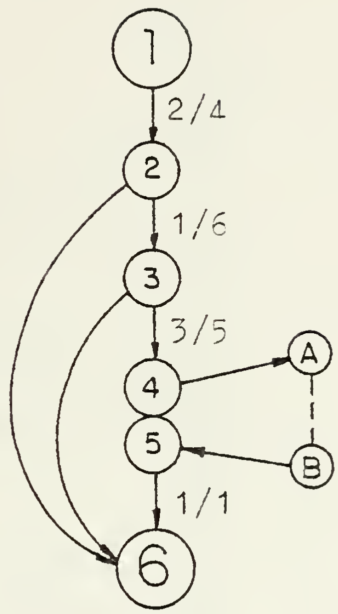
Percentage errors found: 50.93

Module: 1 Procedure No.: 79



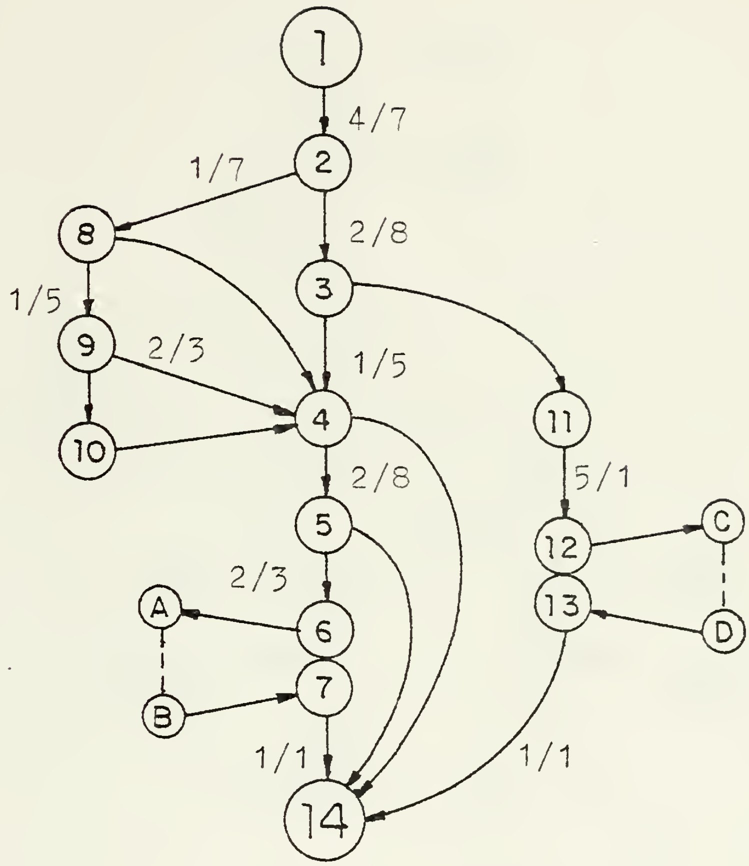
Number of nodes:	25
Number of arcs:	31
Number of paths:	3
Number of source stmts.:	23
Average error found:	0.3628
Percentage errors found:	33.13

Module: 1 Procedure No.: 81



Number of nodes:	8
Number of arcs:	9
Number of paths:	3
Number of source stmts.:	7
Average error found:	0.1449
Percentage errors found:	43.47

Module: 1 Procedure No.: 86



Number of nodes: 18

Number of arcs: 23

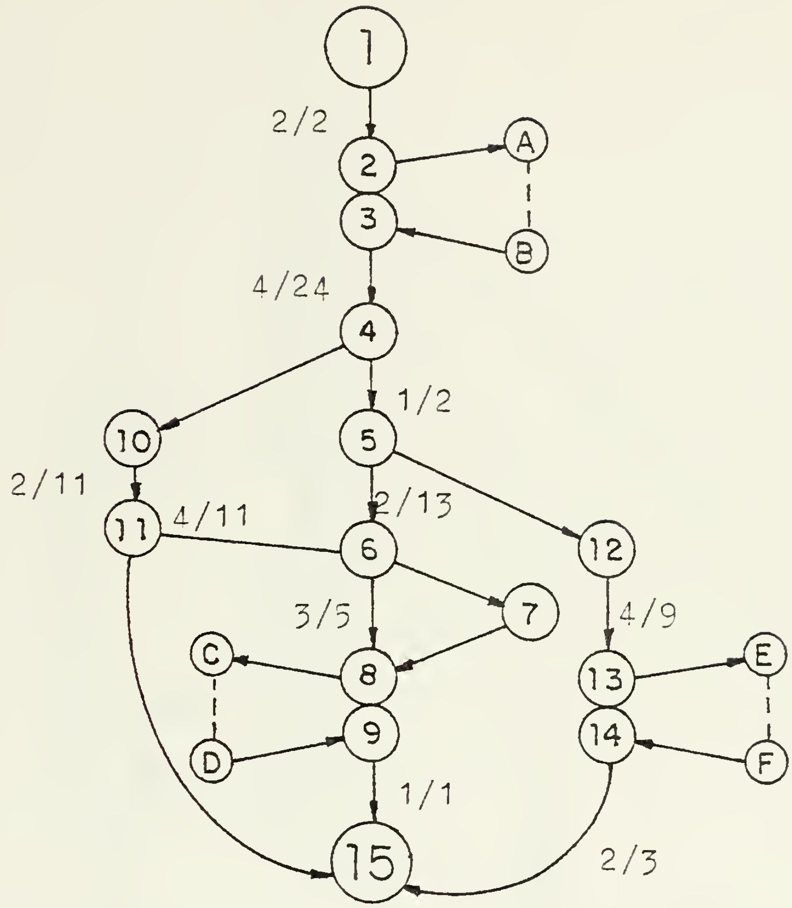
Number of paths: 13

Number of source stmts.: 22

Average error found: 0.3370

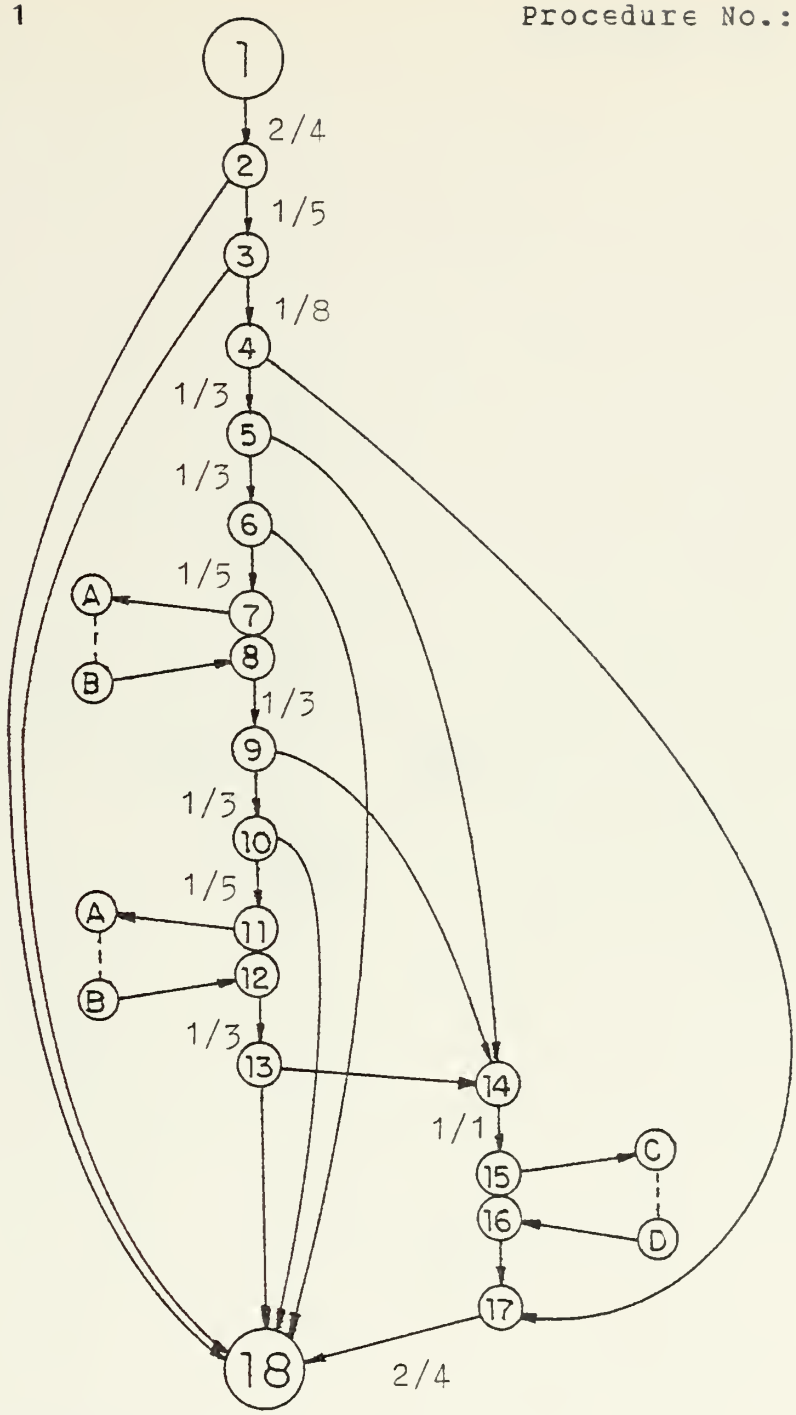
Percentage errors found: 32.17

Module: 1 Procedure No.: 87



Number of nodes: 21
Number of arcs: 22
Number of paths: 6
Number of source stmts.: 25
Average error found: 0.5029
Percentage errors found: 42.24

Module: 1 Procedure No.: 91



Number of nodes: 22

Number of arcs: 30

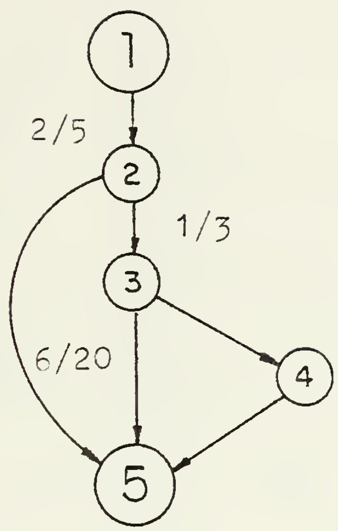
Number of paths: 9

Number of source stmts.: 14

Average error found: 0.1438

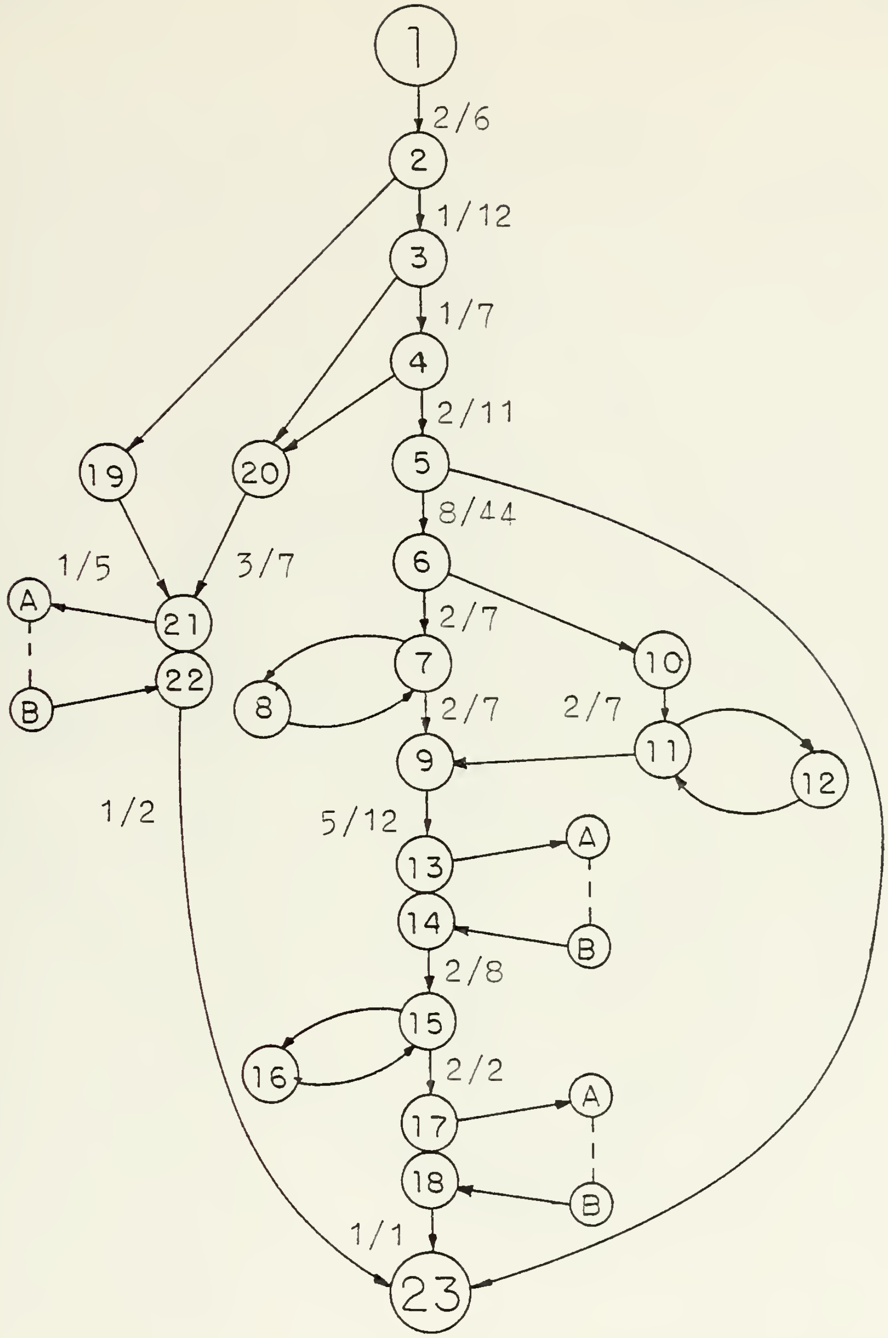
Percentage errors found: 21.57

Module: 1 Procedure No.: 92



Number of nodes:	5
Number of arcs:	6
Number of paths:	3
Number of source stmts.:	9
Average error found:	0.1837
Percentage errors found:	42.86

Module: 1 Procedure No.: 93



Number of nodes: 25

Number of arcs: 34

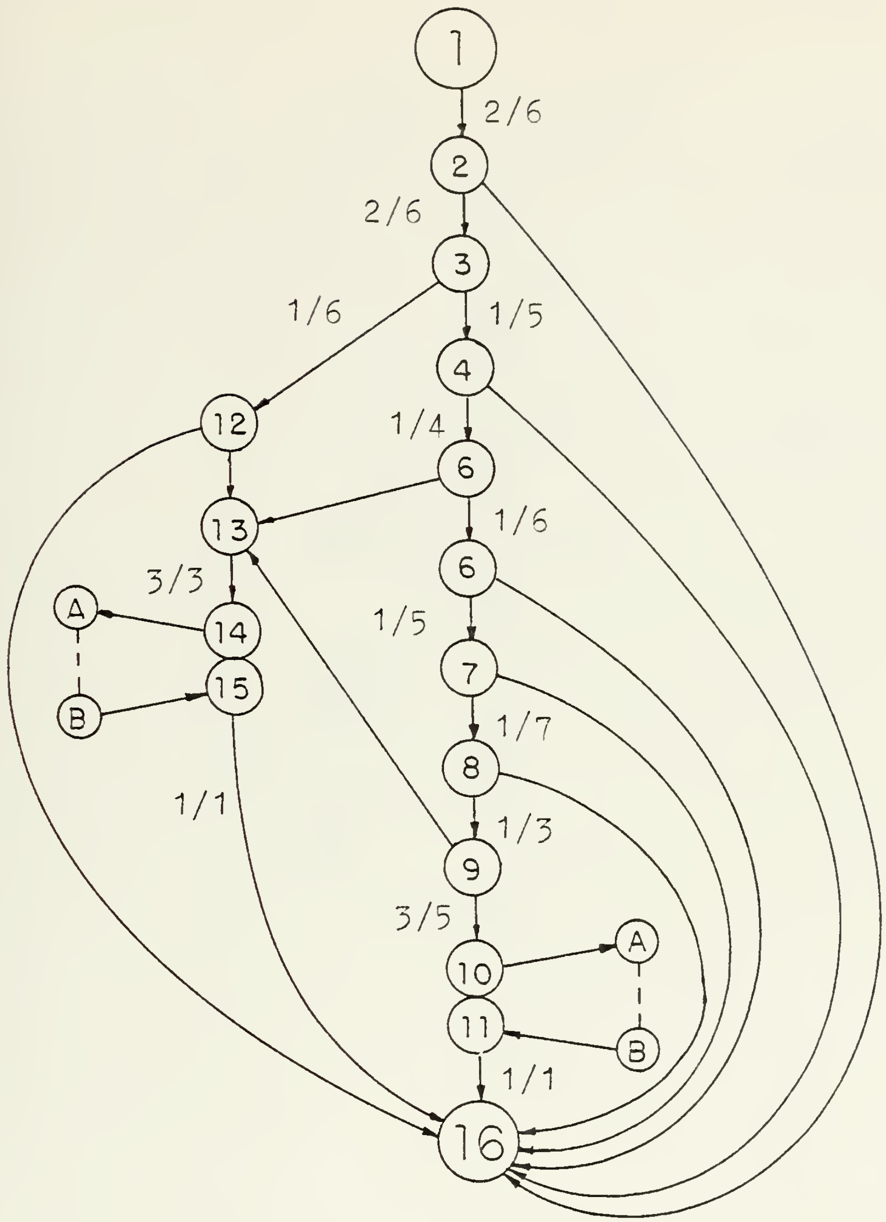
Number of paths: 12

Number of source stmts.: 34

Average error found: 0.3972

Percentage errors found: 24.53

Module: 1 Procedure No.: 95



Number of nodes: 18

Number of arcs: 27

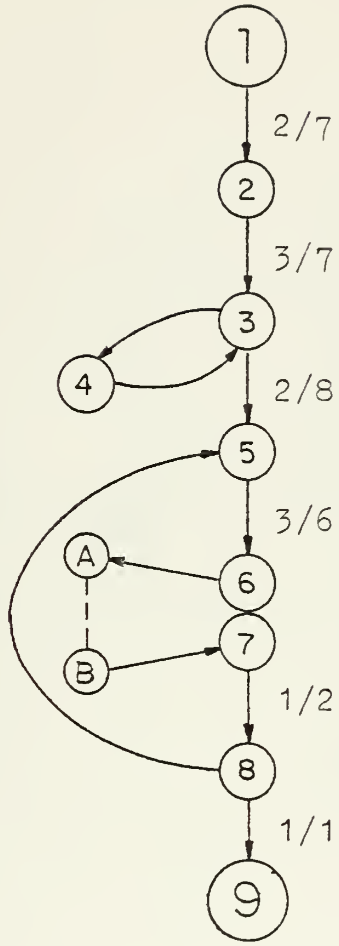
Number of paths: 10

Number of source stmts.: 19

Average error found: 0.1822

Percentage errors found: 20.14

Module: 2 Procedure No.: 15



Number of nodes: 11

Number of arcs: 13

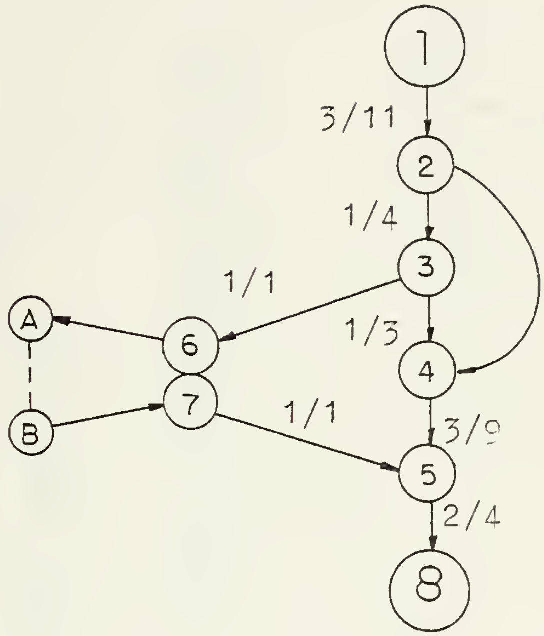
Number of paths: 5

Number of source stmts.: 12

Average error found: 0.0836

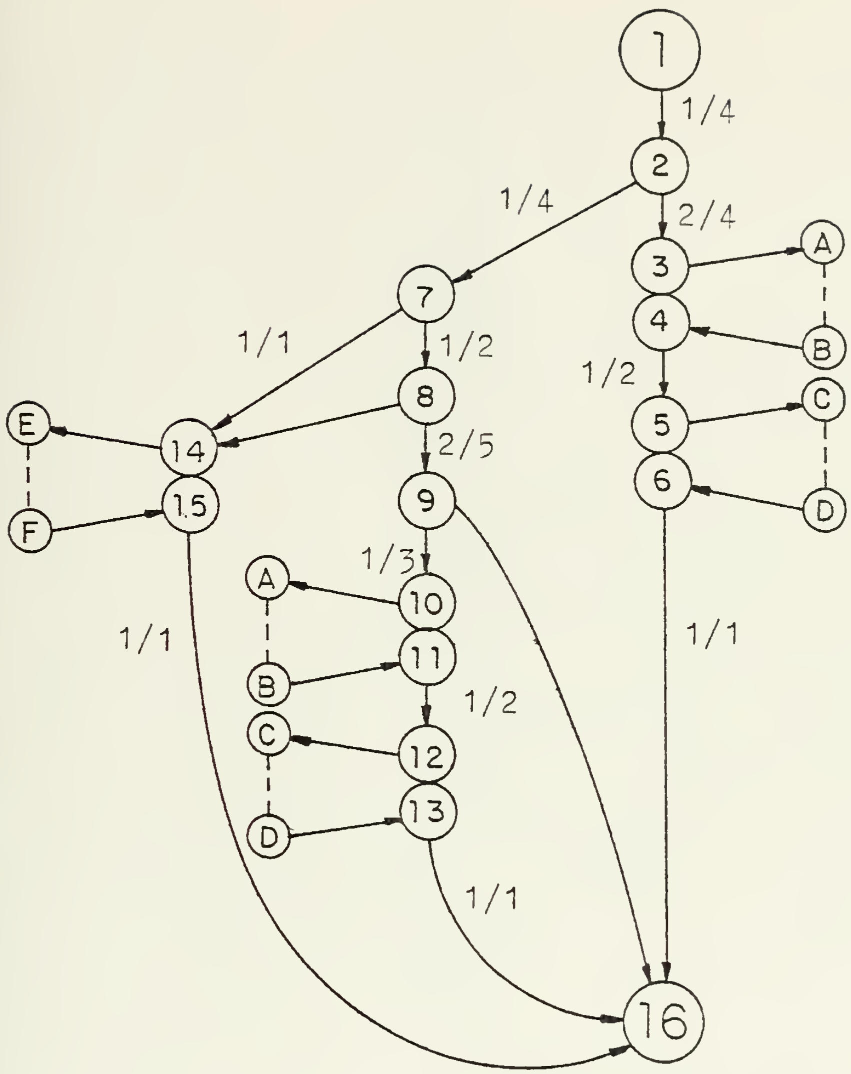
Percentage errors found: 35.59

Module: 2 Procedure No.: 23



Number of nodes:	10
Number of arcs:	11
Number of paths:	3
Number of source stmts.:	12
Average error found:	0.1592
Percentage errors found:	67.66

Module: 2 Procedure No.: 40



Number of nodes: 22

Number of arcs: 27

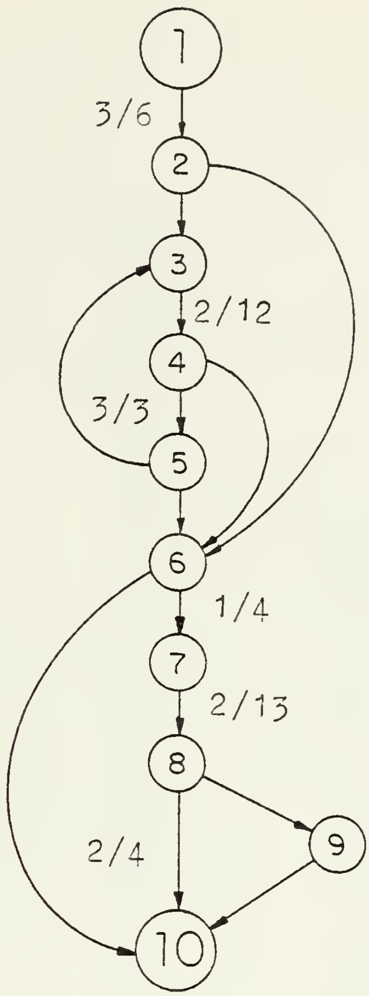
Number of paths: 5

Number of source stmts.: 14

Average error found: 0.2018

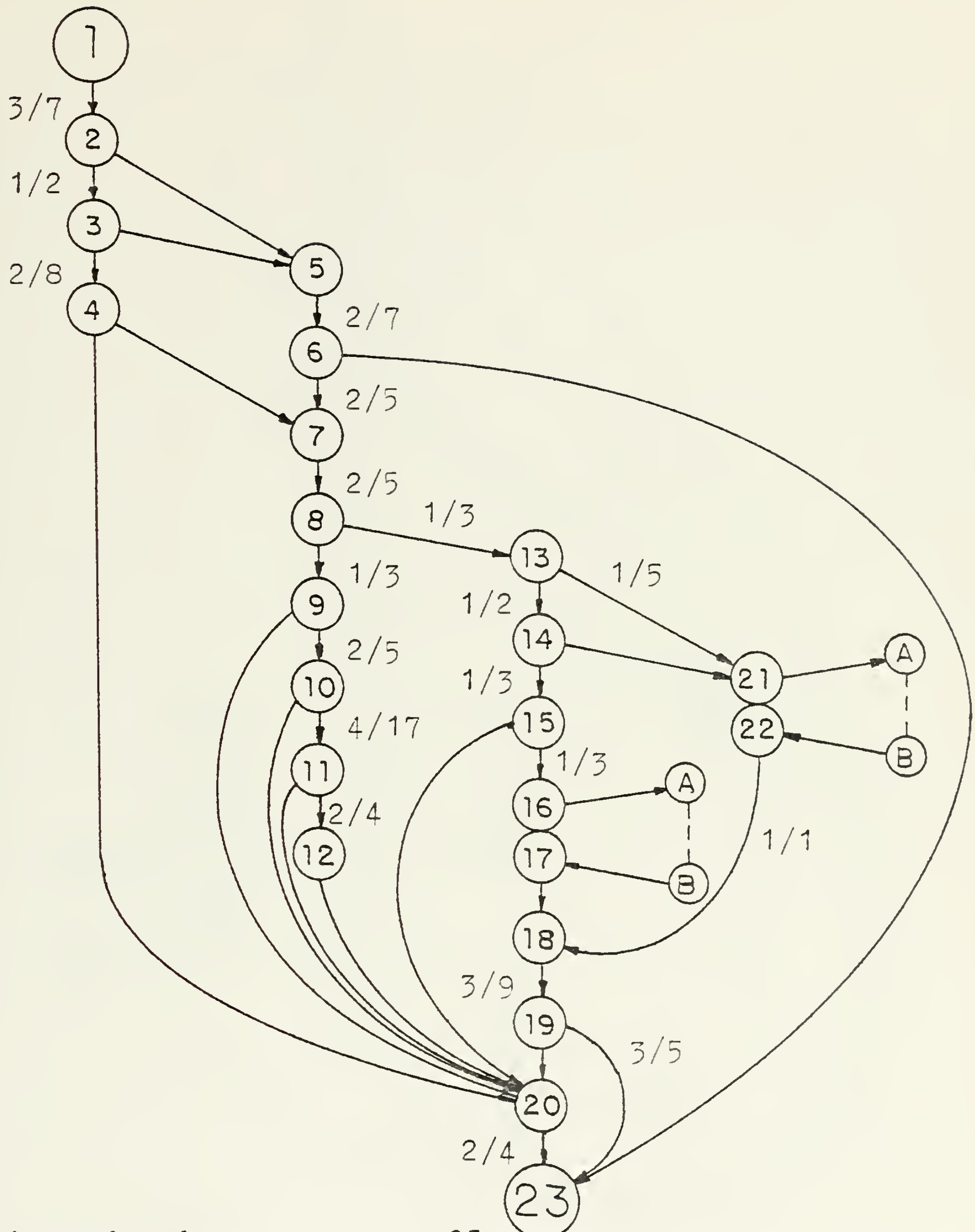
Percentage errors found: 73.51

Module: 2 Procedure No.: 41



Number of nodes:	10
Number of arcs:	14
Number of paths:	12
Number of source stmts.:	13
Average error found:	0.1554
Percentage errors found:	60.96

Module: 2 Procedure No.: 46



Number of nodes: 25

Number of arcs: 37

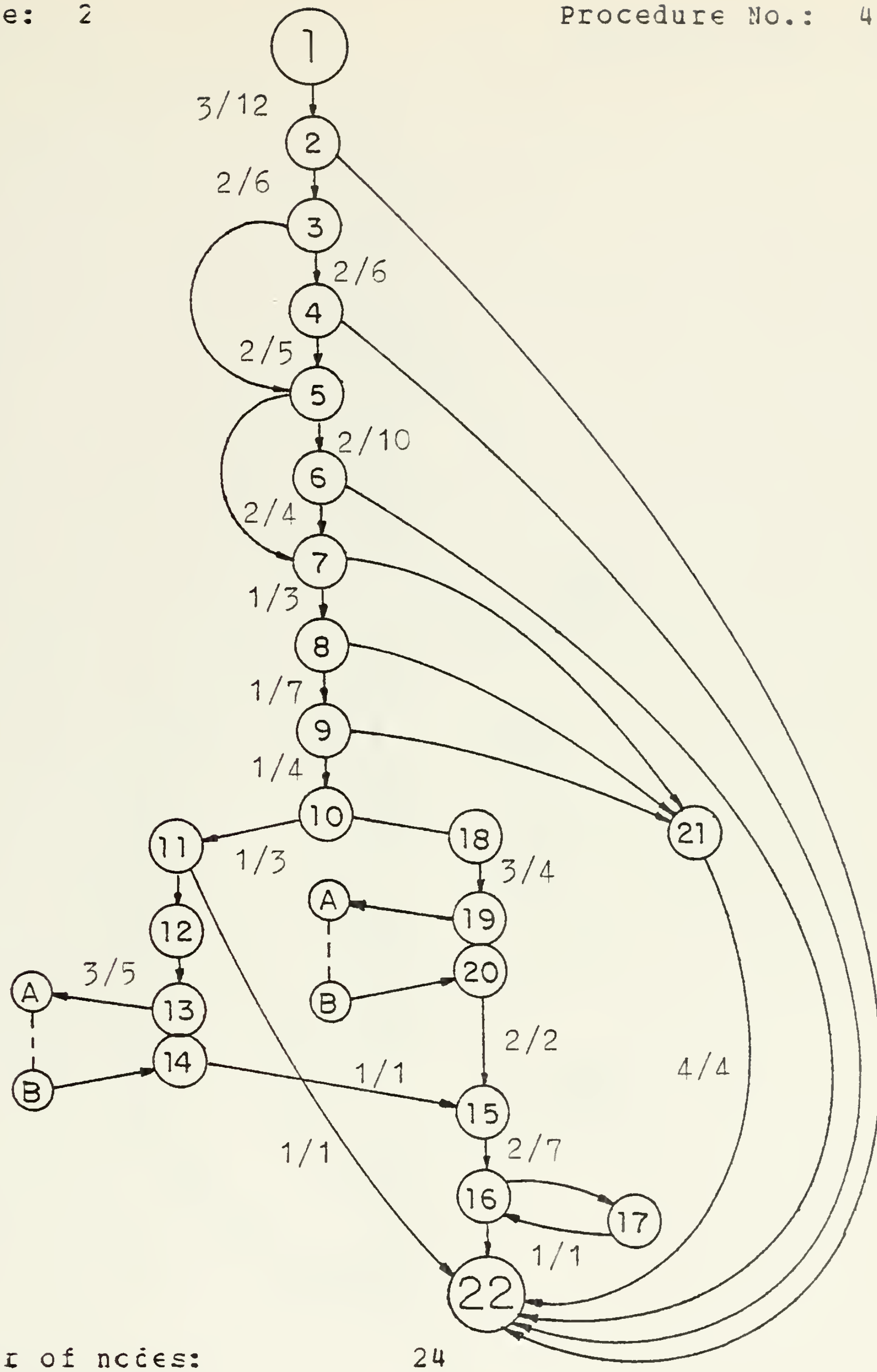
Number of paths: 36

Number of source stmts.: 34

Average error found: 0.1657

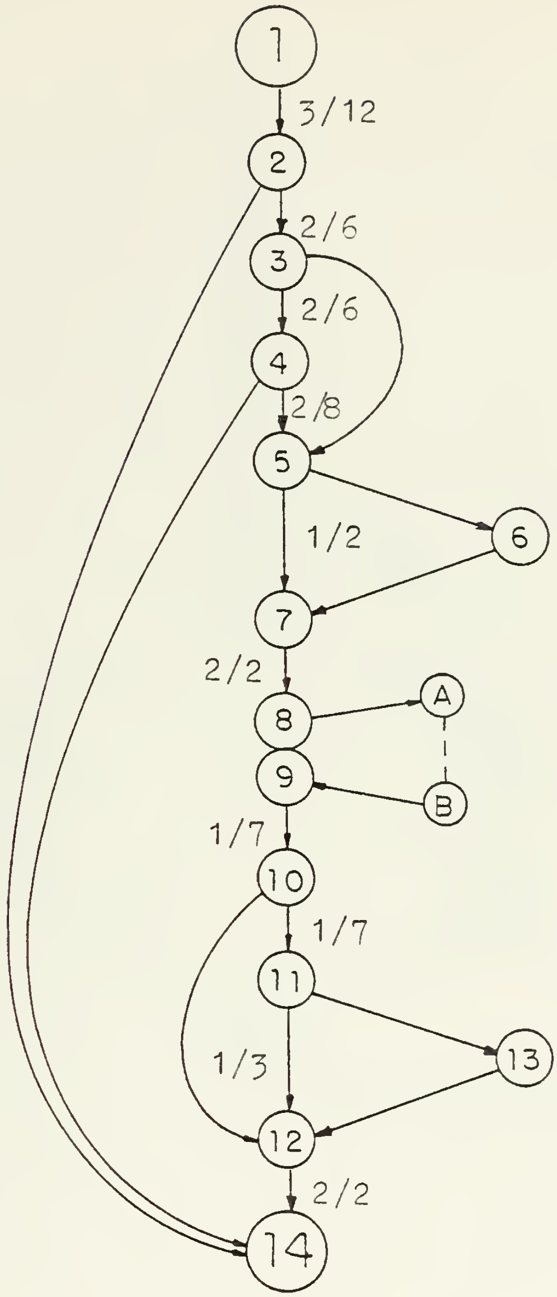
Percentage errors found: 24.86

Module: 2 Procedure No.: 47



Number of nodes:	24
Number of arcs:	34
Number of paths:	36
Number of source stmts.:	34
Average error found:	0.1163
Percentage errors found:	17.45

Module: 2 Procedure No.: 48



Number of nodes: 16

Number of arcs: 21

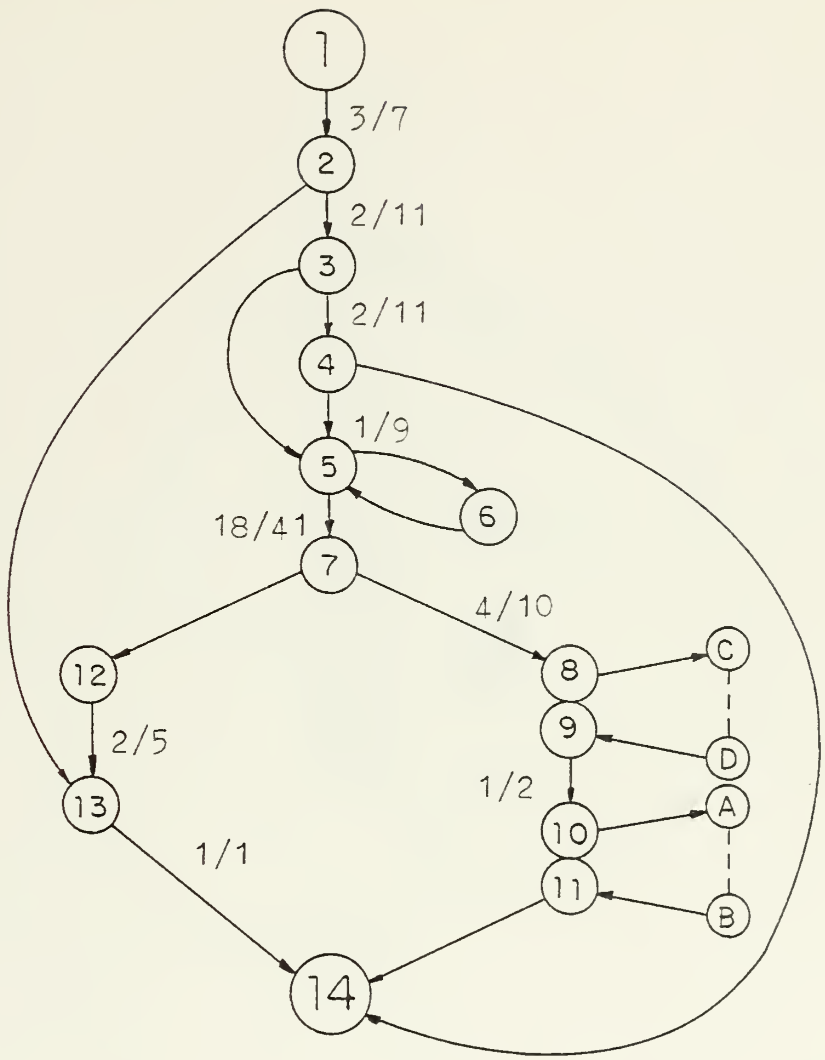
Number of paths: 14

Number of source stmts.: 17

Average error found: 0.1580

Percentage errors found: 47.40

Module: 2 Procedure No.: 73



Number of nodes: 18

Number of arcs: 22

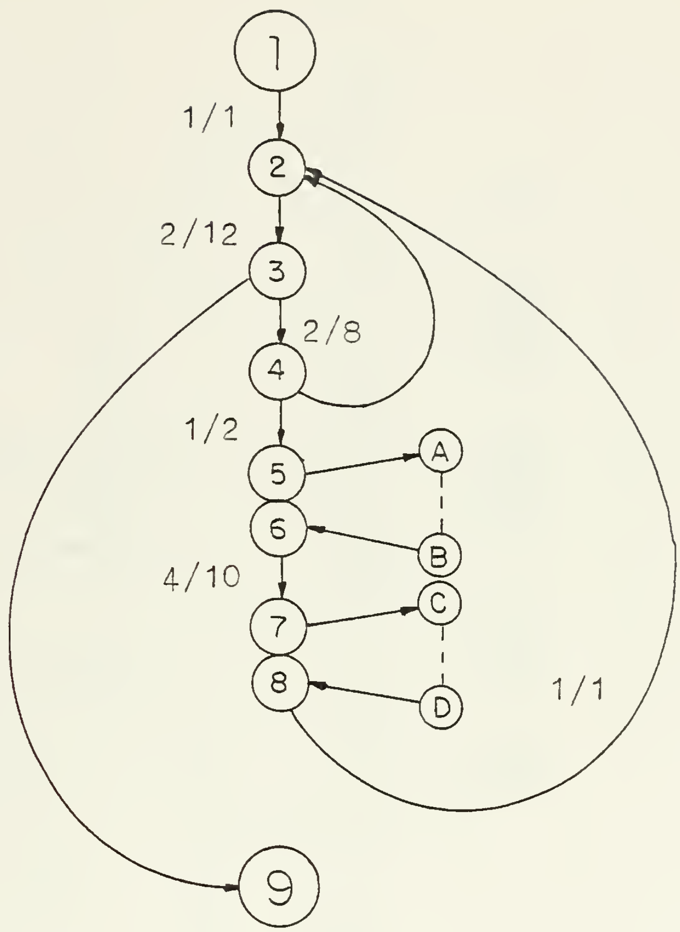
Number of paths: 10

Number of source stmts.: 34

Average error found: 0.1885

Percentage errors found: 28.28

Module: 2 Procedure No.: 79



Number of nodes: 13

Number of arcs: 14

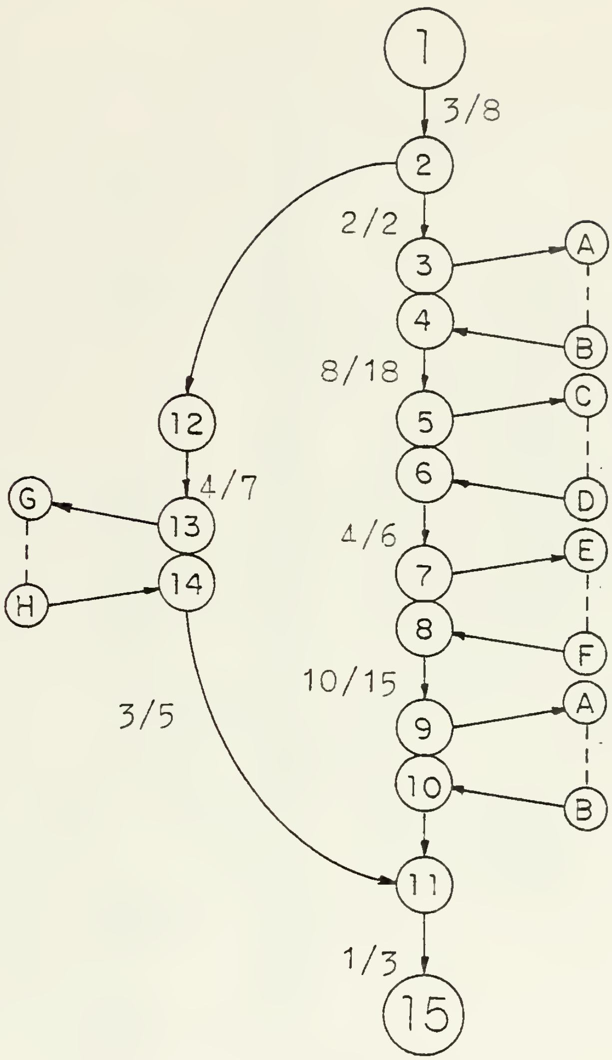
Number of paths: 5

Number of source stmts.: 11

Average error found: 0.1379

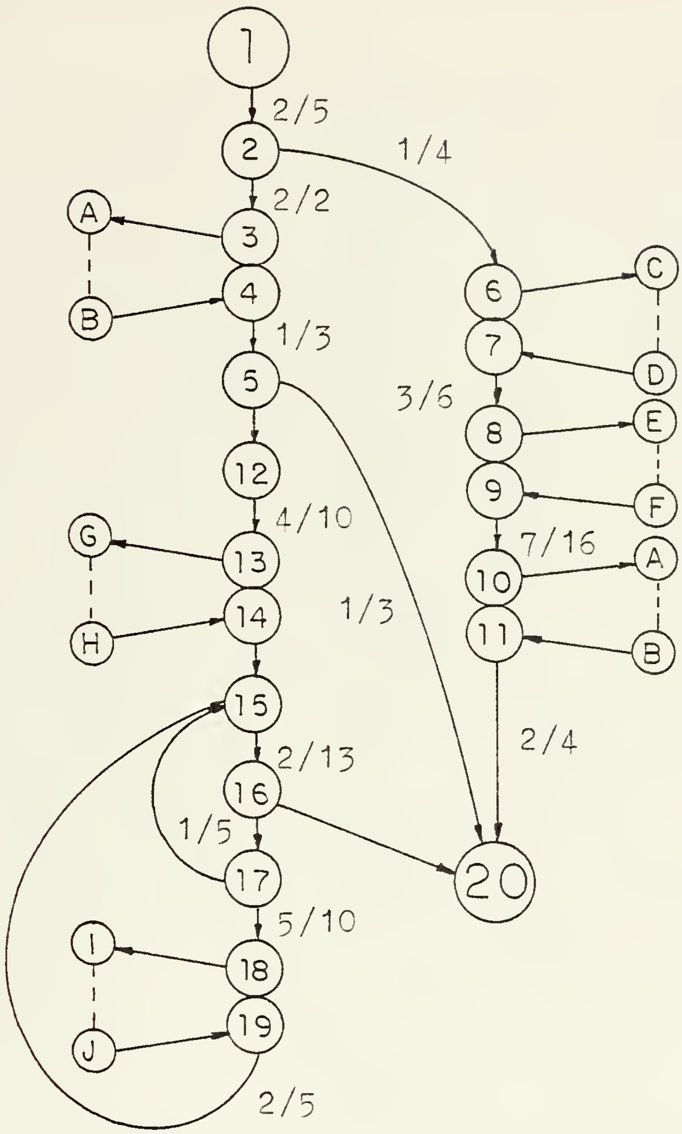
Percentage errors found: 63.94

Module: 2 Procedure No.: 82



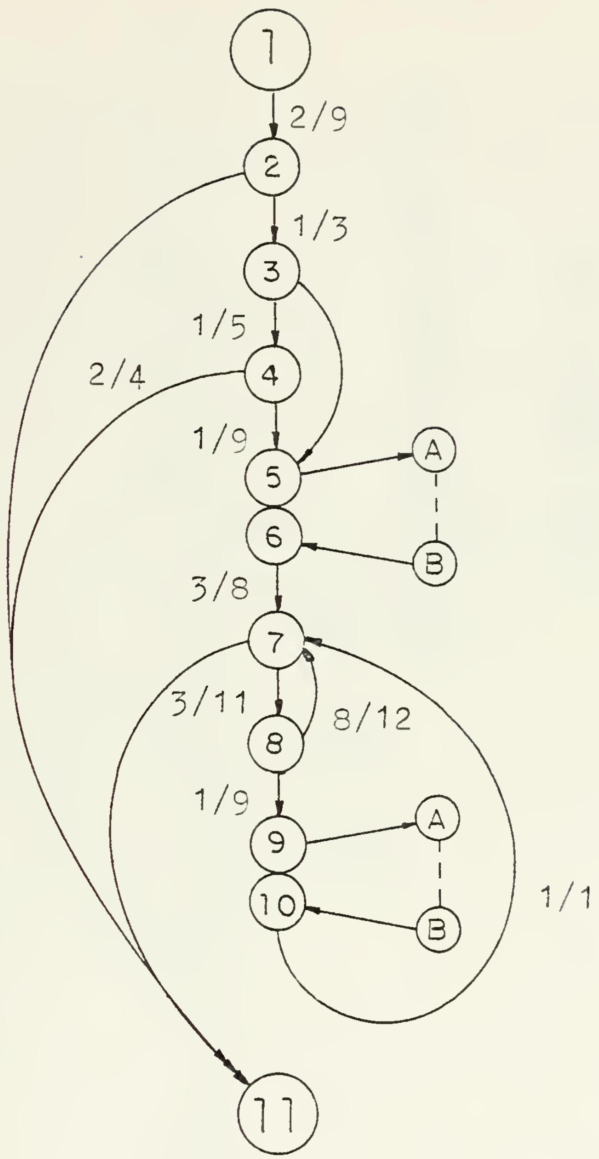
Number of nodes:	23
Number of arcs:	24
Number of paths:	2
Number of source stmts.:	35
Average error found:	0.1130
Percentage errors found:	16.47

Module: 2 Procedure No.: 86



Number of nodes:	30
Number of arcs:	34
Number of paths:	6
Number of source stmts.:	33
Average error found:	0.2042
Percentage errors found:	31.56

Module: 2 Procedure No.: 90



Number of nodes: 13

Number of arcs: 18

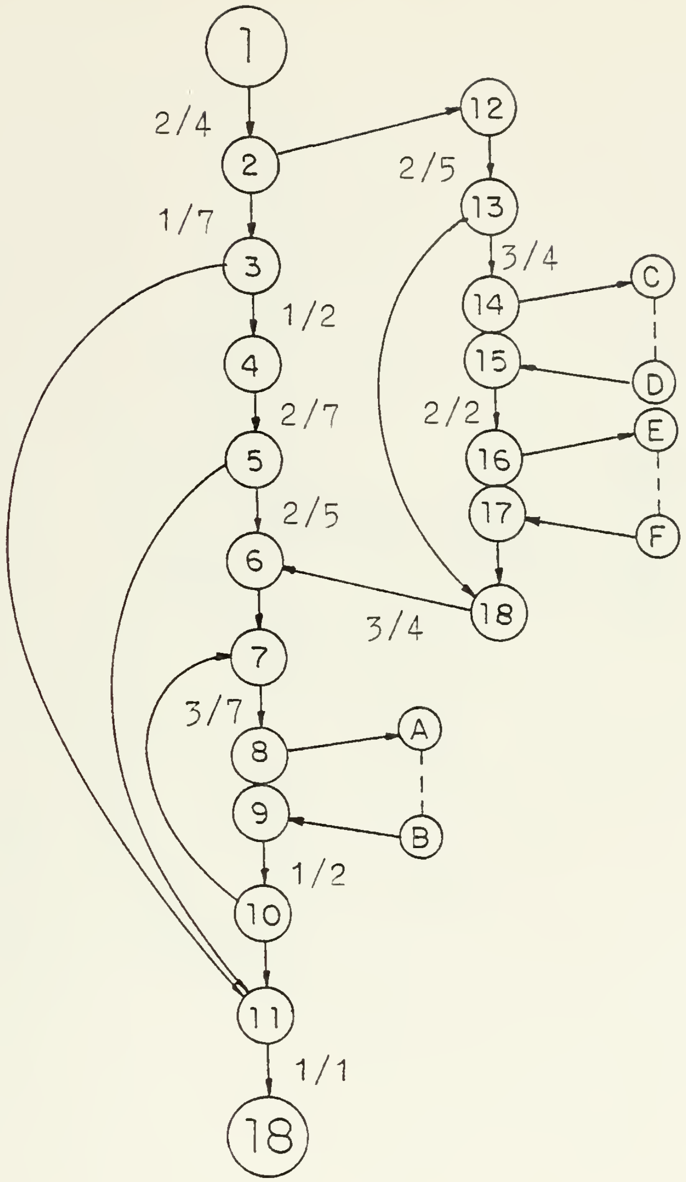
Number of paths: 8

Number of source stmts.: 23

Average error found: 0.0958

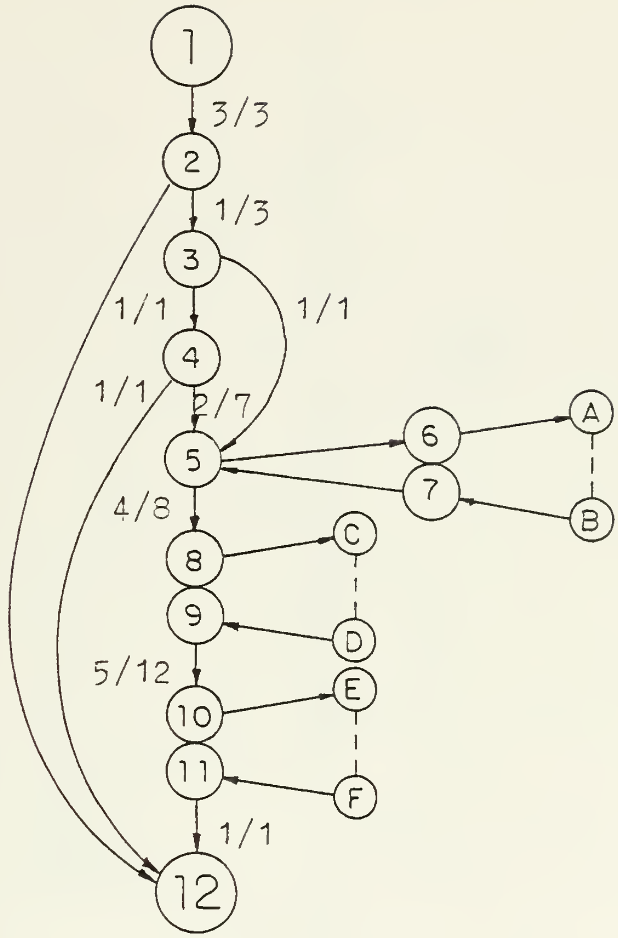
Percentage errors found: 21.24

Module: 2 Procedure No.: 99



Number of nodes:	25
Number of arcs:	30
Number of paths:	10
Number of source stmts.:	23
Average error found:	0.1513
Percentage errors found:	33.55

Module: 2 Procedure No.: 122



Number of nodes: 18

Number of arcs: 21

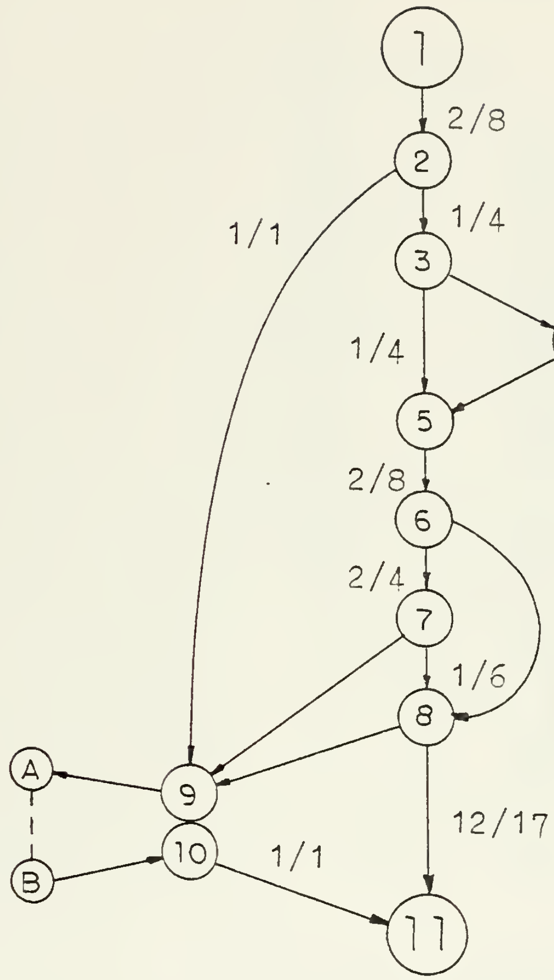
Number of paths: 6

Number of source stmts.: 20

Average error found: 0.1686

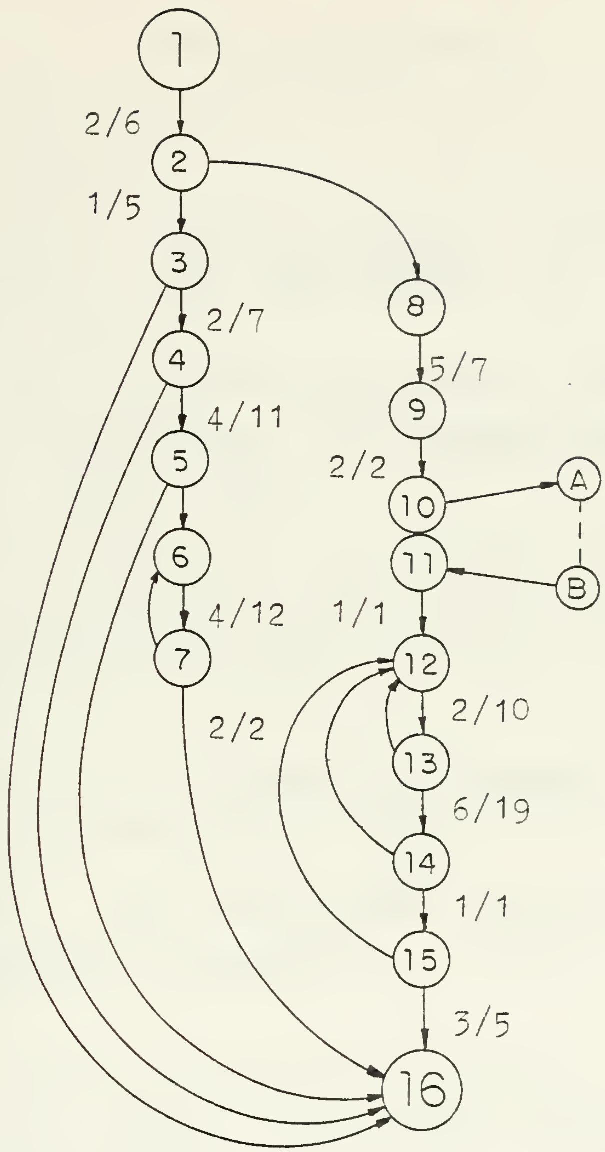
Percentage errors found: 42.99

Module: 2 Procedure No.: 137



Number of nodes:	13
Number of arcs:	17
Number of paths:	11
Number of source stmts.:	23
Average error found:	0.1178
Percentage errors found:	26.12

Module: 2 Procedure No.: 149



Number of nodes:	18
Number of arcs:	25
Number of paths:	9
Number of source stmts.:	35
Average error found:	0.2357
Percentage errors found:	34.34

LIST OF REFERENCES

1. Baker, F.T., " System quality through structured programming", *FJCC Proceedings*, v. 41 part I, p. 339-343, 1972.

2. Baker, F.T., " Chief programmer team management of production programming", *IBM Systems Journal*, v. 11 nr. 1, p. 56-73, 1972.

3. Boehm, C. and Jacopini, G., " Flow Diagrams, Turing Machines and Languages with only Two Formation Rules", *Communications of the ACM*, v. 9 nr. 5, p. 366-371, May 1966.

4. Constantine, I.I., *Concepts in Program Design*, p. 7 to 17, I.L. Constantine, 1967.

5. Dijkstra, " GO TO-Statement Considered Harmful", *Communications of the ACM*, v. 17, Nr. 3, p. 147/148, March 1968.

6. Fleet Computer Programming Center, Pacific, *Naval Tactical Data Systems, Programmers Guide*, Vol. I and II, 1 December 1964.

7. Green, T.F., *Software Error Detection Model*, M.S. Thesis, Naval Postgraduate School, Monterey, 1975.

8. Hetzel, W.C., *Program Test Methods*, p. 7 to 14, 225 to 238, Prentice-Hall, 1973.

9. Maynard, J., *Modular Programming*, p. 6 to 18, Auerbach Publishers, 1972.

10. Naval Postgraduate School, *System Test Methodology* ,

by Bradley, G.H., Howard, G.T., Schneidewind, N.F., Motgomery, G.W., and Green, T.F., Vol I and II, July 1975.

11. Schneidewind, N.F., Validation Tests for a Software Error Simulation Model, Proceedings of the 1976 Summer Computer Simulation Conference, Washington D.C., July 12-14, 1976.

12. Van Tassel, D., Program Style, Design, Efficiency, Debugging and Testing, p. 44 to 47, 166 to 192, Prentice-Hall, Inc., 1974.

13. Yohe, J.M., " An Overview of Programming Practices", Computing Surveys, v. 6,4, p. 221-243, December 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Professor Norman F. Schneidewind, Code 32B Department of Operations Research and Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
5.	Marineamt -A 1- 294 Wilhelmshaven Federal Republic of Germany	1
6.	Dokumentationszentrale der Bundeswehr (See) 53 Bonn Friederich-Ebert-Allee 34 Federal Republic of Germany	1

7. Kommando Marinefuehrungssysteme 1
294 Wilhelmshaven
4. Einfahrt
Federal Republic of Germany

8. LCDR Michael A. Kirchgaessner 1
Hermann-von-Vicaristr. 5
775 Konstanz/B.
Federal Republic of Germany