

**Calhoun: The NPS Institutional Archive**  
**DSpace Repository**

---

Reports and Technical Reports

All Technical Reports Collection

---

1975-07

## System Test Methodology Vol. I

Bradley, G.H.; Howard, G.T.; Schneidewind, N.F.;  
Montgomery, G.W.; Green, T.F.

Monterey, California: Naval Postgraduate School

---

<http://hdl.handle.net/10945/63316>

---

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

*Downloaded from NPS Archive: Calhoun*



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

<http://www.nps.edu/library>

NPS55SS75072 A

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



SYSTEM TEST METHODOLOGY VOL. I

by

G. H. BRADLEY

G. T. HOWARD

N. F. SCHNEIDEWIND

G. W. MONTGOMERY

T. F. GREEN

July 1975

Approved for public release; distribution unlimited

Prepared for:  
Naval Air Development Center  
Warminster, Pennsylvania

NAVAL POSTGRADUATE SCHOOL  
Monterey, California


Rear Admiral Isham Linder  
Superintendent

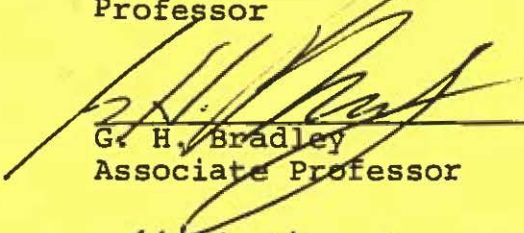
Jack R. Borsting  
Provost


This work herein was supported by the Naval Air Development  
Center, Warminster, Pennsylvania.

Reproduction of all or part of this report is authorized.

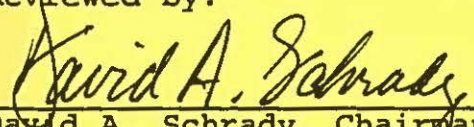
Prepared by:

  
\_\_\_\_\_  
N. F. Schneidewind  
Professor

  
\_\_\_\_\_  
G. H. Bradley  
Associate Professor

  
\_\_\_\_\_  
G. T. Howard  
Associate Professor

Reviewed by:

  
\_\_\_\_\_  
David A. Schrad, Chairman  
Department of Operations Research  
and Administrative Sciences

Released by:

  
\_\_\_\_\_  
Robert Fossum  
Dean of Research

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS55SS75072	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYSTEM TEST METHODOLOGY VOL. I		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) G. H. Bradley                      G. W. Montgomery G. T. Howard                      T. F. Green N. F. Schneidewind		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, Ca. 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS N62269/75/RQ/02014
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Air Development Center Warminster, Pennsylvania		12. REPORT DATE July 1975
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		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)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  The Naval Postgraduate School has conducted a research project during the period 30 June 1974 to 30 June 1975, entitled System Test Methodology under the sponsorship of the Naval Air Development Center. A progress report was submitted on 15 January 1975. The purpose of this project was to develop a methodology and tools for conducting system tests of avionics or other complex hardware/software systems.		

## 20. Continuation

Two areas which received major emphasis were prototype testing and maintenance testing. A methodology for conducting prototype tests was developed. In addition, a simulation model was prepared for aiding the designer and tester in identifying and diagnosing faults which may occur during prototype testing.

A maintenance testing methodology, which involves the use of tests to partition faults into subsets, was developed for identifying faults. In addition to the above areas, research was undertaken to develop models for investigating the relationship between error detection capability and program structure in computer software, using simulation and analytic approaches. The models would be employed during software design for identifying program structures with poor error characteristics and during test planning for the purpose of allocating test resources in accordance with error characteristics.

In addition to the progress and final reports, computer program source decks and operating instructions for the system (prototype) test simulation and the software error detection (simulation and analytic) models have been provided to NADC.

Lastly, three national conference proceedings publications and presentations and two Master of Science in Computer Science theses have resulted from this research project.

## TABLE OF CONTENTS

I.	Introduction	I-1
II.	Prototype Testing	II-1
III.	Maintenance Testing	III-1
IV.	Software Error Simulation	IV-1
V.	Analytical Results for the Error Detection Model	V-1
VI.	Issues in System Testing	VI-1
VII.	Conclusions and Recommendations	VII-1
	List of References	R-1



## INTRODUCTION

The Naval Postgraduate School has conducted a research project during the period 30 June 1974 to 30 June 1975, entitled System Test Methodology under the sponsorship of the Naval Air Development Center. This is the final report of the project. A progress report was submitted on 15 January 1975.

The purpose of this project was to develop a methodology and tools for conducting system tests of avionics or other complex hardware/software systems.

Two areas which received major emphasis were prototype testing and maintenance testing. These topics are covered in Section I and Section II, respectively. A methodology for conducting prototype tests is described in Section I. In addition, a simulation model is presented for aiding the designer and tester in identifying and diagnosing faults which may occur during prototype testing. A description of this model is contained in Appendix A.

The maintenance testing methodology presented in Section III involves the use of tests to partition faults into subsets, so that the actual fault can be identified. In addition to the above areas, research was undertaken to develop models for investigating the relationship between error detection capability and program structure in computer software. A simulation approach and an analytic approach are described in Section IV and Section V, respectively. The models would be employed during software design for identifying program structures with poor error characteristics and during test planning



for the purpose of allocating test resources in accordance with error characteristics. An example of applying the error simulation model to an actual FORTRAN program appears in Appendix B; directions for use of the model will be found in Appendix C; and a listing of the simulation program is contained in Appendix D. A description of the analytic model computer program appears in Appendix E.

Various issues in testing which are germane to maintenance testing and recovery from errors are described in Section VI.

The major conclusions which resulted from each research effort and recommendations for possible applications and future work will be found in Section VII.

In addition to the progress and final reports, computer program source decks for the system (prototype) test simulation, the error simulation model, and the analytic error detection model have been provided to NADC.

Lastly, three national conference proceedings publications (References 9, 12, and 17) and presentations and two Master of Science in Computer Science theses have resulted from this research project.

## II. PROTOTYPE TESTING

### A. MOTIVATION FOR SYSTEM TEST METHODOLOGY

Software is the major expense in computer systems today. As an example, the Air Force allocated between one billion dollars and one and a half billion dollars in 1972 for software development. This was about three times the annual expenditure on computer hardware and accounted for four to five percent of the Air Force budget for the year. Boehm [10, 11] indicates that these high figures are representative of the industry as a whole. He predicts that by 1985 software expenditures in the Air Force will account for ninety percent of the total ADP system costs. Of this enormous amount of money spent on software, a disproportionately large share was spent on testing and the trend is not one of improvement. Boehm states that "during the 1970s the Air Force can expect to spend almost half of its software budget for military space operations on the checkout and test phases of computer program implementation: two to three times as much as it will pay for having the program coded." With such an effort invested in testing software, it should be relatively error free but this has not been the case historically. The Apollo Manned Spaceflight Program had one of the most tested systems in the world, yet major software failures occurred in Apollos 8, 11, and 14. The failure on Apollo 11 occurred in the extremely critical phase of lunar landing. The situation is no better in other areas; each new release of OS/360 has approximately 1000 new software errors. It is not necessary to look at such large complicated systems to discover that present testing is inadequate. The person who has not had an encounter with a computer program error such as an incorrect billing is an unusual person in

today's society. Since testing consumes such a large proportion of the resources allocated to system development and has produced such poor results, it is time to develop a new approach to system testing.

## B. TESTING PROBLEMS

### 1. Multiplicity of Testing Activities

Many of the terms used in the area of testing are subject to a wide variety of interpretations. The word "testing" has been misused and many non-testing activities have been associated with the word. Testing may be defined to be the process of determining if a system meets the stated functional specifications. Quite often debugging is thought of as a testing activity. This is incorrect. Debugging starts with a known error and works towards a correction [13]. Recently, a significant body of literature and activity have been addressed to designing computer programs in a structured fashion in order to eliminate or minimize the occurrence of software errors [14, 15]. The theme of some of these efforts is that if we design programs correctly through structured programming, there will be very little need for testing. Although these efforts do a lot to reduce the potential for errors, they do not act as a substitute for testing.

Other testing activities include verification, validation, certification, proof of correctness, and performance testing. Hetzel [13] discusses these activities in relation to program testing. Verification is concerned with the program's logical correctness based on execution of the program in a test environment. Validation is concerned with the logical correctness of a program in a given external environment. Certification implies an authoritative endorsement that a

program is of a certain quality. A proof of correctness deals with the logical correctness without regard to the environment. Performance testing involves an evaluation of the performance properties of a computer program or system, such as resource utilization. Each of these activities has much to offer. The problem arises when one of the approaches is assumed to equate to complete testing. It is clear that improved software quality must be approached from several fronts: improved design techniques, improved programming management and improved methodology.

## 2. Test Design

There are many fundamental questions that must be answered in designing a test of an information processing system. One such question is what should be tested? Too often a tester ends up testing an incomplete or modified version of the system that is easier to test than the real system. Often the tester is faced with a large set of input combinations to be tested. In this case, the question becomes: How can a subset of the test inputs best be selected to thoroughly test the system? Another important issue is how should the test efforts be organized? It is important to obtain the most information about the system from every test run. It is important to establish test data recording procedures at this time in order to insure that all error information will be recorded. This can be accomplished by properly organizing the tests in a logical sequence. Tests should be related to types and sources of errors. Gruenberger [16] states that "part of the art of testing is knowing when to stop testing." This exposes a two sided question the test designer must

face: When is the test finished and what can be said about the system when testing is stopped?

All these questions are further compounded by the fact that there can be no set rule. Every system requires an original test procedure designed to fit its special requirements. Gruenberger suggests "that the intellectual effort to test a program is of the same order as that which created it."

This section presents a test methodology that will help answer these questions. A model is presented that will serve as a framework for the construction of a logical approach to system testing.

#### C. A MODULAR APPROACH TO PROTOTYPE TESTING

A modular approach to prototype testing offers many advantages for the design of the test and the development of the system. The modular design involves breaking a large system into many small parts called modules. The intra-module functions are independent; however, modules interact by means of standard interfaces. Each module performs a major function of the system.

Modularity improves system design and software portability. To an extent, modules may be transferred among machines and operating systems. With standardization of modules, they may be shared among many applications. With modules being shared in this manner, the programming effort is reduced and the reliability of modules is increased since the modules will be tested with each application. The modules may be expanded more easily and changes are easier to incorporate since the effect of a change is localized.

Testability is significantly improved when a modular approach is used. Testing of different modules may be carried out in parallel.

Standardization of modules yields a set of assertions that may be used as test criteria for the modules. Modules may be compiled separately and can be stored in a program library and accessed independently. Modularity allows testing early in the construction of a system. Each module may be tested as soon as it has been constructed instead of waiting for the whole system to be completed before starting to test. Since modules may be reused in future systems, future programming and testing efforts are reduced.

A modular system was chosen for the prototype test model in order to take advantage of the above desirable properties of modularity.

## MODEL DESCRIPTION

### A. THE FUNCTIONAL MODULE CONCEPT

#### 1. Module Definition

When representing a system with the functional model, the lowest element of the system is the module. Since the word module has had wide use throughout the computer industry, it is necessary to completely define the application of the word as used in the model. A module is an entity that performs a function within the system. A function is an activity performed by the system such as a fast Fourier transform. The physical embodiment of a module is the wiring and circuit boards of hardware, or the source or object programs recorded on punched cards or magnetic tape or programs resident in memory, for computer software. By defining a module in terms of functions, a module is freed from the distinction of being only hardware or software.

A module receives inputs and transmits outputs across a boundary. A boundary consists of a location within the system at which the inputs to a module or the outputs from a module may be measured. In order for the tester to assess these inputs or outputs the boundary must be identifiable. In order to accommodate this requirement for an identifiable boundary, it is necessary to consider the composition of modules. The composition of two modules would be a module performing the same functions as the original two modules. For example, one module might be a fast Fourier transform and the other a digital filter module. If it is impossible to identify a point to measure the output from the filter module to the Fourier transform module, the two could be considered as one module that performs the functions of filter and transform. Thus, the entire system could be viewed as a module or a module could be considered to be a small unit of program code. The proper level for identifying modules will be indicated by the functions performed by the system.

A module will be assumed to be free of internal errors for system test purposes. This assumption is predicated on the fact that all modules will receive extensive individual unit testing before the system is assembled. If an error still exists within a module, the test system will detect it only as the error affects intermodule communication. Assuming that the test plan is sufficient to detect all errors external to a module, the only way an error could go undetected would be if its actions were confined to the module itself.

The system may now be described as a collection of modules which has external inputs and external outputs. The selection of modules must be such that every portion of the entire system is

represented by a module and no portion is represented by more than one module.

In performing its function, the module utilizes system resources. These resources may be in the form of data, control signals, or physical resources including both hardware and software units. Thus a resource is an element of the system that is used by modules in performing a function of the system. Resources have two types of attributes. One type deals with the usage of the resource, which is the amount or size of the resource that is assigned or available to be assigned. The other type deals with resource contents, such as the contents of a memory location or the value of a particular control signal. Resources have states. These states indicate the status of the resource. Some examples of the state of a resource are: reading, writing, idle, file empty, file half full, or memory region assigned.

## 2. Task Definition

The work to be performed by a module may be represented as an ordered or random series of tasks. Tasks are the sub-functions performed by a module. A sub-function consists of a step in the algorithm which the module must execute in order to carry out its function. Examples of tasks are the computation of a simple function, storing the result in memory and outputting the result to the printer. This usage of the word task is synonymous with the use of the word "process" as it is used in the operating system literature. The precedence of tasks is determined by the algorithm the module must execute. These precedence constraints may be linear or they could include branching with or without cycles. It is also possible to have no precedence



constraints. In this case any task could be executed whenever the resources were available.

In order to execute a task, the module goes through a series of states. The state of a module is the status of the module at a given time. A partial list of states that a module can enter includes: compute, wait for memory, wait for input/output, wait for CPU, idle, input processing, wait for another module to complete a task, wait for a resource, and interrupted state. The particular state of a module is a function of the set of inputs to the module, resource states, and its previous state. The outputs of a module are a function only of the state of the module. A primary state is a state that a module is required to enter in order to perform a task. Primary states include compute, input processing and output processing. A secondary state is a state in which the module accomplishes no work. Examples of secondary states would be blocked state, wait for input or wait for CPU. The system state is the set of module states. The system state changes when one or more modules changes state.

### 3. Model Notation

The following is a list of symbols used to describe the model. Each symbol is followed by the definition of that symbol as it is used in this system of notation.

- \*  $i$  ----- Module designation,
- \*  $j_i$  ----- Current state of module  $i$ ,
- \*  $k_i$  ----- Next state of module  $i$ ,
- \*  $I_{ij t}$  -- Vector of inputs at module  $i$  when module is in state  $j$  and input starts at time  $t$ ,
- \*  $O_{ikt}$  - Vector of outputs from module  $i$  after the module has transitioned to state  $k$  and output starts at time  $t'$ ,

- \*  $T_{ijk}$  -- Time at which transition of module  $i$  from state  $j$  to state  $k$  occurs,
- \*  $\Delta T_{ij}$  -- Amount of time which module  $i$  spends in state  $j$ ,
- \*  $R_{ij}$  --- Set of resources used by module  $i$  when in state  $j$ ,
- \*  $(l_1, l_2, \dots, l_n)_{ij}$  -- State of  $n$  resources when module  $i$  is in state  $j$ ,
- \*  $(t_1, t_2, \dots, t_n)_{ij}$  -- Time which module  $i$  uses  $n$  resources when in state  $j$ .

#### 4. Model as a Directed Graph

It is possible to represent a system as a series of directed graphs. One graph would be required for each module. The nodes of the graph would represent module states and the arcs would represent state transitions. Other information could be portrayed on the graph. The state dependent information could be associated with the node. This would include the current state of the module, the set of resources used by the module in that state, the state vector for the resources used by the module, the vector of inputs to the module, the vector of outputs from the module and the amount of time the module spends in the state. The arcs could be labelled with the time that the module requires to transition from the source state to the destination state as is shown in Figure II-1. In this figure, the module  $i$  transitions from state  $j$  to state  $k$  at time  $T_{ijk}$ .

These directed graphs would give the tester a convenient means of visually representing the activity of the module. The tester might prefer to show only the primary states of the module and the idle state instead of showing all possible states of the module.

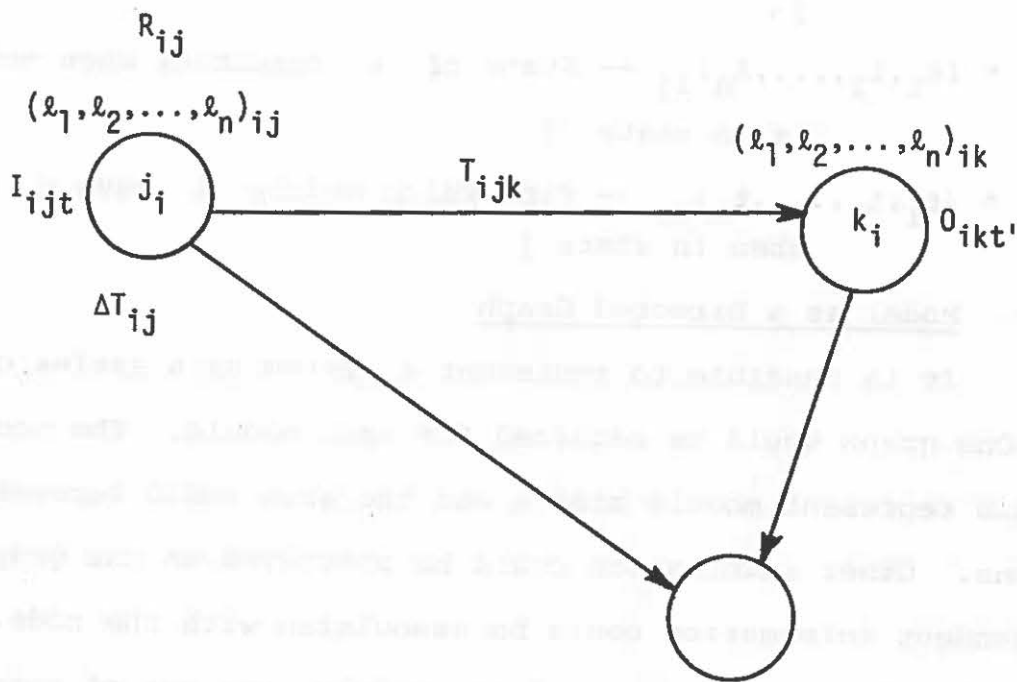


Figure II.1. Directed Graph of Module States

## 5. Time Domain of a Module

A property of a module is that it uses the resources of the system only at certain times. One of the major problems of testing computer systems is to identify when two or more modules will be competing for the same resources. The problem is further compounded if the system possesses multiple CPU's which are running asynchronously. The concept of time domain will be useful to address this problem area. A time domain of a module consists of the times that resources are in use. A graph of the time domains of the modules of the system would be a useful abstraction of the system for the analysis of the timing problem. The resources of the system could be represented on the vertical axis with time expanding along the horizontal axis from the origin. Each area so represented should be labelled with the module and the amount of the resource required. The time domain of a module would be represented by the summation of the areas formed by the product of resources used by the time duration of use. Any intersection of time domains would represent a potential error only if the total demands of the modules exceed the maximum resources available.

One problem with this representation is to find a timing system that applies to all modules when modules are operating asynchronously. In this case the time axis would be the elapsed time from some critical event in the system. The changes in system state would be referenced to this event.

If we define a change in system state as any change in module state, it is possible to consolidate the module state representation into a system state representation and show resource usage conflicts in terms of system states as indicated in Figure II-2. In this figure

### SYSTEM STATES

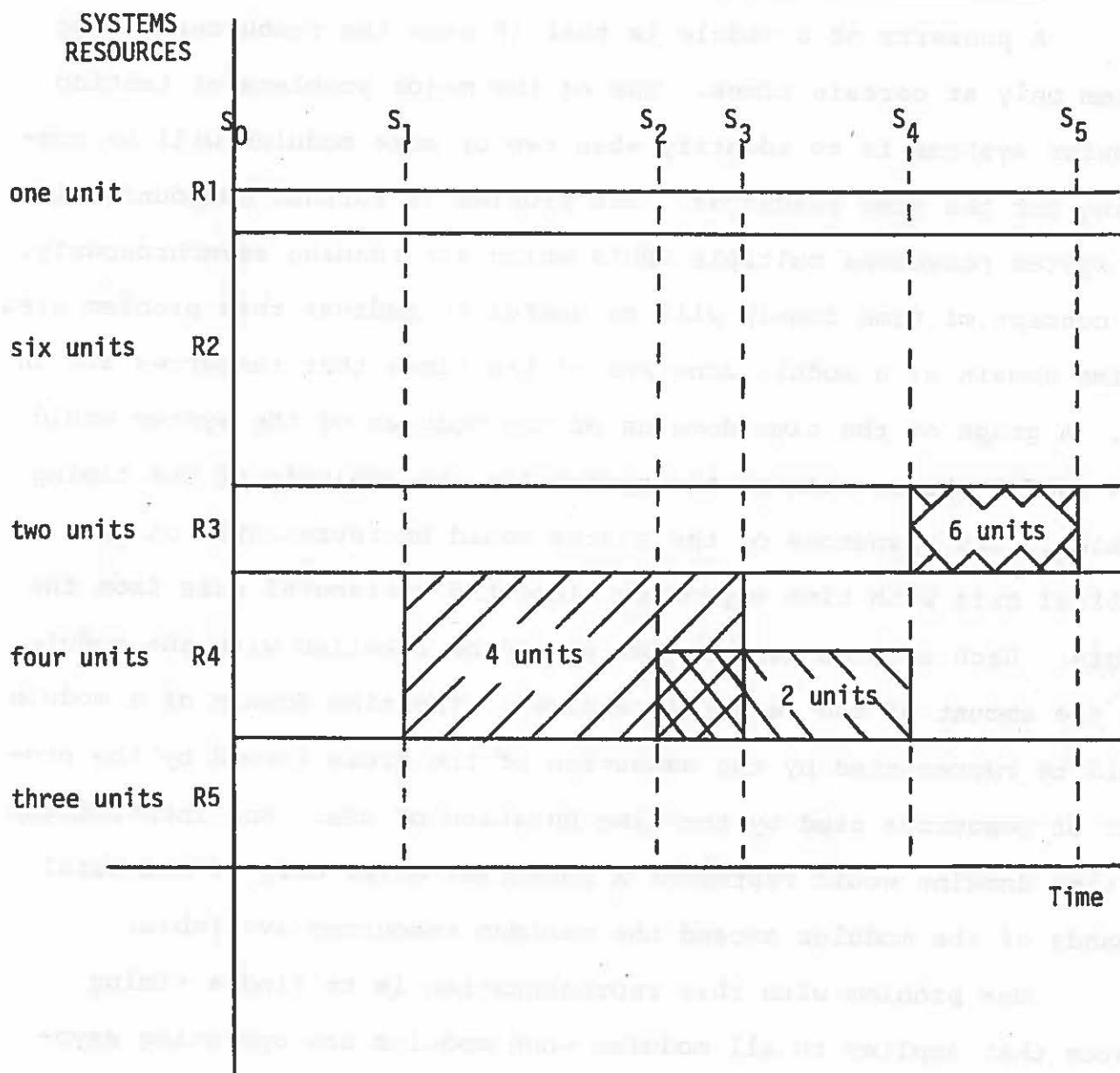


Figure II.2. Resource Conflicts vs. System State

there are five types of resources available to the system. They are labelled R1 through R5. The amount of each resource is indicated on the vertical axis. For example, there are six units of R2 available. There are two resource conflicts portrayed in this system. One occurs in system state  $S_2$ . Here one module requires four units of resource R4 and another module requires two units of R4. The conflict occurs because there are only four units of R4 available. The conflict is denoted by a cross-hatched area. The other conflict is in system state  $S_4$ . A module has requested six units of resource R3 when only two units are available to the system.

The construction of such a graph would be infeasible to do by hand for a real system. A program could be written to produce this type of graph from the time domains of the modules. On this graph the computer could identify resource usage conflicts.

## B. APPLICATION OF MODEL TO TESTING

### 1. Functional Specifications

One of the more difficult processes in producing reliable software is translating user requirements into meaningful design specifications. Boehm, McClean, and Urfrig [4] vividly demonstrate the magnitude of the problem in their study of a large software project. The authors divided errors into two classes. These were design errors and coding errors. An error was considered a design error only if its correction caused a corresponding change to the design specifications. Of the total errors, 64 percent were design errors. This alone is enough to illustrate the need for a valid method of design specification. Even more disturbing was the time frame within the testing in which

the errors were discovered. Of the 54 percent that were not discovered until the acceptance, integration or delivery phases of testing, 45 percent were design errors. The remaining nine percent were coding errors. Errors discovered in these latter stages are more difficult to correct than those discovered during the coding stage. Thus, it is necessary to have a good system of describing design specifications. The functional model provides such a system.

When the functional model is used, the user should be required to define all functions of the system. The functional specifications would consist of a statement of the activities of the system and the associated inputs and outputs. By requiring functional specifications, designers are assured of having a complete detailed description of the system at the beginning of the project. This should reduce the number of design errors.

It is possible to over specify the design of a system. This could prevent the designer from choosing the most efficient method of designing the project. It could also introduce errors into the system design, if the user does not have a thorough knowledge of computers. This problem is avoided by using functional specifications. Details are presented as functions of the system, which is the area in which the user is most knowledgeable. The implementation of the functions is left to the designer, who is in a better position to determine the proper method.

Another pitfall of system design may be avoided by using functional specifications. Frequently, test specifications are not available early in a project because testability is not considered to be a design parameter. Instead, test requirements are formulated as an

afterthought when it is too late to influence the design [5]. Functional test specifications are defined as test specifications which are based on testing the stated functions and observing the corresponding outputs of the system. Functional specifications should be incorporated in the test specifications. Detailed design should not commence until this information is available.

## 2. Documentation

The need for complete and usable documentation should be a primary concern of anyone involved with system design, programming and testing. Poole [6] states "that the lack of good documentation usually means that testing is not performed as thoroughly as it should be and debugging is that much more complicated." Another use of documentation is for the maintenance of the system. Since the life of a system is much longer than the development phase, the designers will probably not be available to help maintain the system. In addition, many people may have access to the software. All changes which result must be documented.

The use of the functional model helps to provide adequate documentation throughout the life of the system. The concept is to force documentation to be an integral part of system development. Two documents have already been discussed. These are the functional specifications and the functional test specifications. These documents should form a segment of the documentation. These should be systematically updated as changes are made to the system.

The documentation should include other information as well. This could include a data base containing information about all errors



that were found in the system to date. Unfortunately, there is a tendency to ignore this aspect and to think of this type of information as something to discard once the error has been corrected [6].

Every incident must be recorded because an outage that may appear insignificant to the user could be an important indicator once it is properly analyzed. The data base could be used to identify modules that are the source of the majority of errors. This classification could be used to direct future testing and debugging. It could also be used to determine which modules are the most unreliable. This would provide a starting point for improving the reliability of the system. This would be particularly applicable if the module that is most critical to the system's operation is also the most unreliable. The data base could also be classified as to type of errors. This would be valuable information when designing a similar system.

Another form of documentation that should be incorporated into the plan for system testing is assertions. These are statements that are introduced into the code by the programmer. These state a fact about the design of the program. These statements may be treated as a comment card or used to produce code to check for the validity of the assertions. The appropriate action would be determined by a parameter passed to the compiler. Two types of assertions could be employed within the model. The first would be global assertions. These would be in the form of specifications for intermodular actions of the system. An example of such an assertion would be:

```
ASSERT RANGE OF ALL ARRAY INDICES IS 0 TO 100.
```

The other level of assertions would be local. The local assertions would be defined by the programmer but within the design specifications.

An example of a local assertion would be:

```
ASSERT RANGE OF I IS 10 TO 20.
```

These assertions could be a permanent feature in the program. They could be activated on the local level to help test a module or on the global level to aid in introducing a change to the system. As such, these assertions would form an important part of the system documentation.

### 3. Test Inputs

Ideally, it would be proper to exhaustively test a system. This implies that every path in the logic of the program be executed and tested. Shooman [8] demonstrates that this will normally be impossible due to the large number of inputs required. The problem presented involved exhaustively testing an assembly language program which solved for the roots of a quadratic equation  $Ax^2 + Bx + C = 0$ . The computer was assumed to have a 12 bit word length and integer arithmetic was used. All syntactical errors had been eliminated and all known special cases such as  $A = 0$  and imaginary roots had been accounted for. The input space to exhaustively execute this program involved  $64 \times 10^9$  combinations of A, B, and C. The program had a run time of 240 microseconds per execution. The time to complete the entire execution of the program over the input space would have been approximately 5,000 hours. To test a program, solutions must be verified by some independent means such as a desk calculator or a different algorithm. This should be done in as many different ways as possible, since there is some probability that two independent approaches will result in the same wrong solution. Obviously, exhaustive testing is infeasible for even a small program.

The problem the tester must solve is how to best select the subset of test inputs from the universe of possible inputs. A method for selecting the inputs for a test is to first identify and rank the modules in a system by the criticality of the modules to the mission success. It is seldom the case that all modules are equally valuable. A technique for determining criticality is to ascertain the consequences to the mission of a module malfunction. A malfunction in some modules would cause a mission abort, while others would result in a degraded mode of operation. The modules are ranked according to criticality. This is based on the criticality of module outputs. The time spent in testing each module can then be allocated using this ranking. The time allocation can be further refined by ranking the criticality of each sub-function of the module. This would be based on the criticality of the sub-function to the performance of the function by the module.

There are other factors that can be used to rank modules for testing purposes. One such criteria would be forecasted errors. Schneidewind [9] has developed a model of the occurrence of errors detected during functional testing of command and control software. It would be possible to rank modules in order of forecasted errors. Work is progressing in the area of developing relationships between program structure, program complexity and the ability to detect errors in a program [12]. Another method of obtaining such a ranking would be through the use of simulation. Critical modules could be identified by their high rate of failure in the simulation.

Once the amount of testing resources allocated to each module has been determined, the proper number of inputs for testing each

module can be estimated. The problem then becomes one of selecting the inputs to thoroughly test each module. The module represents a function which maps the set of inputs into the set of outputs. The inverse mapping could be used to obtain the set of inputs. Given this set of inputs, test cases are selected in order to cover the input set and the program as thoroughly as possible. Particular attention must be given to inputs that are involved in the control flow of the program. Once this has been done, unusual cases are investigated. A possible source of unusual cases would be indicated by the set of inputs. Values are picked that are combinations of the extremes of the range of inputs.

#### 4. System Representation

Having developed the notion of a module, it is necessary to investigate the method that will be used to represent a system as a collection of modules. A system is comprised of asynchronously operating application software modules, hardware modules and executives. Figure II-3 gives a generalized representation of a processing system. The system represented in this figure is comprised of two asynchronously operating executives, A and B. These are connected to two separate control buses noted by Control Traffic Bus A and Control Traffic Bus B. Each bus connects the application software modules and hardware modules that are controlled by the executive on the bus. An example of the traffic on this bus is a hardware generated interrupt occurring at the conclusion of an input/output operation. A subsystem is comprised of one executive, the modules that it controls, and the control bus

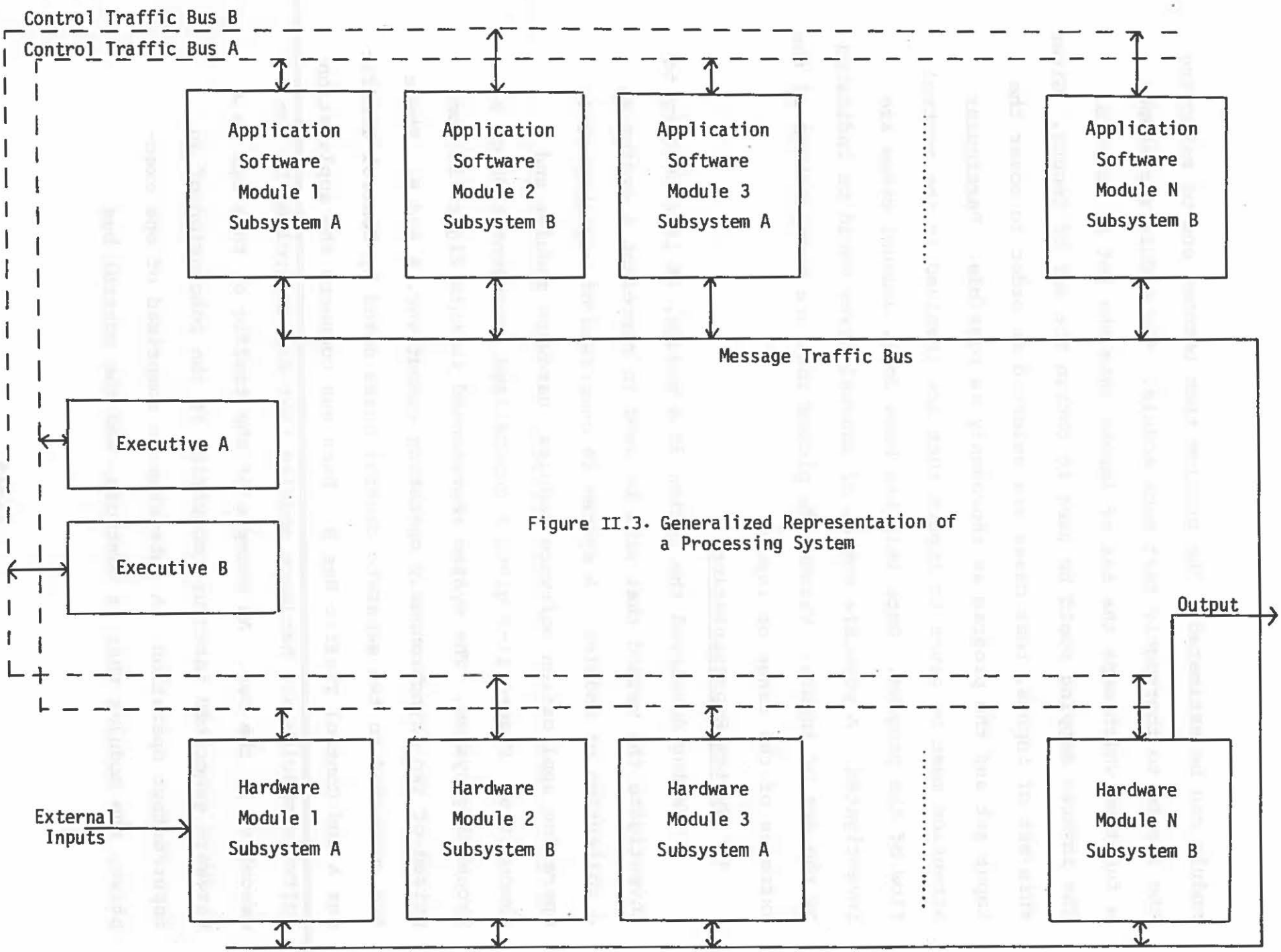


Figure II.3. Generalized Representation of a Processing System

connecting the modules to the executive. There is a message traffic bus connecting all modules. An example of the traffic on this bus is a module passing a computed value to another application module. External inputs and outputs are identified.

This representation of a system has many useful applications to testing. The model may be used to verify the correct functioning of two types of intermodule communication. The first is message traffic. The traffic on the message bus could be checked against the functional test specifications for correctness. The second concerns control traffic. The traffic on the control buses could be checked in a similar manner. Other problem areas that could be investigated using the model include:

- \* Are the various state transitions possible, based on the values of the resource states?
- \* Are there any blocked or deadlocked states?
- \* Are the amounts of time in each state excessive?
- \* When a module state transition occurs, are the resource state vectors correct?
- \* Are there times that a module holds resources excessively?

## SIMULATION

### A. A SIMULATION OF THE MODEL

A simulation of the model was constructed. The simulation was an event store type of simulation. It was written in FORTRAN IV to run on the Naval Postgraduate School's IBM 360/67. The simulation used the model representation with the user providing a description of the system to be simulated. This description required the number of

modules, the number of tasks, the precedence among tasks, number of resources and resource usage. A complete description of the simulation appears in Appendix A.

The simulation showed that the model could represent a system. A simulation of this nature could be useful in testing.

## B. USE OF SIMULATION IN TESTING

### 1. Investigation of Timing Problems

Timing problems are extremely difficult to investigate in a real system due to the fact that any test equipment installed internal to the system disturbs the timing of the system. Equipment installed external to the system may not be able to gain the required information either because of synchronization or access problems. By using a simulation of the system, the tester may observe various timing parameters. The tester is able to observe timing problems that could not be observed on the real system. This is accomplished without disturbing the timing of the real system.

Another problem area that could be investigated through the use of simulation is the reaction of the system to various rates of input. In the simulation it is possible to vary the mean time between arrival of inputs. This parameter could be decreased on each run to determine the maximum input rate that the system could receive and still process an acceptable number of inputs. Another method would be to plot average time to process a complete input versus input rate. This graph could be used to determine an acceptable range of input rate. This method of analysis could be used when testing a system that has to produce periodic outputs, such as a system with

a graphic display that has to be refreshed at a specified rate.

If the time that a module spends in a particular state is expressed as a variable instead of a constant, a simulation would be an invaluable aid to the tester in investigating the operation of the system. One approach would be to observe the operation of the simulated system with all modules functioning at the maximum time duration. Another method would be to use various combinations of module operating times to determine under what circumstances the system would fail or performance would be degraded. This can easily be done on a simulated system but would be impossible to do on a real system because the tester would be unable to control the time a module spends in a state.

Another timing problem facing the tester is the system clock rate. Often the tester would like to slow the system down or perhaps speed it up in order to observe some particular action of the system. This would be important if the tester was unable to measure the output of a real module because another output arrived before the first output could be measured. In a real system, it may be impossible to change the timing of each component of the system by the same amount. This would be particularly difficult in a multi-executive system. With simulation, the tester is able to adjust the timing of the system.

Some problems do not occur until the system has processed a large number of inputs. The tester may not be able to cycle the real system through a large number of inputs due to lack of time or equipment availability. However, in a simulation, the time scale may be greatly compressed, allowing the tester to cycle the system many



times. This would greatly increase the probability of discovering latent bugs. Similarly, in a simulation the user's ability to specify the initial state of the system allows starting tests under some arbitrary condition that might only be achieved in an actual system by running for a long period.

## 2. Fault Insertion

Dijkstra [1] contends that "testing can only determine the presence of errors, not their absence." One approach would be to know the reaction of the system to every possible error and combination of errors. Using this knowledge, one could simply observe the reaction of the system and state what errors were or were not present. Unfortunately, the set of every possible error, combination of errors and system reaction is an immense set. Therefore, it is impracticable to prove the absence of faults by using the above approach. However, this approach using simulation, could be used to greatly expand the subset of errors that the tester could detect.

The tester may purposely introduce a fault into the simulated system. The reaction of the system to this fault could be catalogued for later reference. This information could be used to identify modules that are affected the most by a class of errors. This set of modules would be noted for special testing. This information could also be used to ensure the validity of the test plan. If the group of tests included in the test plan did not encompass the reactions observed in the simulation, then the tests would not be able to detect particular faults.

## 3. Partial System Simulation

Frequently the tester will not have the time, assets, or motivation to perform a simulation of the entire system to be tested. In this situation, simulation of certain parts of the system may be

desirable or the tester could choose to simulate the entire system in less detail. Campbell and Heffner [2] relate a case history illustrating this point. A simulation model was constructed of a system being developed. The skeleton system was working before the model was debugged. When the model was finally working, no one was certain which version of the real system the simulation results were meant to represent. However, some of the designers used simple simulations that they developed to study certain aspects of the system. The authors concluded that "ambitious large-scale models generated by professional model makers are less helpful than simpler work done by the system developers themselves." A simulation with less detail was more useful in this case than a complete simulation.

Quite often in prototype testing, a module or modules will not be present when the tests are scheduled to commence. This could be due to late delivery or to a module being modified after preliminary testing proved the module needed modification. This could also be caused by a planned action such as phased delivery. A simulation of this module would allow the tests for the rest of the system to continue. Simulating the missing module would be particularly easy if the system had been described in the form of a functional model. If all the information required to functionally represent the model is present, then a simulation can be constructed from this information.

Another use of simulation involving less than the whole system is the use of a test data generator. When a system is tested in the laboratory, it may be necessary to simulate the inputs to a system. Since there is no reason to believe that all modules will be present during the entire test phase, the tester may desire to have the test

data generator simulate the output from any module. Thus, the test data generator could also substitute for any missing module as the system was being tested. Not only would the generator act as an output generator, but it would also act as the termination for module outputs which are intended for missing modules. If these outputs must be accounted for, because the operation of the partial system would not be entirely representative of the operation of the complete system. Notice that this procedure is applicable to a top down testing approach because the test data generator could simulate inputs from dummy modules.

#### 4. Pitfalls of Simulation

After having spent much time and effort to develop a simulation, the tester may find that the simulation addressed the wrong problem or solved no problem at all. The validity of a simulation is the consistency between the simulation and the real system it represents. Proof of the validity of a simulation is almost impossible, especially if the real system has not been constructed. By the time the real system has been constructed and the validity of the simulation has been disproved, irrevocable decisions may have been made based on test data from the simulation.

Although validity is a major problem in simulation, it is by no means the only problem. A list of problem areas that may cause a misunderstanding of the system being simulated is presented in Fishman [3]. These include incorrect input parameter specification, influence of initial conditions on data and misuse of estimates. The author provides suggestions on ways to control these problems.

Prototype testing may result in design changes. Each change requires a change in the simulation model. If the tester has not allowed for such an occurrence in budgeting simulation resources, the model would not represent the real system. Also, there would be a time lag in modifying the model. This could have a serious effect on the test schedule if this contingency is not included in the test plan.

## APPLICATION OF MODEL TO PROTOTYPE TESTS

### A. APPROACH

#### 1. Test Plan

In order to apply the functional model to the problem of prototype test it is necessary to develop a test plan. A test plan should be created as part of the design plan. As a minimum the test plan should discuss the following major elements:

- \* define modules,
- \* define module states,
- \* identify inputs and outputs for each module state,
- \* identify module interfaces,
- \* identify tasks,
- \* define resources and resource states, and
- \* identify resource usage for each module state.

The test plan must also include the system functional specifications and functional test specifications. In addition, it should include the test procedures. This would identify acceptance criteria, such as the allowable divergence between desired and actual output values, time duration of tests, allowable number and types of malfunctions,

number and distribution of test replications, and methods for checking test results. The test plan should identify major testing milestones. These would identify major sections of testing that must be completed before system development can continue.

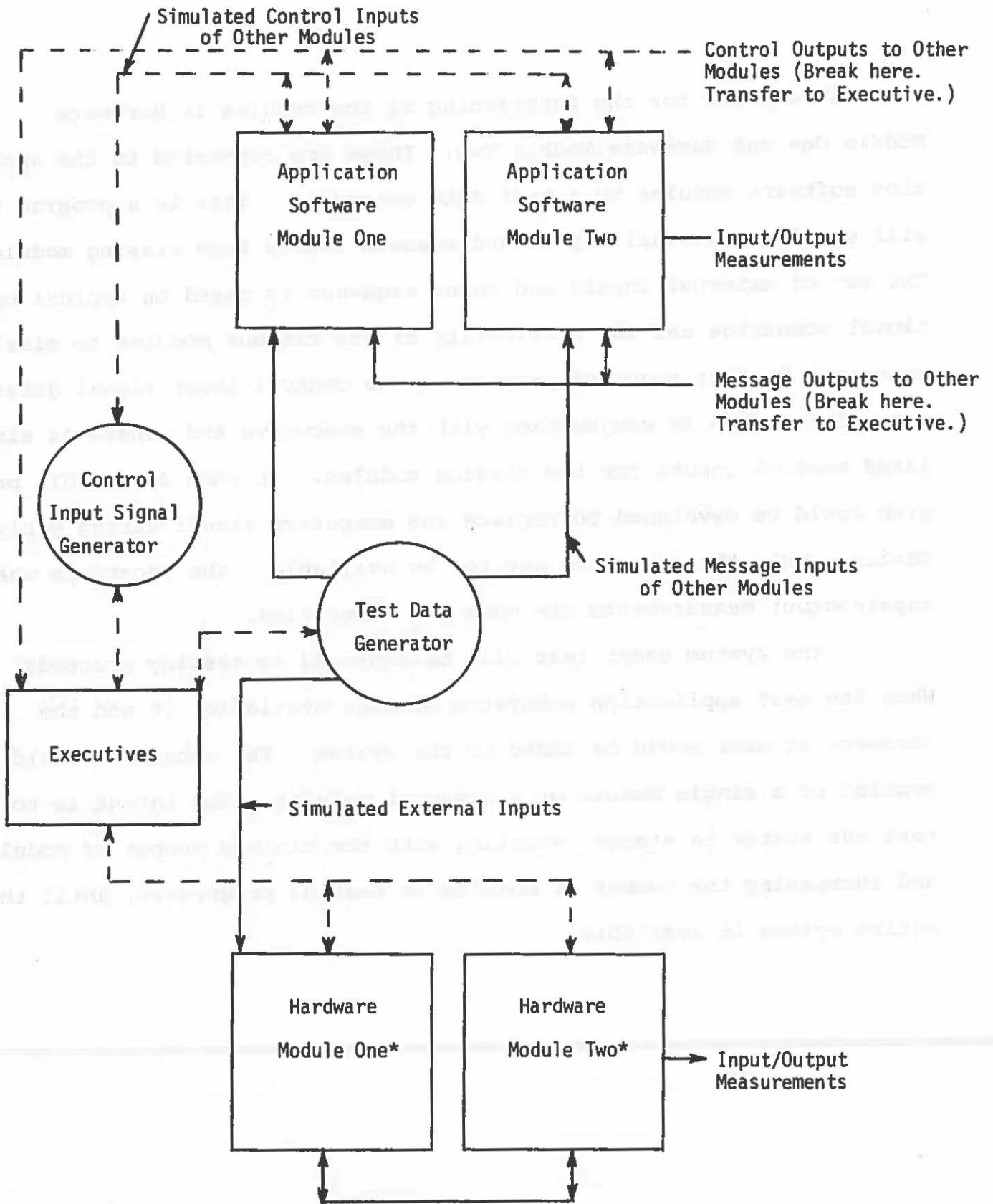
The test plan should document the subsystems that will be tested. This will require the development of a method of isolating a subset of the system to test it without the effects of the remaining system being introduced. These identified subsets of modules are called subsystems and will be used to test the system in stages.

Besides the modules in a subsystem, the test plan must also define a set of measurements which will indicate whether correct outputs are being produced for given inputs and define the hardware and software locations of the measurements. The plan must describe how to instrument the system in order to obtain these measurements.

The test plan should develop some organizational structure. This would include who is to do the testing and the resources to be used in testing. The plan should include who is responsible for maintaining the documentation. This would include test data, error information, design changes and test modifications.

## 2. Subsystem Testing

The modularity of the model allows the user to commence testing at an early date. This will require the testing of a subsystem. The subsystem is defined in the test plan and testing will commence as soon as all the modules of the subsystem are available. This is illustrated in Figure II-4. In this figure there are two application modules in the subsystem that are ready to be tested. The only hardware



\*This is the only hardware required for functioning of application modules one and two.

Figure II-4. Prototype Test Configuration

that is required for the functioning of the modules is Hardware Module One and Hardware Module Two. These are connected to the application software modules by a test data generator. This is a program that will simulate external inputs and message inputs from missing modules. The set of external inputs and input sequence is based on typical operational scenarios and the criticality of the various modules to mission success. Another required program is the control input signal generator. This works in conjunction with the executive and generates simulated control inputs for the missing modules. In some cases this program could be developed to replace the executive itself during early testing, when the executive may not be available. The locations where input/output measurements are made are identified.

The system under test will be expanded as testing proceeds. When the next application subsystem becomes available, it and the hardware it uses would be added to the system. The subsystem could consist of a single module or a group of modules. The intent is to test the system in stages, starting with the minimum number of modules, and increasing the number of modules as testing progresses, until the entire system is available.



### III. MAINTENANCE TESTING

#### A. INTRODUCTION

In this section we focus exclusively on the problem of maintenance testing. By maintenance testing we mean the testing which is done after a system has been released and placed into operational use. This is distinguished from prototype testing which is done on the original or prototype system for the purpose of determining whether the system actually constructed meets the design specifications and performance requirements established in the earlier stages of the development process. Prototype testing is essentially a certification process. Maintenance testing, on the other hand, is directed toward the question of whether a particular copy of a system remains in the same condition as it was when first placed in service.

Our goal is to present a method which might be used as the basis of maintenance testing. The idea of partitioning which we consider here is not a new idea. It has been explored and developed extensively in the context of testing digital circuitry [18], but it has apparently not been examined in the context of systems testing or software testing. After the idea is presented, a discussion of some of the problems in applying the method in a real testing situation is included.

#### B. DISCUSSION OF SYSTEM FAULTS

We define a system fault as any hardware or software condition which causes the system to deviate from its design specification in an observable manner. Observations can be made at several levels from



the component level to the functional level. In this section we are not concerned about the level at which observations are made, but we do make some additional assumptions about the nature of system faults and it is important that these be clearly understood.

We assume that the faults under consideration are non transient in the sense that, whatever condition or fault occurs, it remains until corrected. Thus spurious results are not observed in the testing procedure. This means that when given the same initial state of the system the same input test conditions always produce the same test output. Thus the tests are repeatable in the sense that the system being tested is not changing during the test period. In a real testing situation it is often the case that apparently spurious results are obtained. The practical difficulty in reproducing them generally lies in the inability to reproduce the test conditions exactly. This often occurs because a sequence of tests interact. Earlier tests may change memory or write over critical values or otherwise change the state of the system. Thus for purposes of our discussion we will assume that the system under test has a reset capability so that the state of the system is the same before each test. The system may contain a fault, but it contains the same fault until fixed. The effect of this reset assumption is to make each test in a series of tests act independently so that exactly the same information is obtained by applying test 1 and then test 2 as is obtained by applying first 2 then 1.

In our discussion of partitioning we assume:

- a) the set of all faults under consideration can be enumerated. We denote the faults by  $f_1, \dots, f_n$  and we let  $f_0$  denote the condition of no fault,

- b) any fault in the system remains in effect until it is corrected (so that test results are repeatable as discussed above),
- c) the system being tested contains at most one fault,
- d) the system being tested is reset to some initial state before the application of each test.

The most restrictive of these assumptions is probably the first. In a complex system the number of things which can go wrong is immense and, to be able to detect and isolate individual faults, considerable precision is required to distinguish among the many similar faults.

The assumption that the system contains only a single fault when tested can perhaps be justified by assuming that the test procedure is repeated frequently, so that each fault is detected before others occur. This is obviously invalid for massive failures in which a number of faults arise simultaneously from the same cause. On the other hand, if certain combinations of faults are thought to be likely, they can be handled by defining them at the outset as a single fault.

### C. DISCUSSION OF PARTITIONING

We denote by  $T_i$ ,  $i = 1, \dots, m$  the tests which can be applied to the system. It is convenient to think of the maintenance test procedure as being applied to a system containing an unknown one of the faults  $f_j$ ,  $j = 0, \dots, n$ . The purpose of the tests is to determine which of the conditions  $f_0, \dots, f_n$  exists in the particular system under test.

The testing procedure consists of applying a sequence of tests to the system. Each test results in some observable outcome.

We assume that there is a finite set of possible outcomes which we designated  $O_k$   $k = 1, \dots, p$ .

If we had a single test which was powerful enough, there would be a distinct outcome associated with each fault. Such a test would be comprehensive in the sense that no other tests would be required to isolate the fault. Such a test is said to have full resolution. A test of this type would be very extensive and complicated and, although it fits within our discussion here, our thinking is oriented toward less comprehensive tests. Thus we will suppose that the individual tests under consideration do not provide full resolution, but to be useful they must provide some resolution among the faults.

Figure III-1 illustrates the process of applying test  $T_i$  to a system containing fault  $f_j$  with the result that outcome  $O_k$  is

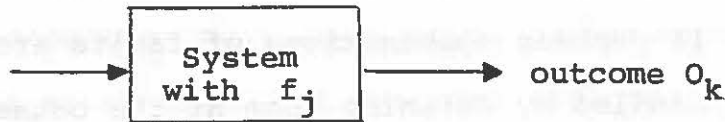


Figure III-1. A Typical Test.

observed. It is necessary to fully characterize the performance of each test  $T_i$  in the presence of the faults  $f_j$ , and we imagine that for each test the resulting outcome is known in the presence of each fault. This is illustrated in Figure III.2.

	$f_0$	$f_1$	$f_2$	$f_3$	$\dots$	$f_n$
$T_i$	$O_3$	$O_1$	$O_2$	$O_1$	$\dots$	$O_3$

Figure III-2. Typical Test Results for Test  $T_i$ .

The data of the type shown in Figure III-2 can be obtained in several ways. These include:

- a) analysis of the system design,
- b) experimentation with a real system,
- c) simulation of the system behavior.

The first method relies on the system designers, engineers, and programmers to determine from their knowledge of the system how it will behave in the presence of each fault under consideration. The experimental method involves obtaining a fault-free copy of the system, inducing the desired faults, applying the tests and recording the results. The simulation method is nearly the same as the experimental method except that the observations are taken not from the real system but from a model of it, probably a computer simulation.

It is not intended that the tests, when applied to a system containing an unknown fault, result in a pass or fail. Some faults produce the same outcome under test  $T_i$  as the fault-free system. For example, the results in Figure III-2 indicate that  $f_0$  and  $f_n$  both produce outcome  $O_3$ , but it would be misleading to apply  $T_i$  to a system, obtain  $O_3$ , and claim that the system passed that test. Actually, the test  $T_i$  is unable to discriminate between  $f_0$  and  $f_n$ .

The application of a single test serves to partition the set of all possible faults into  $p$  mutually exclusive and collectively exhaustive subsets corresponding to the  $p$  possible outcomes.

For notational purposes we denote the set of all faults which produce outcome  $k$  when subjected to test  $i$  by  $S_{ik}$ . Thus when

test  $i$  is applied with the result  $O_k$  we can conclude that the fault actually present is one of those in set  $S_{ik}$ . This is illustrated in Figure III-3.

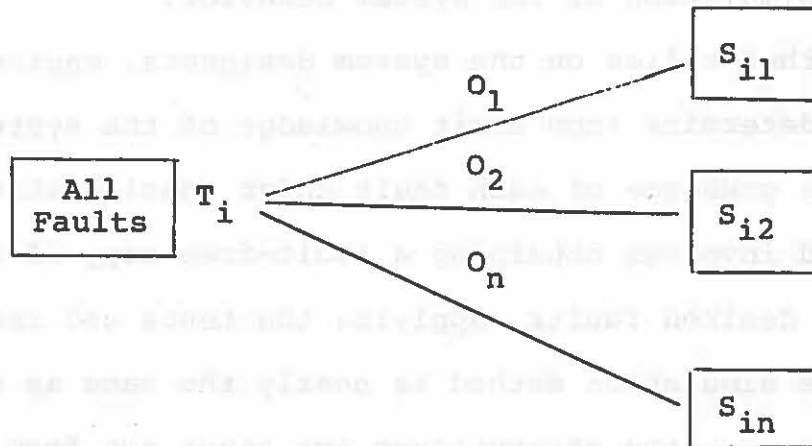


Figure III-3. Partitioning the Faults with Test  $T_i$ .

Further testing can be applied to the sets  $S_{ik}$ . Suppose  $T_j$  is applied. This will result in one of the test outcomes say  $O_r$  with the conclusion that the system under test contains one of the faults which is in both the sets  $S_{ik}$  and  $S_{jr}$ .

The maintenance test problem is to select an efficient set of tests which can successively partition the set of possible faults in smaller and smaller sets so that ultimately the actual fault can be isolated. Possible criteria for test selection will be discussed after the examples.

#### D. SEQUENTIAL AND COMBINATIONAL TESTING

In any testing situation where a sequence of tests is to be applied to a system the question will arise whether later tests in

the sequence are to be selected on the basis of the results from earlier tests or not. The case where earlier test results do influence the selection of later tests is called a sequential testing procedure, otherwise combinational.

Sequential testing is a more powerful method in that fewer tests will generally be required to isolate a fault since the sequential nature of the procedure allows the selection of later tests which are more capable of discriminating among the remaining possible faults. Combinational procedures, although independent of observed test results, may be easier to implement in checking out a software system since less storage space is required to store the tests and less logic required to implement them.

Examples

To illustrate the ideas of partitioning consider the following data used in examples 1 and 2. The table entries 1, 2, and 3 refer to outcomes  $O_1, O_2,$  and  $O_3$ .

	$f_0$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$	$f_7$	$f_8$
$T_1$	1	2	2	1	3	2	1	1	3
$T_2$	2	2	2	2	3	2	1	1	2
$T_3$	2	1	3	1	2	2	3	2	3
$T_4$	1	3	3	2	2	2	3	3	3
$T_5$	2	3	1	3	3	1	3	2	2
$T_6$	3	2	3	2	2	1	1	2	1

Table III-1. Data for Examples 1 and 2.

Example 1: A Combinational Test Procedure

In this example we illustrate the result of applying tests  $T_1, T_2, T_3$  in that order to a system containing one of the faults  $f_0, \dots, f_8$ . The results are portrayed as a test tree. See Figure III-4.

Notice that the application of  $T_1$  partitioned the faults into three sets: The first, associated with outcome  $O_1$ , containing faults  $f_0, f_3, f_6$  and  $f_7$ . Subsequent application of  $T_2$  further partitioned this set into two sets containing faults  $f_6$  and  $f_7$  associated with outcome  $O_1$  and faults  $f_0$  and  $f_3$  associated with  $O_2$ . Notice that  $O_3$  is not possible since the actual fault in this case is known (after applying  $T_1$ ) not to be  $f_4$ . The test  $T_2$  is ineffective in the event that  $T_1$  yields outcome  $O_2$ .

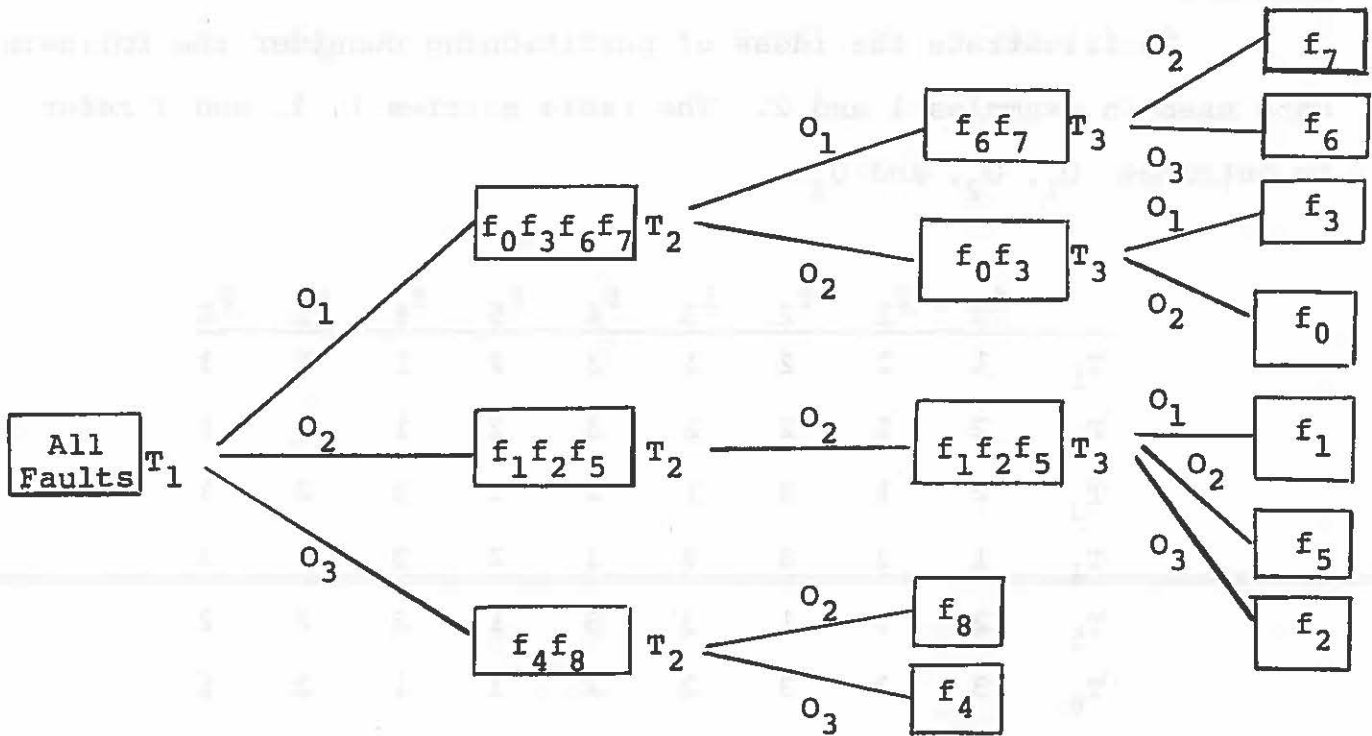


Figure III-4. Test Tree for Example 1.

**Example 2: A Sequential Test Procedure**

In this example different tests are applied depending on the outcome of previous tests. The test tree is shown in Figure III-5. Notice that in this case, particularly when  $T_1$  yielded  $O_2$ , the later tests could be selected to make best use of the information already available.

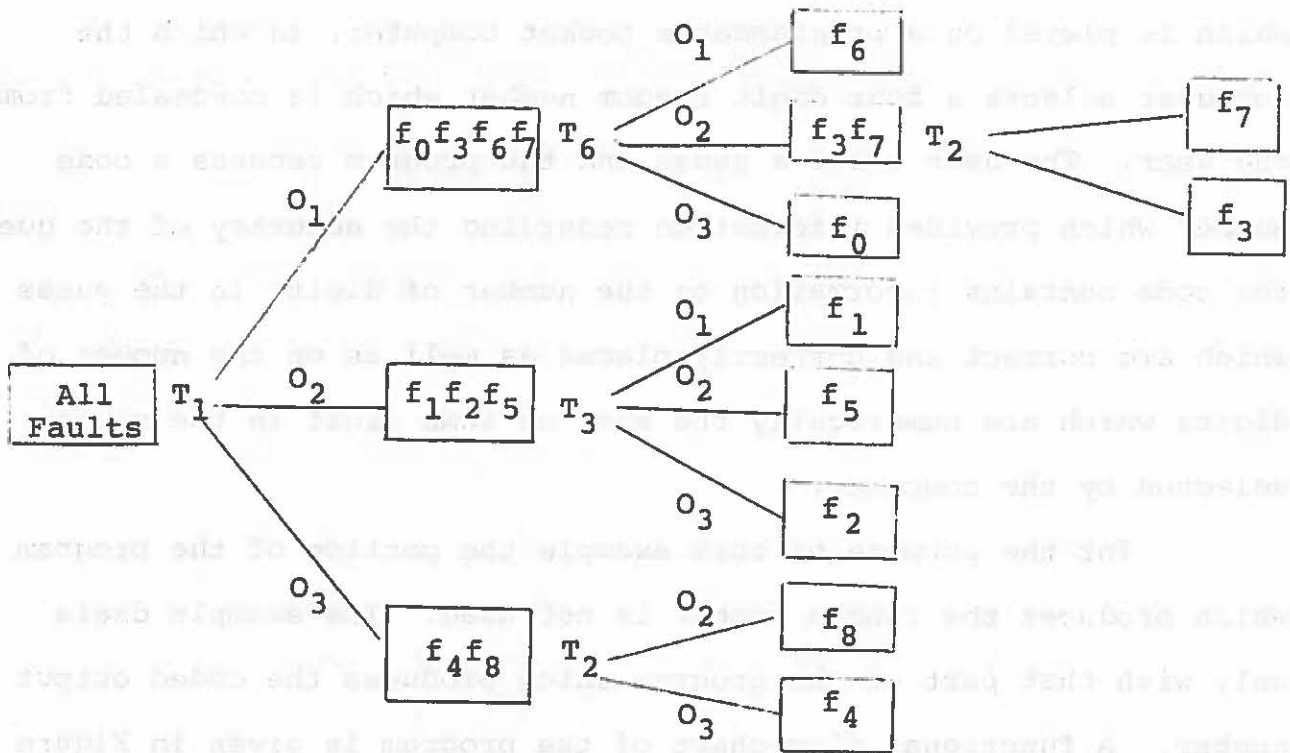


Figure III-5. Test Tree for Example 2.

Reflecting on these possible test trees raises several questions: Should the test procedure be sequential or combinational, how shall possible test sequences be compared and what is the "best"



procedure? For example, if the system is very likely to be fault-free, then a test sequence which begins with  $T_4$  is attractive unless  $T_4$  is very expensive or time consuming. On the other hand  $T_4$  is nearly worthless if the probability of having a condition other than  $f_0$  is large. After considering one more example, we will address these questions of test selection by considering several possible test objectives.

Example 3:

This example illustrates the application of the methodology described to a small program. The program is an interactive game which is played on a programmable pocket computer, in which the computer selects a four digit random number which is concealed from the user. The user makes a guess and the program returns a code number which provides information regarding the accuracy of the guess. The code contains information on the number of digits in the guess which are correct and correctly placed as well as on the number of digits which are numerically the same as some digit in the number selected by the computer.

For the purpose of this example the portion of the program which produces the random number is not used. The example deals only with that part of the program which produces the coded output number. A functional flow chart of the program is given in Figure III-6. The actual machine implementation will not be discussed here. All the subscripts on  $N$  should be interpreted mod (4).

The program assumes the four digits of the actual number selected by the computer are  $N_1, N_2, N_3, N_4$  in that order. The four digits of the guess are  $W_1, W_2, W_3, W_4$ , respectively.

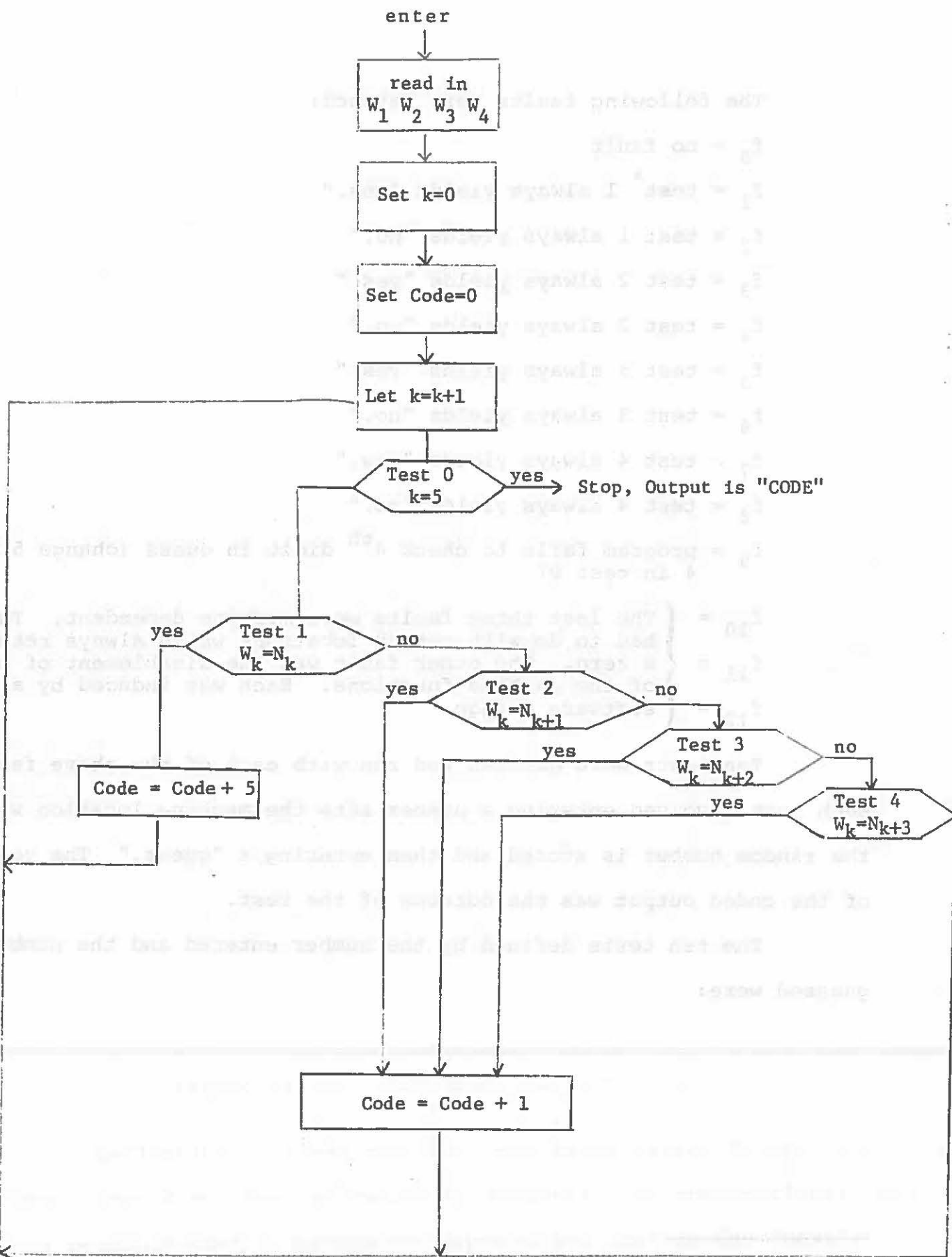


Figure III-6. Functional Flow Chart for Example 3.

The following faults were defined:

$f_0$  = no fault

$f_1$  = test\* 1 always yields "yes."

$f_2$  = test 1 always yields "no."

$f_3$  = test 2 always yields "yes."

$f_4$  = test 2 always yields "no."

$f_5$  = test 3 always yields "yes."

$f_6$  = test 3 always yields "no."

$f_7$  = test 4 always yields "yes."

$f_8$  = test 4 always yields "no."

$f_9$  = program fails to check 4<sup>th</sup> digit in guess (change 5 to 4 in test 0)

$f_{10}$  =  $f_{11}$  =  $f_{12}$  = { The last three faults were machine dependent. Two had to do with memory locations which always returned a zero. The other fault was the disablement of one of the machine functions. Each was induced by a software change.

Ten tests were defined and run with each of the above faults.

Each test involved entering a number into the machine location where the random number is stored and then entering a "guess." The value of the coded output was the outcome of the test.

The ten tests defined by the number entered and the number guessed were:

---

\*The use of the word "test" in this context refers to the tests in the flow chart, not the maintenance tests.

	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>
enter	1111	1111	1111	0123	0123
guess	1111	1234	0011	0000	0123

	T <sub>6</sub>	T <sub>7</sub>	T <sub>8</sub>	T <sub>9</sub>	T <sub>10</sub>
enter	0123	0123	0011	0011	0011
guess	3012	0011	1111	1234	0011

The experimental test results are shown in Figure III-7.

	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>
T <sub>1</sub>	20	20	4	20	20	20	20	20	20	15	5	2	20
T <sub>2</sub>	5	20	1	8	8	8	5	8	5	5	0	0	8
T <sub>3</sub>	10	20	2	12	12	12	10	12	10	5	5	10	12
T <sub>4</sub>	8	20	3	8	5	8	7	8	7	7	1	20	4
T <sub>5</sub>	20	20	0	20	20	20	20	20	20	15	5	5	8
T <sub>6</sub>	4	20	4	4	0	4	4	4	4	3	1	5	8
T <sub>7</sub>	8	20	3	8	5	8	7	8	8	11	1	10	4
T <sub>8</sub>	12	20	4	12	11	12	12	12	12	7	5	2	4
T <sub>9</sub>	1	20	1	4	2	4	1	4	1	5	0	0	4
T <sub>10</sub>	20	20	4	20	20	20	20	20	20	3	5	10	12

Figure III-7. Test Results for Example 3.

The test results given in Figure III-7 reveal that no two of the tests produce the same output for every fault. However, three of the faults ( $f_3, f_5, f_7$ ) produce the same output for every test. Thus no test plan using these ten tests will be able to distinguish among these faults.

Just to illustrate the method, an unknown one of the thirteen programs, each of which contained one of the faults  $f_0, \dots, f_{12}$ , was loaded and testing was undertaken to determine which program it was. The test tree used is shown in Figure III-8.

Since the number of possible outcomes is fairly large (12), relatively few tests are required for fault isolation. In this example only two were required to determine that fault 10 was present.

#### E. TEST OBJECTIVES

It is generally true that the testing procedure is limited by time available, computer storage, and other considerations. We will present several criteria by which test sequences can be evaluated. We assume that each test has an associated cost, perhaps the time required to implement the test.

If we can assume that we have a probability distribution over the set of possible faults so that we know the probability that each fault is present, we can select the test sequence to minimize the expected cost of testing. In this case the test sequence could be arranged to seek the most likely faults first, since testing will terminate upon the discovery of a fault. Alternatively, the objective might be to select the smallest set of tests (or minimum cost set) such that the probability of identifying any fault is at least  $\alpha$ , where  $\alpha$  is some preset parameter.

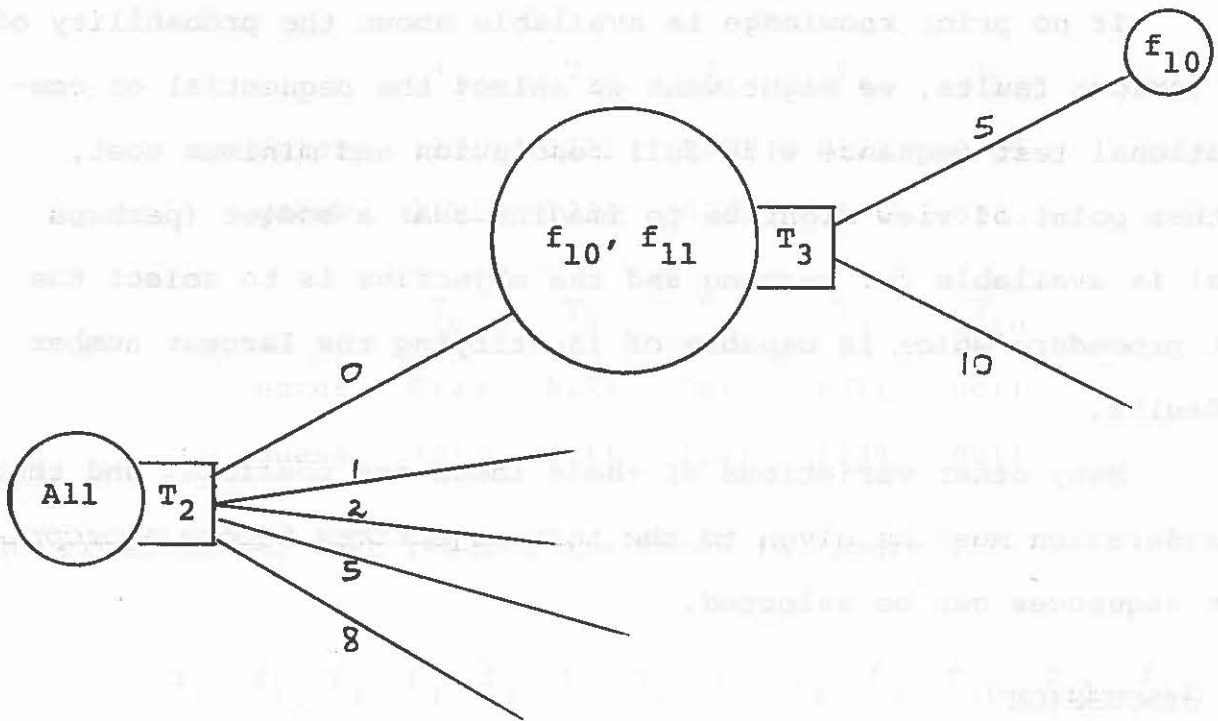


Figure III-8. Test Tree for Example 3.

If no prior knowledge is available about the probability of the various faults, we might want to select the sequential or combinational test sequence with full resolution and minimum cost. Another point of view might be to imagine that a budget (perhaps time) is available for testing and the objective is to select the test procedure which is capable of identifying the largest number of faults.

Many other variations of these ideas are possible, and thorough consideration must be given to the test objectives before appropriate test sequences can be selected.

#### F. DISCUSSION

The partitioning method just described is a general approach to systems maintenance testing. The practicality of the method depends on the extent to which the fault assumptions hold and the extent to which tests can be devised to discriminate among the faults. The issue of resetting the system to an initial state before the application of each test is also crucial. If it is impractical to reset the system after each individual test in a series of tests, then it is possible to redefine the entire series of tests as a single test. If this is done, the reset assumption is met. Of course, the new test which is in reality a series of tests, is much more comprehensive than any of the original tests alone. The data in Figure III-2 must, of course, be constructed for the new single test, not the individual tests.

The applicability of the method obviously depends on the ability to collect the data shown in Figure III-2 and this in turn

requires either a thorough analysis of the system by its designers and builders or experimentation with a fault-free system (or a model of it). The prototype system during its testing should be carefully checked out and thoroughly instrumented, and may approach a fault-free system. Practical considerations however imply that the fault-free system might never be obtained; nevertheless, the partitioning method can proceed with the understanding that the system defined by  $f_0$  is the standard whether it is fault-free or not.

The definitions of the faults to be considered must be unambiguous and, if the data in Figure III-2 is to be gathered by the experimental method in which faults are induced into an otherwise fault-free system, the faults must have some physical realization in the hardware or software. It is not sufficient to define a fault as "something is wrong with the memory unit" or "the data bus is not working properly." Fault definitions must be much more precise than this. The level of detail at which a fault must be defined probably leads to an enormous number of faults in any practical application and it is in this area that sound judgment must be exercised to prevent the approach from becoming unmanageable.

A first attempt at applying this method to a complex system might concentrate on only one class of faults, for example control faults or branching faults. If a flow chart were available, each decision point could be identified and the possible branches from that point listed. A fault would correspond to always selecting one of the branches or never selecting one of the branches. With consideration limited to this class of faults the number of faults would



remain manageable. However an item to keep in mind is that while our discussion assumes that the only faults which can occur are those pre-cataloged faults, it may be that some other fault has actually occurred. The behavior of the system is not known for such an occurrence and unless it produces some outcome other than  $O_1, \dots, O_p$ , we will erroneously identify the fault as being in our catalog.

#### IV. SOFTWARE ERROR SIMULATION

Much of the software development costs, which were mentioned in Section II, are for testing, debugging and integration; a significant part of the costs after releasing the software are for correcting errors. Thus there is current interest in the error characteristics; number, type (overflow, sequence control) and location of software errors in a program. It is generally accepted that computer programs with a complex structure, that is one with a high incidence of branch instructions and loops, are harder to debug and test and more errors persist after release than for programs with a more simple structure. An error simulation model<sup>1</sup> is presented here which investigates the relationship of program structure to error detection and test effort.

Since structure can be controlled during the design phase and measured through all phases of a computer project, the study of the relationship between structure and error characteristics is valuable to the manager of a software project. Complex program structures with poor error characteristics should be avoided. Poor error characteristics result when many errors are located in complex structures in such a way that error detection would prove difficult during testing. In cases where complex program structures may be necessary to help meet program size or speed limitations, it is useful to have an indication of the additional testing which may be caused by complex structures. It is also useful to be able to compare the error characteristics of design alternatives that have different program structures.

---

<sup>1</sup>The suggestion to use a simulation model to study software error detection was given by Dr. Samuel Litwin, a consultant to the Naval Air Development Center.

## NEED FOR RELIABLE SOFTWARE

### A. SOFTWARE COSTS

The production of software can be divided into three phases:

- \* analysis and design,
- \* writing programs and
- \* test and integration.

Data on how time, effort and money are divided among these three phases gives some indication of why software production is so costly. The fraction of time, effort and money for each phase differs from application to application; however, data from some large projects show similar experience. Estimates are given in [19] and [20] for some military command and control systems: Analysis and design is about 35 percent, writing programs 15 percent and test and integration 50 percent. For space projects the estimates are 35, 20, 45 percent. For the IBM 360 operating system the estimates are 35, 15, 50 percent. Data for business application indicates less for testing and integration and more for analysis and design than the above data. The surprising amount of time, effort and money for test and integration is often the item most underestimated in planning computer projects, as described in [12].

### B. DEFINITIONS

In the field of software engineering there is little agreement on the definition of terms, such as the definition of software reliability. In order to make the understanding of this paper easier, the following definitions will be adhered to in as much as possible.

## 1. Terms

Software reliability is the probability that a computer program will perform its intended function for a specified time interval under stated operating conditions, [21].

Reliability prediction is intended to provide an estimate of future probability of successful operation.

Testing is an effort to determine the presence of software errors, not their absence.

Software error is a mistake in program design or implementation which leads to undesirable results during program execution.

Module is a particular physical combination of program instructions that is independent of others with respect to compiling, assembling and loading and which performs a specific function.

Program is a set of modules.

Program complexity may be described by characteristics such as program size, incidence of branch instructions, incidence of loops, incidence of subroutine calls and variety of instructions.

Non-branch instructions may be either computational or input/output instructions.

Structured programming is a programming technique, [22] in which a program with one entry and one exit can be written using only the following programming progressions:

- \* Sequence
- \* IF THEN ELSE
- \* DO WHILE

Directed graph is a geometric graph, consisting of nodes and arcs, with a direction of traversal associated with each arc.

### C. CLASSIFICATION OF ERRORS

Software errors are classified as follows:

- \* Mistakes in logic at the flow chart level,
- \* Computation and assignment,
- \* Sequencing and control,
- \* Input/output,
- \* Declarations,
- \* Keypunching/clerical errors committed in writing instructions on coding sheets,
- \* New errors introduced as a result of design changes:
  - unexpected side effects caused by changes,
  - logical flaws in change to design,
  - inconsistencies between changed design and implementation,
  - inconsistencies in original and changed hardware

### D. TESTING AND ERROR DETECTION

The life cycle of a program is composed of the following phases:

- \* Design and analysis,
- \* Module development and testing,
- \* System integration testing,
- \* Functional testing,
- \* Maintenance.

The cost of error detection and repair during system integration testing is three times that of testing an individual module during module development testing, [23]. Therefore, the objective should be to reduce the number of errors detected during system integration testing and increase the number (proportion) discovered during module development testing.

In many moderate and large computer projects, a programmer writes and debugs a module and then gives it to a test group. The test group tests the module, integrates it with other modules and then continues testing. The module is tested by supplying an input to the module and then comparing the outcome to the known correct outcome. If there is a mismatch between observed and correct output, an error has been detected. When an error is detected the module is given to a programmer who locates and corrects the error and then returns the module to the test group. Notice the distinction between testing, which is supplying inputs and observing outputs, and debugging, which is the highly individualized detective work needed to locate and correct errors. In debugging, the programmer needs a detailed knowledge of the structure and operation of the module. The tester is frequently unaware of module structure and operation; he needs only to understand the function of the module.

Most computer programs have a large number of potential inputs; each may exercise a program in a different way. The sequence of instructions of the program that results from a particular input is called the "path" or "thread" associated with that input. Testing by submitting inputs to the program checks only the paths associated with those inputs. For programs with a very large number of inputs, testing can be only a relatively small sampling of all possible inputs, as described in [12].

## ERROR DETECTION MODEL

### A. NEED FOR A MODEL

Testing is a critical part of software projects because it measures and affects the final quality of the software and it consumes

a large part of project time and resources. Testing also reveals the strengths and weaknesses of the analysis, design and coding of the software and gives an estimate of the success or failure of the software after release. Thus, it is important to understand the testing process and to understand the relationships between testing and the various decision variables that may be controlled during analysis, design and coding.

A difficult facet of program testing involves the selection of inputs. The tester, who generally is not the person who wrote the code, does not know the specific path that an input will execute. Presently there is no software tool that would automatically allow the tester to force an input to follow a certain path. Some test systems allow the tester to select whichever instruction is to follow the previous one. In this way a particular path is followed, [24]. This is obviously a slow and cumbersome way to check out all, or many, of the possible paths in a program.

Obviously, inputs should be chosen so that a high percentage of the critical paths of the program will be exposed to testing. However, this objective must be weighed against the cost of machine time for debugging and the cost of programming personnel for error correction. A related matter is the determination of when to stop testing. It is usually infeasible to subject a program to all possible input combinations because of resource constraints. Various software packages are available for recording and analyzing the following types of data: count and frequency distribution of types of instructions executed; indication of code which is not executed; and indication of code which is impossible to reach, [25]. Although this type of

instrumentation is helpful for tracing program behavior, once a set of inputs is selected, it does not solve the problem of selecting the number and type of inputs in the first place, [17].

Thus, there is a need for a model to examine the relationships between the number of inputs and paths traversed, for a given program structure, and the number of remaining errors, fraction of the program exposed to testing, execution time and repair time. It is of interest to determine the number of inputs required to achieve a specified number of remaining errors for various structures, when the same number of original errors is used with each structure. In addition it is desirable to identify programming structures which have complexities that make it difficult to detect errors.

## B. BASIC MODEL DESCRIPTION

### 1. Model Characteristics

Program complexity may be described by characteristics such as program size, incidence of branch instructions, incidence of loops, incidence of subroutine calls and variety of instructions. Another view of program complexity can be obtained by considering the structure of the program to be a series of nodes, arcs and loops in the form of a directed graph as shown in Figure IV-1.

In the directed graph used in the simulation model, nodes represent connection points where parts of the program may merge and/or branch and arcs represent a sequence of nonbranching instructions such as computation and input/output. Instructions are located in arcs and errors are located in some of the instructions. An input defines a path from the start node to an exit node. Beginning at the



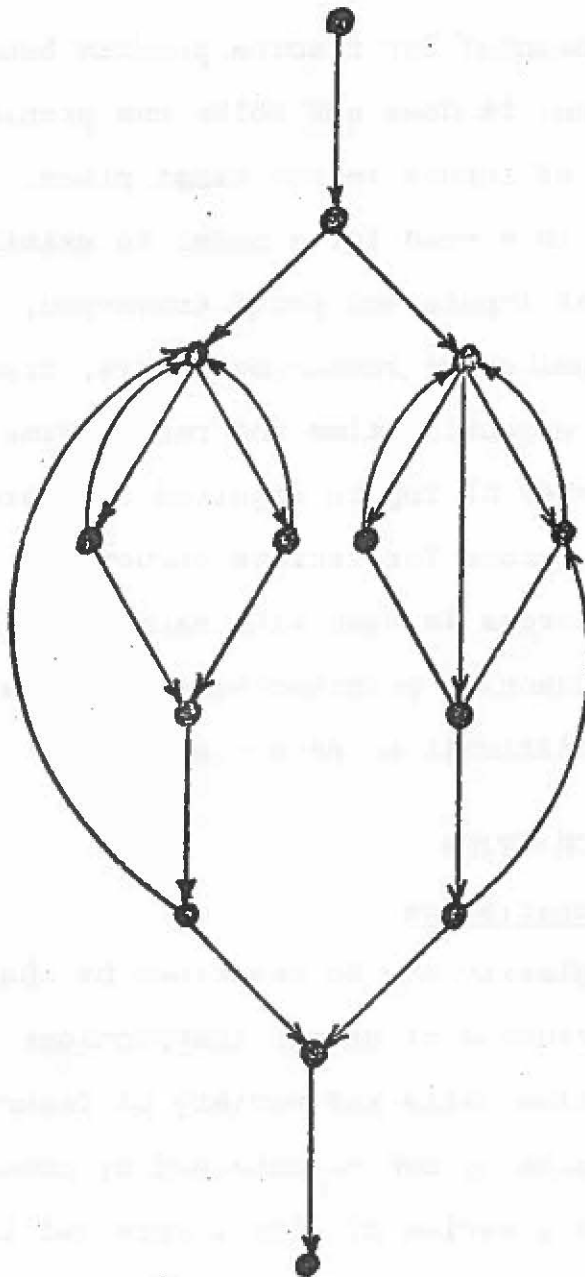


Figure IV-1. Directed Graph Representation of a Program.

start node an input causes execution of the instructions on its path, consuming test time, until an error is encountered. After the error is thus detected, it is repaired, consuming repair time. There is, however, some risk that the repair will introduce a new error in some instruction. Restarting at the initial node execution is begun again with the same input. This process is repeated until there are no errors on the path.

Some relative measures of program complexity which are applicable to a directed graph representation of program structure are:

- ratio of actual number of arcs to the maximum possible number of arcs,
- ratio of nodes to arcs,
- ratio of loop arcs to total arcs.

The size of a program is a measure of complexity in an absolute sense. In terms of a directed graph structure, size is determined by the number of nodes, which establishes the number of branch points in a program, and by the number of arcs, which establishes the degree of straightline coding between branch points. Increasing values of the above relative and absolute measures represent increasing program complexity.

## 2. Model Simulation

The error detection model was written in FORTRAN IV and has been developed and used on the Naval Postgraduate School's IBM 360/67 computer. The program has been executed 40 times in the production mode. The simulation program consists of 639 FORTRAN statements, requiring 194,000 bytes of main memory and executes in 40 to 55 seconds, depending on the type of simulation involved. The directions for use

of the error simulation program are listed in Appendix C and the FORTRAN listing for the simulation model is in Appendix D. The directed graph was input to the simulation as a node-arc incidence matrix. Lacking detailed information about the distributions of the pertinent variables in actual systems, there were no statistical dependencies among the variables established. Thus, the random variables were chosen to be independent and to possess the Markov property. This also makes the model more tractable for obtaining an analytical solution.

The number of instructions per arc is an independent exponential random variable truncated to an integer. Errors are inserted by making the number of instructions between errors an independent exponential random variable, which results in a Poisson distribution of errors per interval of instructions. Errors are inserted by scanning the arcs of the node-arc incidence matrix by columns until the count of instructions from the last error equals the random number.

An input is a sequence of random numbers that determines which arc to traverse at each branch node. For each branch node the probability of taking each arc is equal. This could be changed to test the sensitivity of error detection to different branch probabilities.

The repair times for errors are exponentially distributed. If many programmers work on error repair with each repairing only a small number of errors, the effect of experience on error repair may be small so that a constant repair rate corresponding to the exponential distribution would be appropriate. If few programmers work on repairs, experience would be a factor and an increasing repair rate distribution would be appropriate. For example, the log-normal is

sometimes used to represent the distribution of hardware repair time, [26].

The execution times of instructions are exponentially distributed. It was assumed that the execution time of an instruction does not depend on past instruction times. This assumption may not hold if the programmer tends to sequence his instructions in certain patterns.

When errors are repaired, the potential introduction of new errors is simulated. New error insertion is based on the ratio of the number of instructions changed by error repair to the total number of instructions in the arc. The arc where the new error is to be inserted is determined on an equal probability basis.

The simulation is written so that any distribution or parameter can be changed for the purpose of sensitivity analysis. The choice of distributions may have a significant effect on the simulation results for a given structure; however, since the objective is to evaluate results on a relative basis across various structures, the choice of distributions does not seem to be critical.

For each input, data are collected on the number and location of errors detected, number and location of new errors, number and location of remaining errors, number of arcs traversed, time to execute instructions and time to repair errors.

The simulation model was written so that it would be possible to generate random times for each instruction executed and for each error repaired as the simulation proceeds. However, if the instruction times and repair times are independent and identically distributed as described above, then it is possible and computationally desirable to count the number of instructions executed and the number of errors

repaired and multiply these by the average instruction executing time and the average error repair time, respectively, in order to obtain a very good estimate of each total time.

### C. MODEL ASSUMPTIONS

A basic assumption of the model is that the tester has some knowledge of the program structure, but that for a given input he does not know the specific path that it will execute. In actual software projects the test group has flow charts and program listings; however, it is infeasible to analyze this information because it may contain thousands of lines of coding. Because of the size of the program, the complicated internal logic and the large number of paths, the relationship between inputs and outcomes is rarely understood. One example is in the testing and maintenance of large operating systems. The relationship of inputs to outcomes is so poorly understood that even after an error has been detected it is often difficult to determine an input that will reproduce the error.

A further assumption of the model is that the tester gains no information as the testing proceeds that will influence his choice of subsequent inputs. In actual software projects the tester should try to make best use of any information gained during testing. Various software packages are available for recording the following types of data: count and frequency distributions of instructions executed, indication of code that is not executed and indication of code that is impossible to reach [25]. However, there are other factors that may make it difficult to effectively use the information gained during testing. For example, the test plan may be specified in advance with

no modifications allowed or inputs may be restricted to those that will be typical for the program in actual operation. For these reasons the model assumptions seem reasonable as applied to functional testing.

The probability distributions which were used are listed below.

<u>Property or Event</u>	<u>Probability Distribution</u>
Instructions per arc	Exponential
Instruction execution time	Exponential
Original error occurrence	Exponential
Time to repair an error	Exponential
Iterations per loop	Uniform
Number of instructions affected by repair	Uniform
New error occurrence	Uniform  (based on ratio of instructions changed/instruc- tions in arc)
Arc selected for new error insertion	Uniform
Arc selected at branch point for traversal	Uniform

Since little is known about the type of probability distribution which is associated with the above program properties and execution events, the selection of distributions was, of necessity, based on assumptions. However, it was felt that the assumptions were reasonable. For example, the seeding of original errors was based on

the number of instructions between errors being exponentially distributed, or equivalently, the presence of an error was independent of the presence of other errors. A second example was that instructions were placed in arcs according to an exponential distribution, or equivalently, the number of instructions between branch points was exponentially distributed. This implies that the number of instructions between two branch points was independent of the number of instructions between other branch points. Although the choice of distribution may have a significant effect on the simulation results for a given structure, the objective was to evaluate results on a relative basis across the various structures so that choice of distribution was not critical. Although it was possible to vary both the type of distribution and its parameters, the usual procedure was to keep these factors constant and vary program structure, number of inputs and input traversals.

#### D. MODEL USES

The model can be used to influence software design decisions by making it possible to compare the error detection characteristics of alternative program structures. This is valuable, since error detection characteristics are good indicators of the time and resources consumed by testing. The design flow charts and estimates of branch probabilities and number of instructions can be used to specify programs in the form of a directed graph. The program is then seeded with errors and subjected to random inputs.

The model can also be used to identify the measure or measures of complexity that best predict the ability to detect errors. To do

this it is necessary to gather data from the model on the error detection characteristics of a variety of different structures and then do a statistical analysis. This would make it possible to measure the complexity of different programs and then compare the estimates of error detection characteristics. Although some data has been generated, further work is necessary to identify good measures of complexity.

There are other situations where it is useful to be able to compare structures. A frequent problem is to evaluate the cost of adding some additional feature to the program. The results of the model can be used to compare error detection characteristics of the original and modified structure. The problem of how to allocate test effort among structures of different size and complexity can also be addressed.

## ANALYSIS OF SIMULATION RESULTS

### A. THE EFFECT OF INCREASING THE NUMBER OF INPUTS

#### 1. Model Testing

One would expect that initially there are many errors detected in a program with each input and then the number of errors detected decreases as additional test inputs are used, because much of the program is exposed to testing initially. This is illustrated in Figure IV-2. The percent residual errors decreased stepwise as the number of inputs increased. In the testing of actual software, after finding many errors, there may be long periods of time with no error detection followed by a new group of detected errors.



30 nodes, 50 arcs, 6 loops  
18 original errors, 11 added errors

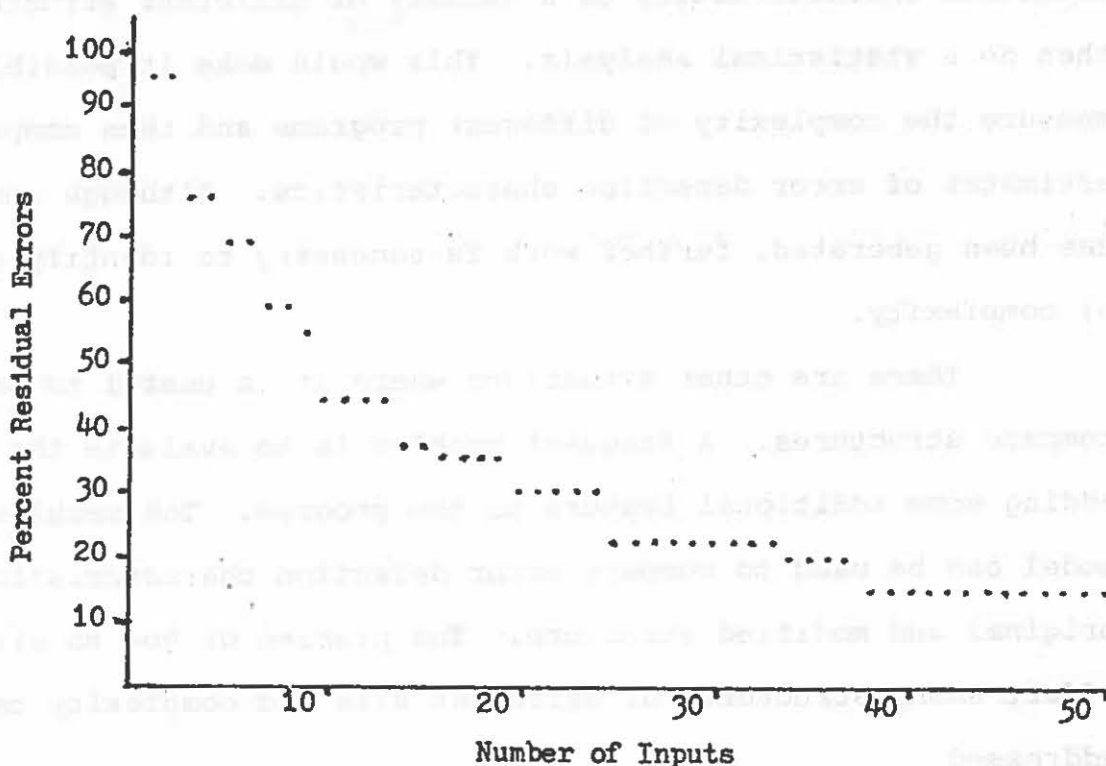


Figure IV-2. The Effect of Increasing Inputs on Residual Errors.

Recall that each input in the model detected all the errors in its path, from the input node to one of the output (terminal) nodes. In order to explain the stepwise action in Figure IV-2 it must be realized that although the paths through the program were, in general, different from previous paths, portions of these paths may have involved only arcs that have been previously traversed. The model had well defined steps where no new arcs were tested for a number of unique input paths, as shown in Figure IV-3. Thus it can be seen how a new group of errors was detected when the model tested previously untested parts of the program.

However, just because an arc has been previously tested does not imply that it was error free. As each new detected error was

30 nodes, 45 arcs, 6 loops  
 28 original errors, 17 added errors

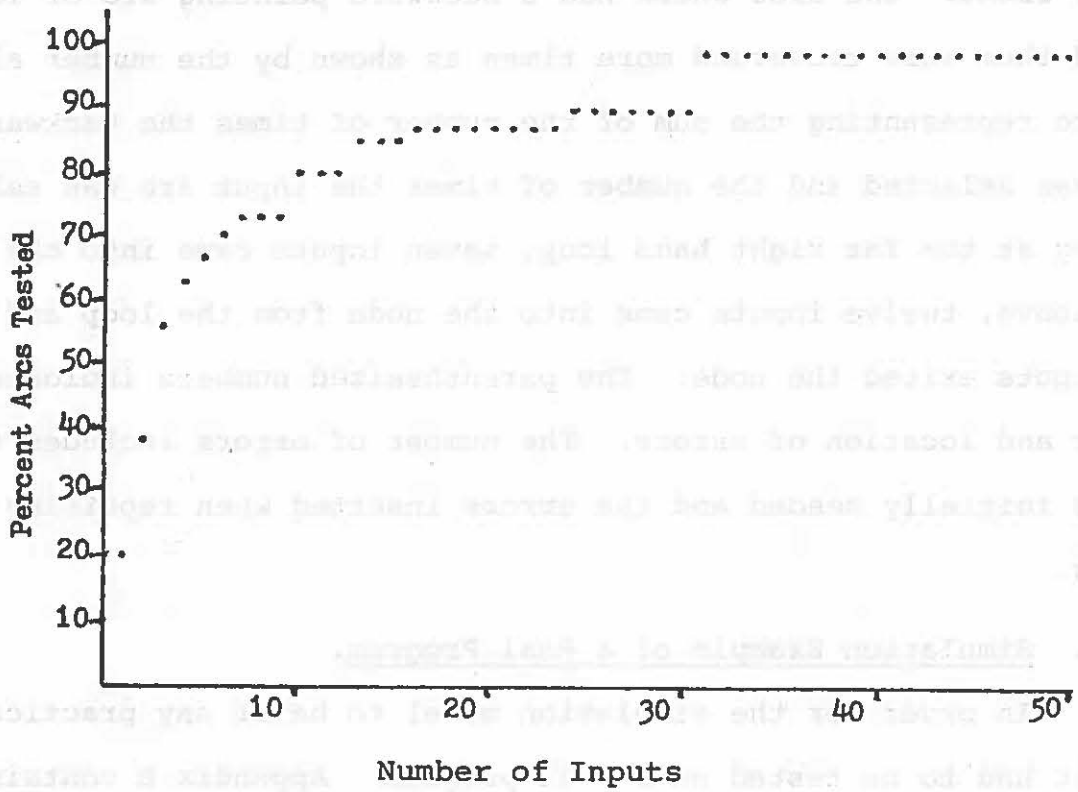


Figure IV-3. The Effect of Increasing Inputs on Arcs Tested.

repaired, there was some small probability that a new error was introduced in some other portion of the program. This newly inserted error may have been inserted in a previously tested arc. A check was made on the coverage of the arcs by the simulation model. The structure checked had 30 nodes, 40 arcs and 6 loops as shown in Figure IV-4. The numbers along the arcs indicate the number of times the arc was traversed. For example, the source arc at the top was traversed 50 times, or there were 50 different inputs. Every time an input reached a node it had an equally likely opportunity to select any one of the arcs emanating from the node. The simulation results bear

this out as Figure IV-4 illustrates, where the 50 inputs traversed the top arc and split below with 25 going to the left and 25 going to the right. The arcs which had a backward pointing arc or loop around them were traversed more times as shown by the number alongside the arc representing the sum of the number of times the backward loop was selected and the number of times the input arc was selected. Looking at the far right hand loop, seven inputs came into the node from above, twelve inputs came into the node from the loop and 19 of the inputs exited the node. The parenthesized numbers indicate the number and location of errors. The number of errors includes the errors initially seeded and the errors inserted when repairing detected errors.

## 2. Simulation Example of a Real Program.

In order for the simulation model to be of any practical use, it had to be tested on a real program. Appendix B contains the code and structure of a textbook FORTRAN program for computing Bessel Functions. The column labeled "node" corresponds to the nodes in the directed graph representation of the program in Figure IV-5. This particular program was selected as an example of a good computational program, since it was presented in a numerical analysis text, [26] as an example of a poorly coded program, since a casual reading of the code showed a lack of use of structured programming techniques. Another reason for the selection was that the program could be broken down into 30 nodes, which was the same as the test structures. It also fit within the range of structures tested having 43 arcs and 9 loops. The first number inside of the parenthesis represents the number of instructions in the arc and the second number represents

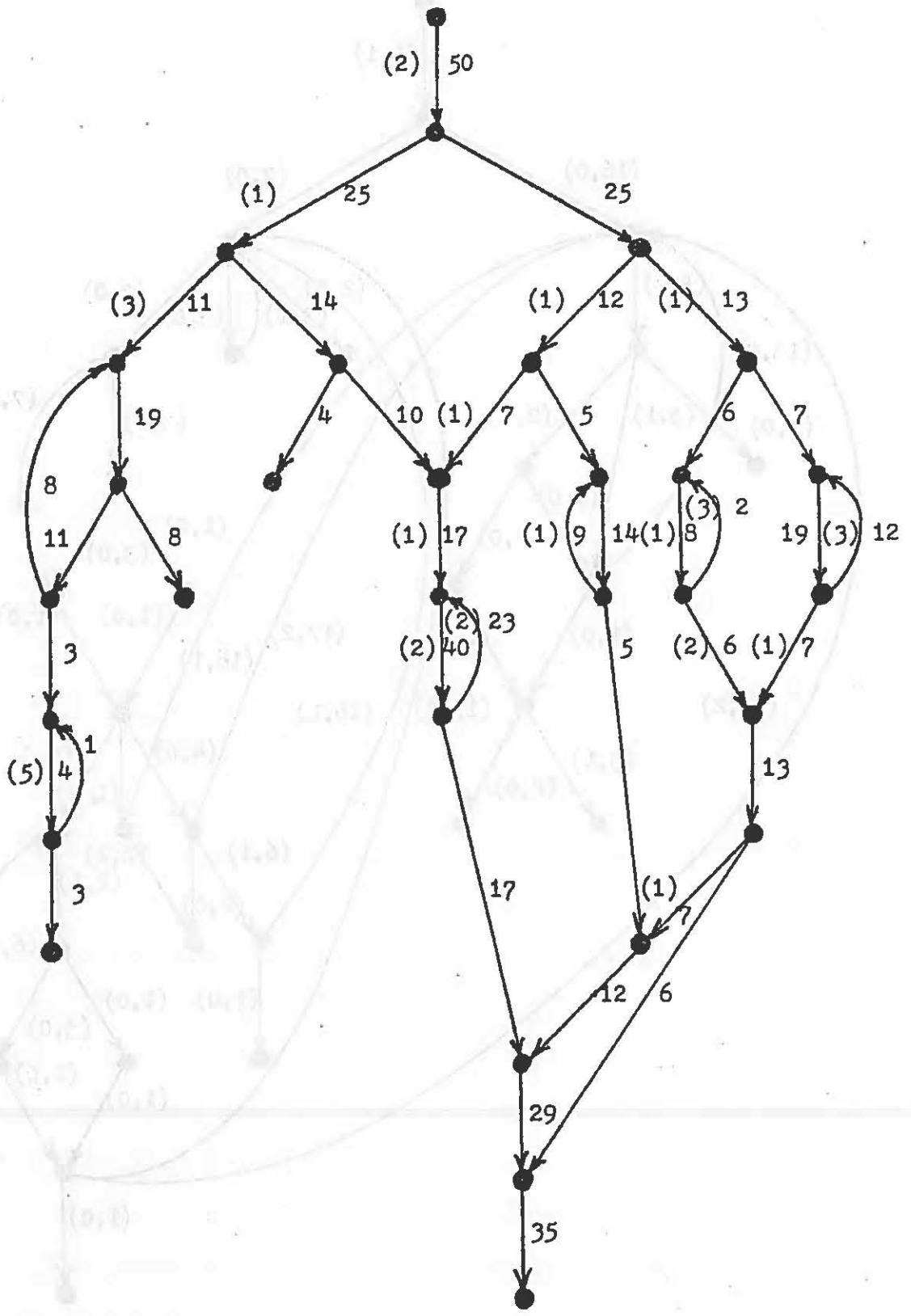


Figure IV-4. Arc Traversal and Error Patterns.



the number of errors, original errors plus added errors, in the arc.

Fifty randomly selected inputs were run through the structure. With 16 errors initially seeded, five errors, or 16.7 percent of the total errors seeded, still remained after fifty inputs. Comparing this result with a test structure with 45 arcs and 6 loops and another test structure with 44 arcs and 10 loops, the percent residual errors in the FORTRAN program was high, illustrated by Figure IV-6. By analyzing the paths each input traversed it was noted that six of the nine loops in the FORTRAN program, all emanating from the bottom of the graph, going to the top of the graph, were very seldom used, thus not giving each individual input an opportunity to loop back up to the top of the graph, and thus test more branches for a given input. This was borne out by the results in Figure IV-7, which showed that the percentage of arcs tested was lower for the FORTRAN program.

#### B. THE EFFECT OF INCREASING THE NUMBER OF ARCS

Intuitively, given two programs with the same number of nodes, and a different number of arcs emanating from the nodes, one would expect that the program with the greater number of arcs, or the more complex program, would have the higher percentage of residual errors. By the same reasoning one would expect the more complex program to have fewer arcs tested with a given number of inputs.

Fifty random inputs were used on each of the following program structures. Each structure contained thirty nodes and six loops. Retaining the concept that each node represents a branch or decision point in the program, the most simple structure that can be defined,

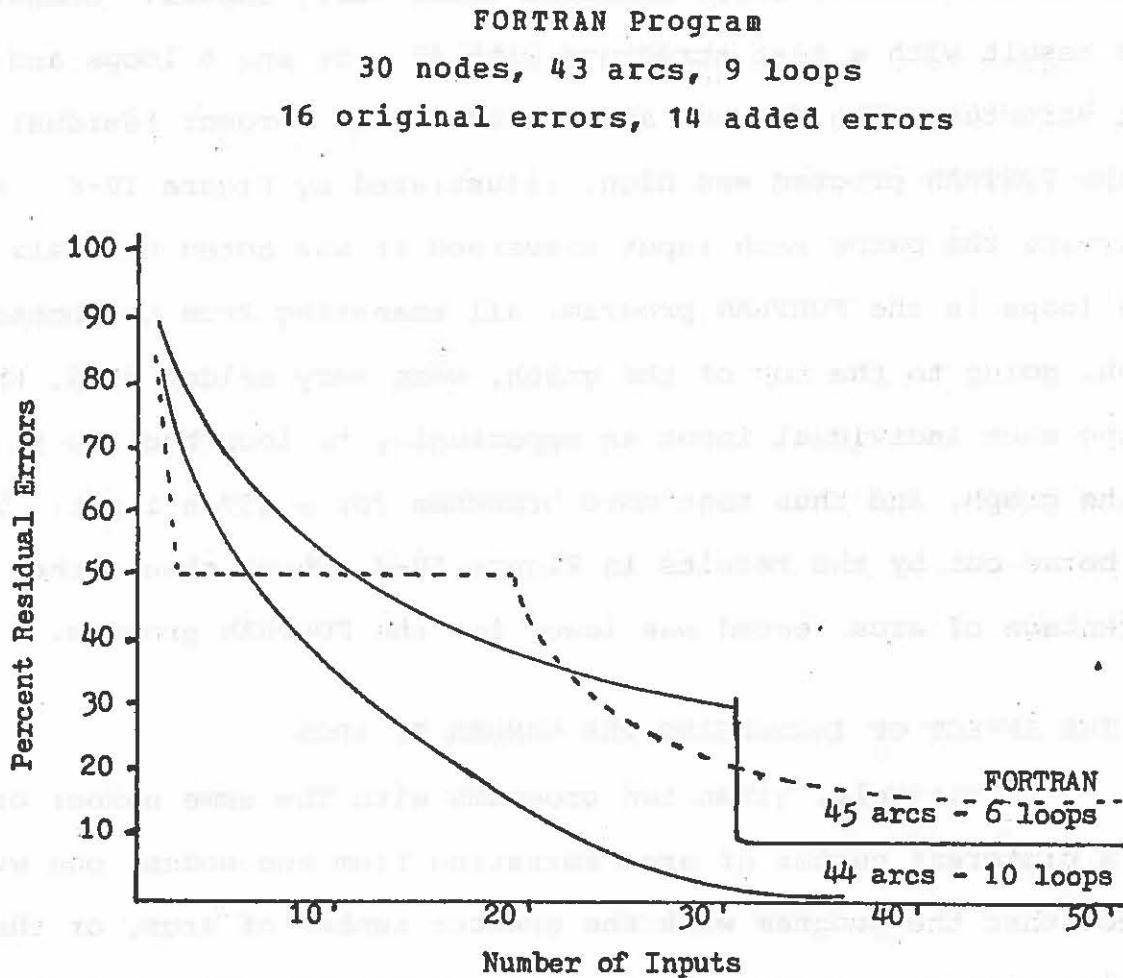


Figure IV-6. Residual Error Pattern

FORTRAN Program  
 30 nodes, 43 arcs, 9 loops  
 16 original errors, 14 added errors

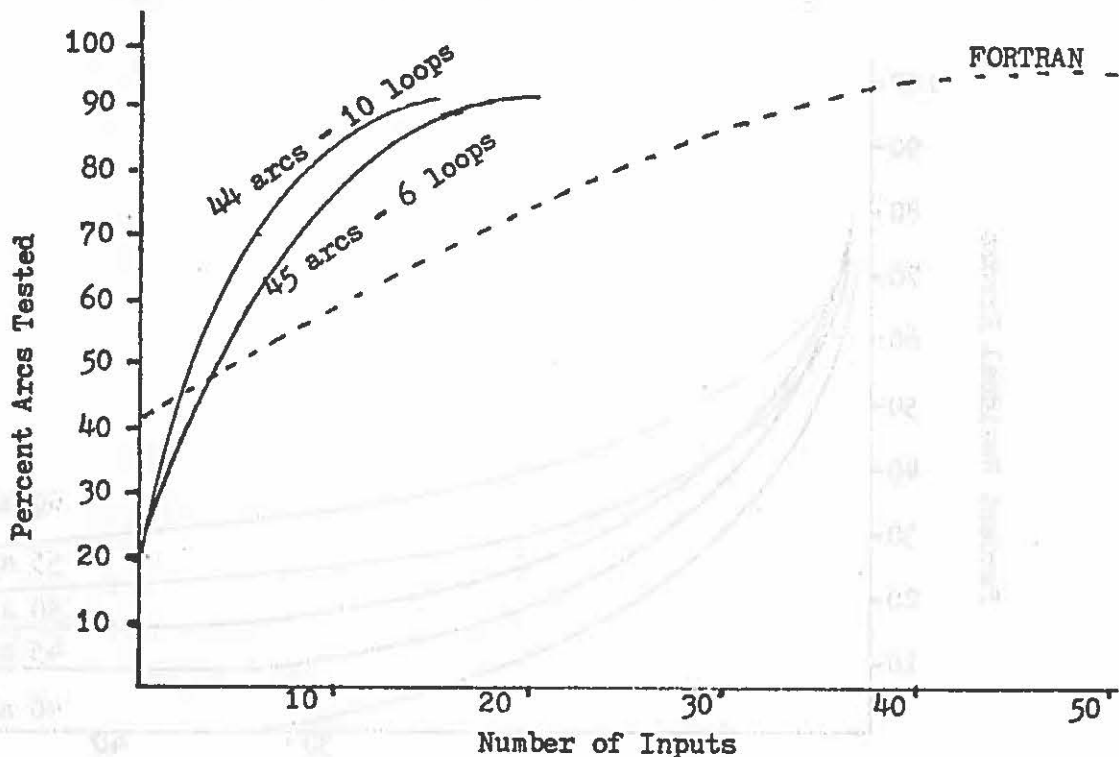


Figure IV-7. Arcs Tested Pattern

using thirty nodes, must have a minimum of forty arcs. By definition, to establish a node there must be at least three arcs, in any combination, either terminating or emanating from the node; thus, the minimum number of arcs in a structure is:  $(3/2)$  (the total number of nodes) minus the number of entry and terminal nodes. Recall that an arc was defined as either a forward or backward pointing arc, called a loop. Starting with forty arcs and adding five more to each structure, five structures were simulated with 40, 45, 50, 55 and 60 arcs. After fifty inputs the percent residual errors increased as the number of arcs increased, as Figure IV-8 illustrates. The percent residual



30 nodes, 40 - 60 arcs, 6 loops

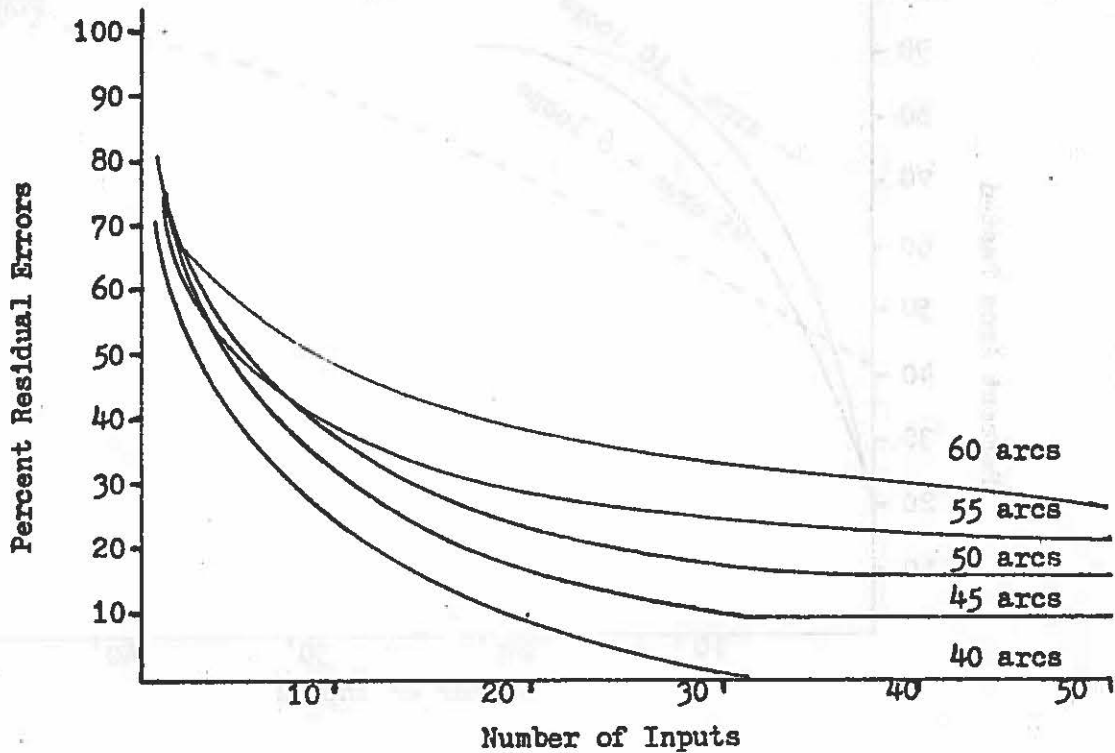


Figure IV-8. Relation Between Complexity and Residual Errors.

errors was chosen as the vertical axis in Figure IV-8 rather than residual errors since the number of errors, original errors plus added errors, varied in each of the five structures. The reason for the variable number of errors was that each time a new structure was defined, the error simulation program would randomly seed all the original errors again, thus errors could have been inserted into the added arcs.

Similarly, Figure IV-9 illustrates the effect of increased complexity on the percentage of the arcs tested. As the number of arcs increased, the percent of the arcs tested decreased.

30 nodes, 40 - 60 arcs, 6 loops

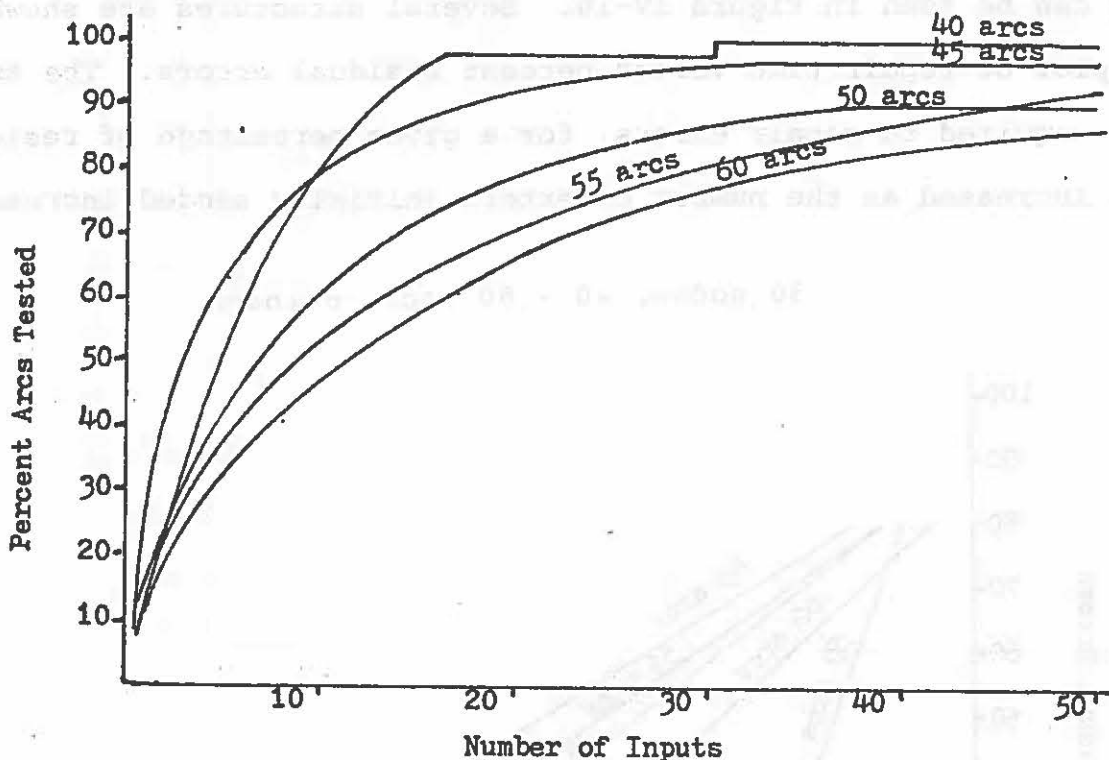


Figure IV-9. Relationship Between Complexity and Percent Arcs Tested.

Examining the paths traversed by each input gave some insight as to why an increased number of arcs caused higher residual error and lower arcs tested percentages. When an arc was added, the number of arcs emanating from a node increased. There was a probability that the added arc could contain an error as the entire structure was seeded with errors anew. As the number of arcs increased, there were also more arcs which provided shorter paths to an exit node by connecting a node closer to the input with a node closer to one of the outputs, thus leaving some intermediate arcs untested.

Repair time turned out to be unrelated to complexity. The number of errors initially seeded controlled the repair time. These results can be seen in Figure IV-10. Several structures are shown in the plot of repair time versus percent residual errors. The amount of time required to repair errors, for a given percentage of residual errors, increased as the number of errors initially seeded increased.

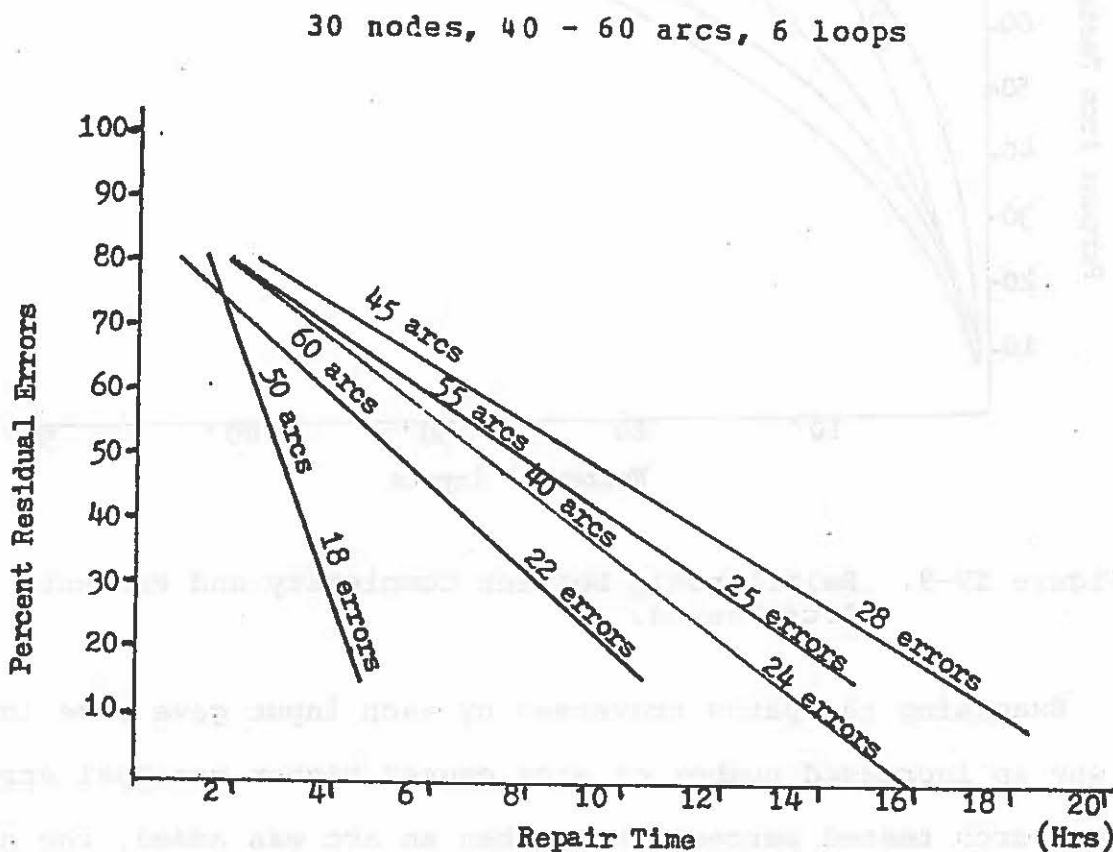


Figure IV-10. The Effect of Arcs on Repair Time

Generally the relationship between the percent arcs tested and the percent residual errors can be described as approximately linear. As the percentage of the arcs tested increased, the percentage of the residual errors remaining decreased. In Figure IV-11 the

30 nodes, 40 - 60 arcs, 6 loops

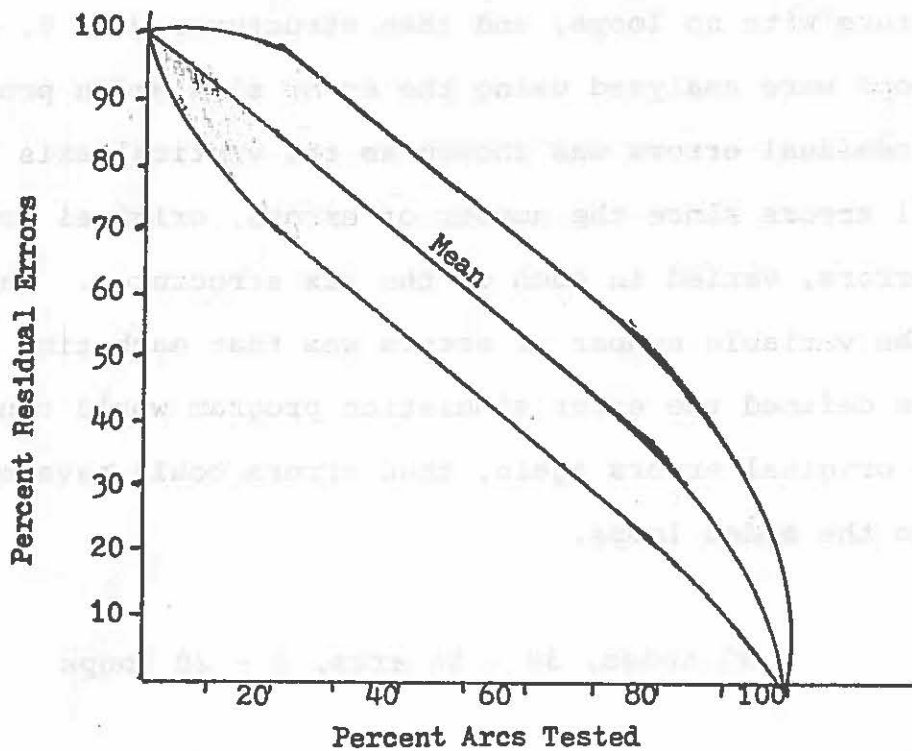


Figure IV-11. The Effect of Complexity on the Relationship Between the Residual Errors and Arcs Tested.

shaded area represents the band of values, corresponding to various structures, for a given percentage of arcs tested, with the mean shown as the middle curve.

#### C. THE EFFECT OF INCREASING THE NUMBER OF LOOPS

Improper loop indexing is usually near the top of a list of most frequently occurring errors, [19]. Many people think that loops should be eliminated as a program structure. Note that the only influence of loops in this model is with respect to coverage. The model does not account for errors in the loop counter or failure to get out of a loop. One of the results of the analysis was that an increase in the number of loops had no significant effect on the

percentage of residual errors, as shown in Figure IV-12. Starting with a structure with no loops, and then structures with 5, 6, 10, 14 and 20 loops were analyzed using the error simulation program. The percent residual errors was chosen as the vertical axis rather than residual errors since the number of errors, original errors plus added errors, varied in each of the six structures. The reason for the variable number of errors was that each time a new structure was defined the error simulation program would randomly seed all the original errors again, thus errors could have been inserted into the added loops.

30 nodes, 34 - 54 arcs, 0 - 20 loops

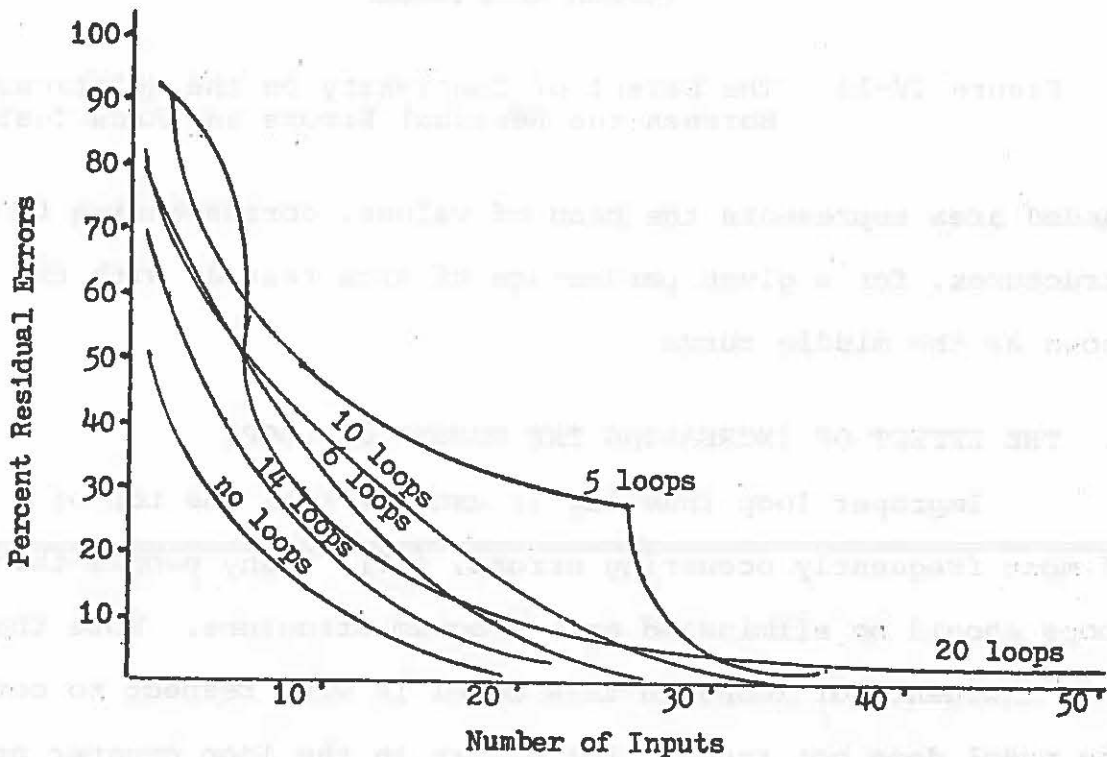


Figure IV-12. The Effect of Loops on Residual Errors.

The reason for the independence of the percent residual errors from loops can be explained by examining Figure IV-13. There was no distinguishable difference between the percent arcs tested in the six cases with 0, 5, 6, 10, 14 and 20 loops.

30 nodes, 34 - 54 arcs, 0 - 20 loops

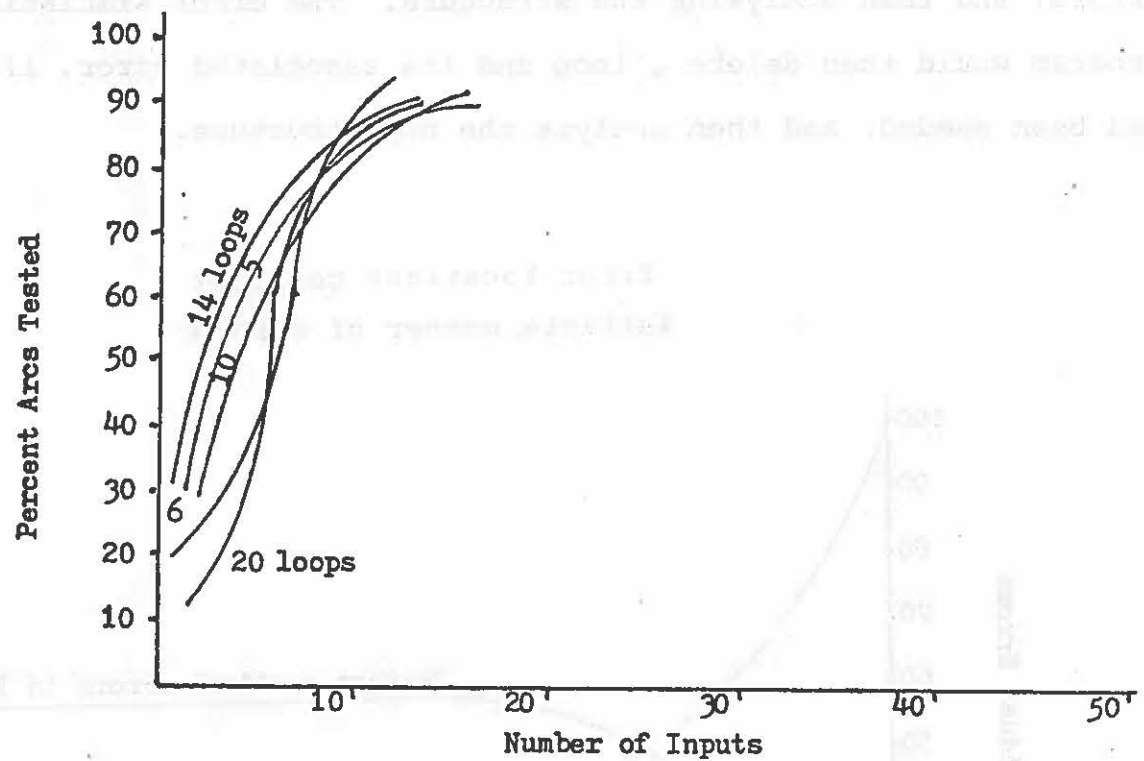


Figure IV-13. The Effect of Loops on Percent of Arcs Tested.

By examining the paths the inputs trace, the explanation of the above becomes obvious. After an input completes a loop, it once again has an opportunity of branching out of the loop, thus testing more arcs than a structure with no loops. Each time another loop was added, the probability of branching out of all the loops increased at approximately the same rate as the increased number of loops. This concept was reinforced by the data shown in Figure IV-14.

After the structure was expanded to nine loops, any additional loops had no effect on error detection. The percent of the total residual errors in the structure that resided in the loops was a constant 59 percent and the percent in the arcs was a constant 41 percent for structures with nine to twenty loops. This data was derived by starting with a structure containing 20 loops, seeding errors, and then analyzing the structure. The error simulation program would then delete a loop and its associated error, if one had been seeded, and then analyze the new structure.

Error locations constant  
Variable number of errors

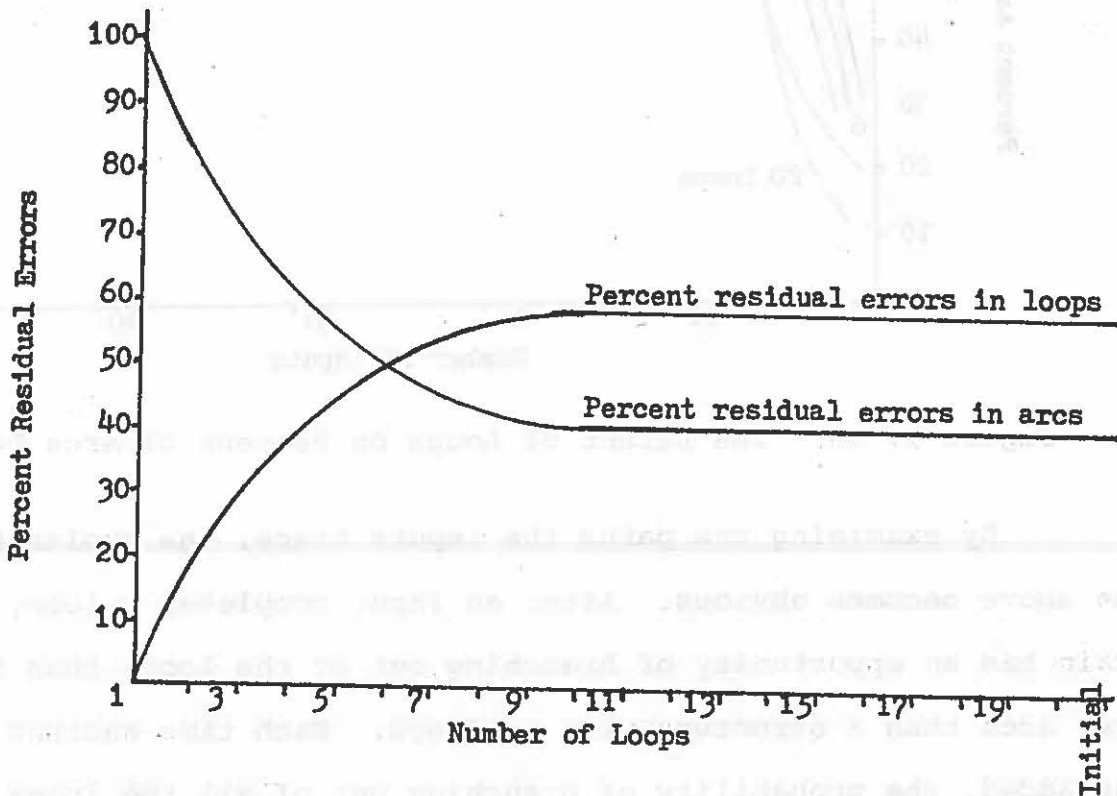


Figure IV-14. Residual Errors in Loops and in Arcs.

It required essentially the same amount of repair time to decrease the percent residual errors to a certain level for all the structures containing loops, as shown in Figure IV-15.

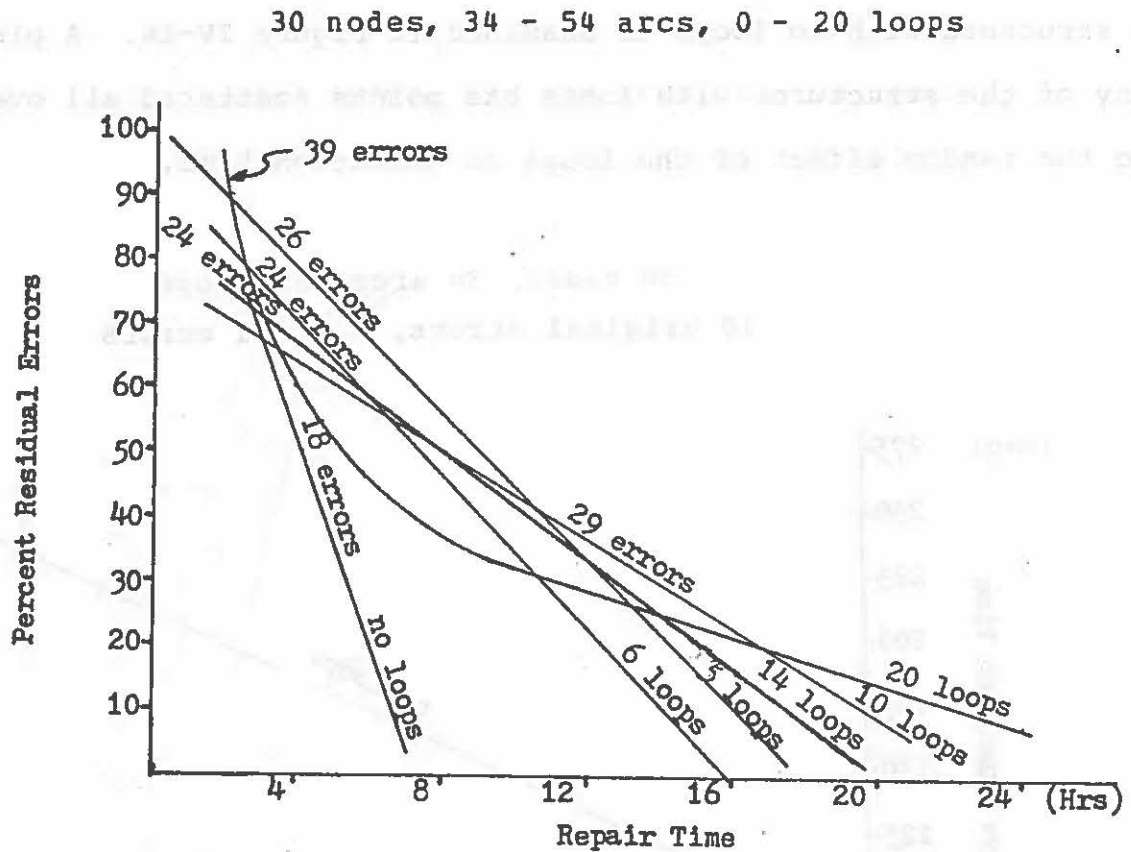


Figure IV-15. The Effect of Loops on Repair Time.

However, the number of errors initially seeded had no distinguishable effect on the repair time of the structures with loops.

It was not possible to make any judgements concerning the determinants of execution time. This was due to the fact that all but one structure tested had loops in it. Loops were executed a variable number of times as determined by a uniform distribution which established the number of iterations. The effect of a doubly



nested D0 loop was captured by allowing an input to have an equiprobable chance of branching back up to the start of the loop or of branching farther down the structure. The relationship between the number of inputs and the cumulative execution time for a structure with no loops is examined in Figure IV-16. A plot for any of the structures with loops has points scattered all over due to the random effect of the loops on execution time.

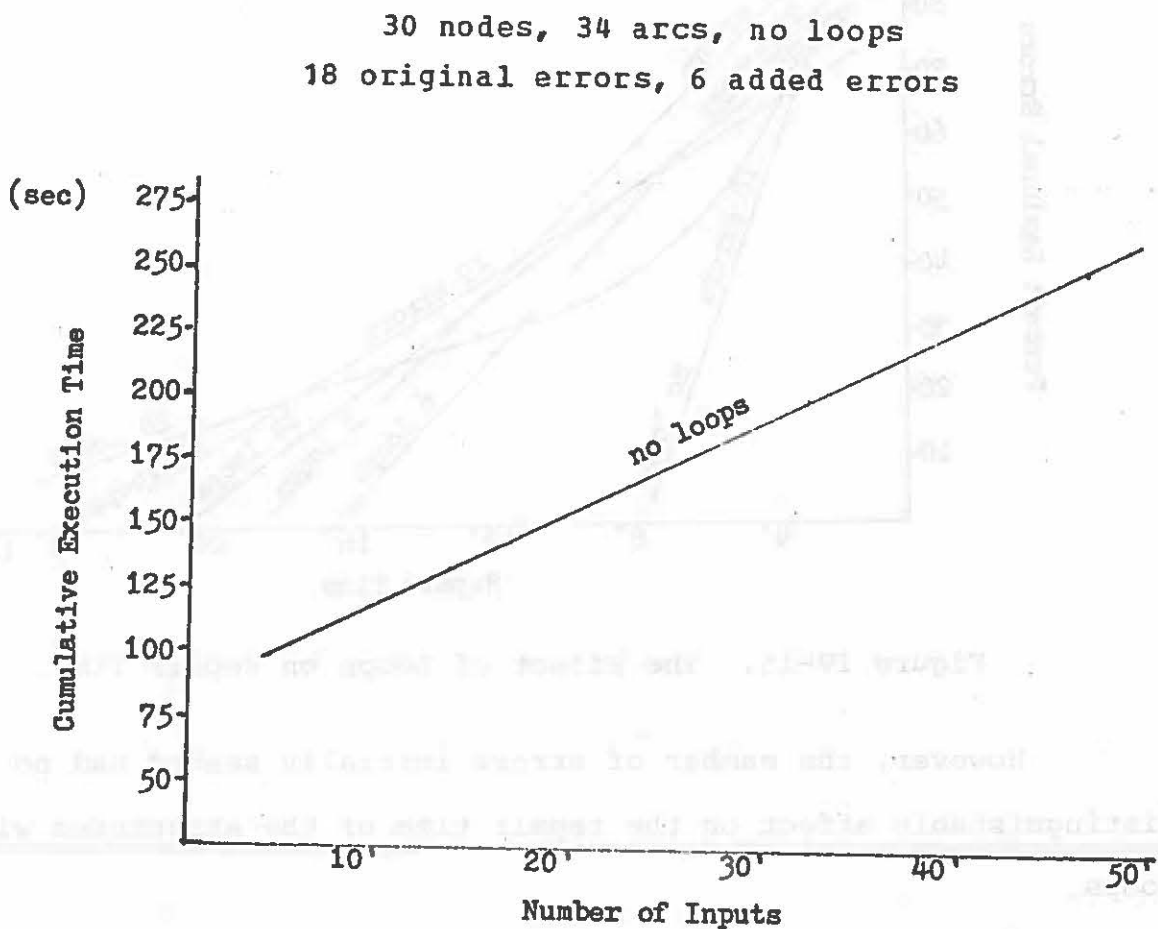


Figure IV-16. Execution Time for a Structure with No Loops.

The effect of the percentage arcs tested, with loops present, on the percent residual errors can loosely be described as a linear relationship. As long as the structures all had loops, the curves of the percentage of arcs tested versus the percent residual errors all fall within a narrow band of values as shown by the shaded area of Figure IV-17. The curve for a structure with no loops is also plotted.

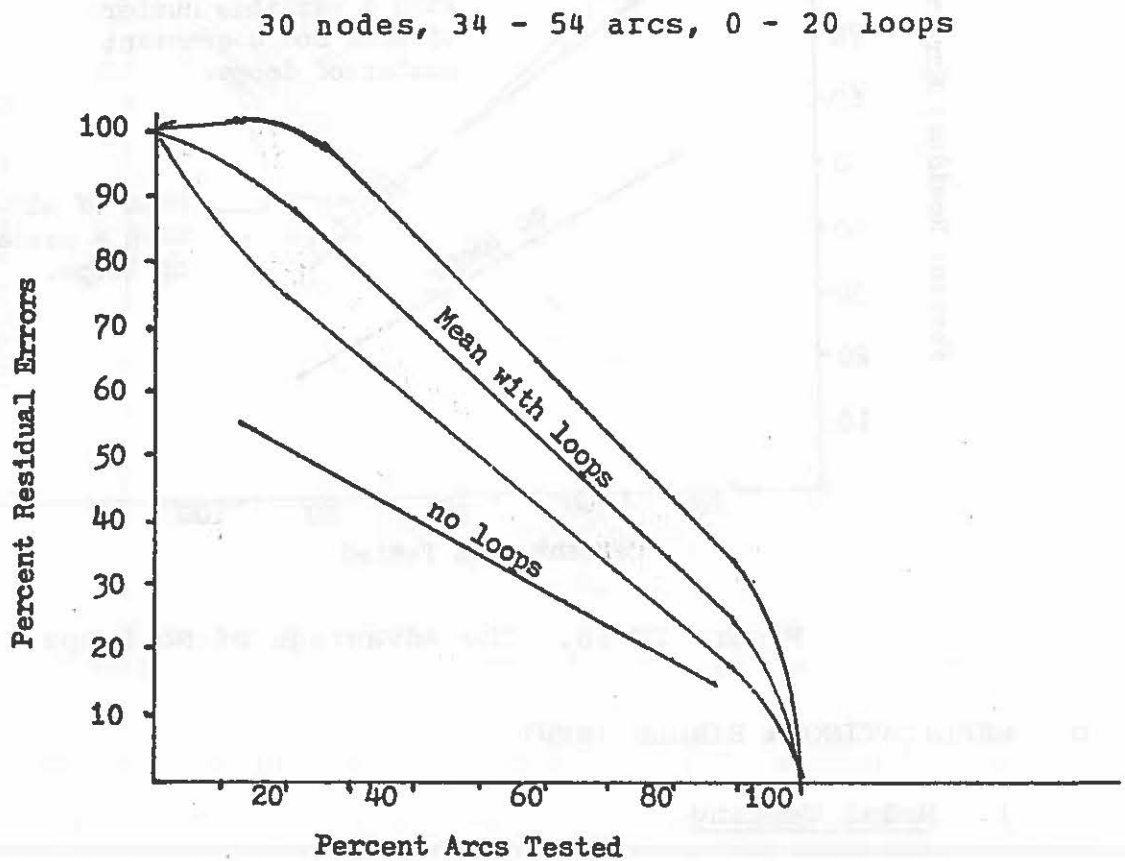


Figure IV-17. The Effect of Loops on the Relationship Between Residual Errors and Arcs Tested.

The mean of structures with a variable number of arcs and a constant number of loops and the mean of structures with a variable

number of loops are plotted in Figure IV-18 illustrates, the structure with no loops required significantly fewer arcs to be tested to achieve the same level of residual error percentage as compared to the structures with loops.

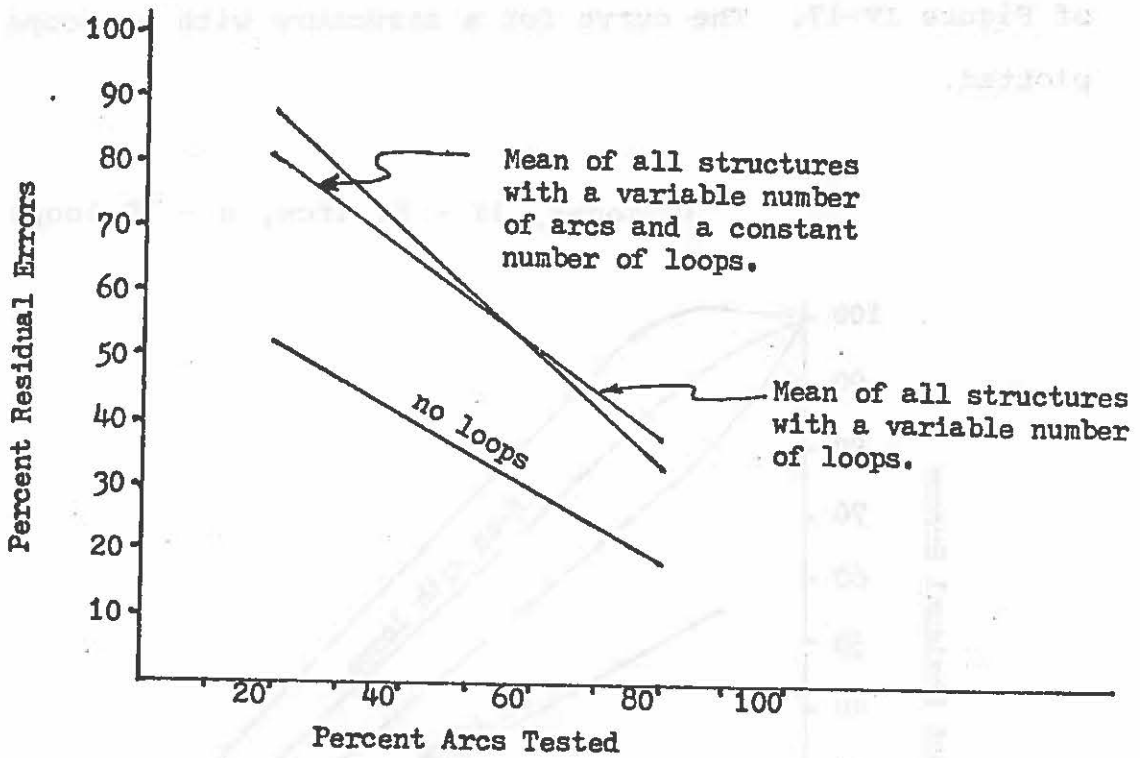


Figure IV-18. The Advantage of No Loops

D. REPLICATING A SINGLE INPUT

1. Model Testing

The usefulness of the model is now examined for predicting the ability of detecting errors in an actual program. Four pieces of information are of importance for a manager conducting module development testing of computer programs. These are: the percent of number of residual errors, the percent or number of arcs tested, the amount of repair time, and the amount of execution time.

The simulation model was used on ten different test structures to see if this information could be predicted. For each structure a single randomly selected input was run and the above data were collected. This process was replicated 100 times, or in other words, 100 randomly selected inputs were used with each input using the same structure and the same number of errors seeded in the same places. Statistics such as mean, median, variance, standard deviation, etc., were calculated.

As an example, the basic 30 node, 40 arc, 6 loop structure had 24 errors initially seeded. The simulation model produced a mean of 78.79 percent residual errors with a standard deviation of 9.10 for one input. Thus, one could estimate that based on 24 original errors, after one input, 78.79 percent of the errors will remain. Similar statistics were determined for percentage of arcs tested and repair time. Execution time was found to have a high variance. For instance, the mean execution time for one input for the above structure was 32.50 seconds with a standard deviation of 50.15. Thus, estimates of execution time based on the mean would be subject to high error.

## 2. Simulation Example on a Real Program.

The simulation model was used on the FORTRAN Bessel Function program described earlier. It was found that, based on 16 original errors, the expected percent residual errors was 84.26 percent with a standard deviation of 9.09, or 15.74 percent of the original errors could be expected to be found and corrected with one input. Similarly, 17.70 percent, with a standard deviation of 8.65, of the arcs could be expected to be tested by one input. Of prime importance to the project manager, 1.41 hours of repair time, with a standard deviation

of 1.18, a relative measure for the manager to use when comparing alternative structures, could be expected to be devoted to detecting and repairing 15.74 percent of the errors.

#### E. THE EFFECT OF COMPLEXITY

The following complexity measures will be used:

\*AMA is the ratio of the number of arcs in the structure to the maximum number of arcs possible for the given number of nodes,

\*NA is the ratio of the number of nodes to the number of arcs in the structure,

\*LA is the ratio of the number of loops to the number of arcs in the structure.

Using these relative complexity measures, it was of interest to see how each of the measures affected the percent residual errors and the percentage of arcs tested. Five different structures with a constant number of loops and a varying number of arcs and six different structures with a varying number of loops were examined. For each structure, 100 replications of a single input were simulated using the error simulation program, and statistics were gathered about the percent residual errors and the percentage of arcs tested.

In Figure IV-19, the percent residual errors after one input increased as the complexity increased. In this case the complexity measure was the ratio of the actual number of arcs to the maximum number of arcs possible with a given number of nodes. Similarly, using the same complexity measure the percent arcs tested after one input decreased as the complexity increased, as shown in Figure IV-20. In both Figures IV-19 and IV-20, the standard deviation from the mean, represented by the dashed lines, decreased as complexity increased.

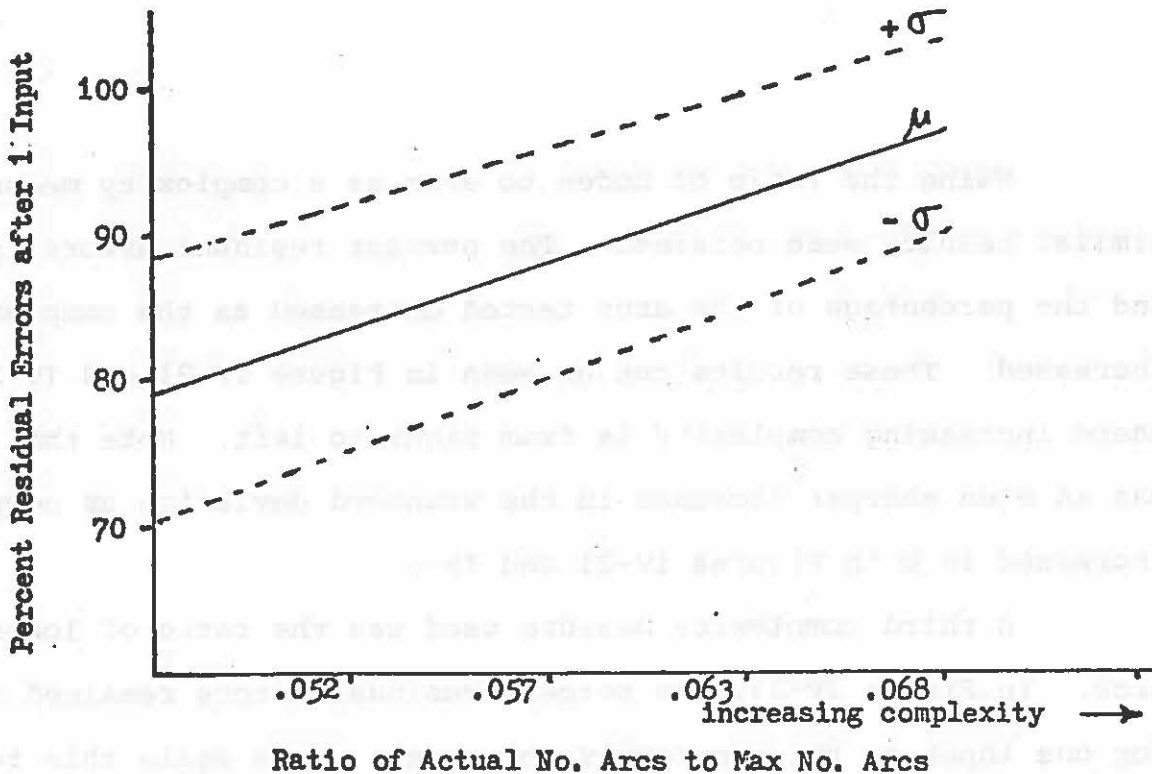


Figure IV-19. The Effect of AMA on Residual Errors.

Constant number of nodes  
Variable number of arcs

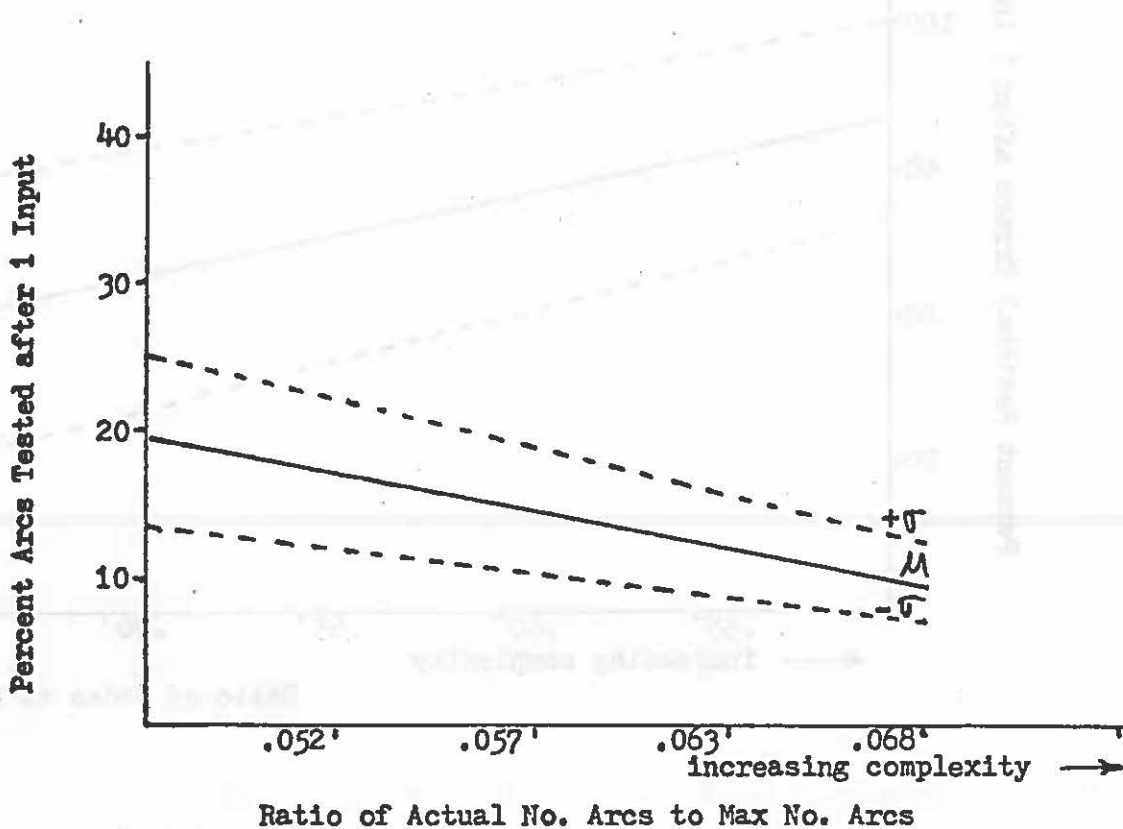


Figure IV-20. The Effect of AMA on Arcs Tested.

Using the ratio of nodes to arcs as a complexity measure similar results were obtained. The percent residual errors increased and the percentage of the arcs tested decreased as the complexity increased. These results can be seen in Figure IV-21 and IV-22 where increasing complexity is from right to left. Note that there was an even sharper decrease in the standard deviation as complexity increased in both Figures IV-21 and IV-22.

A third complexity measure used was the ratio of loops to arcs. In Figure IV-23, the percent residual errors remained constant for one input as the complexity increased. Once again this reinforced

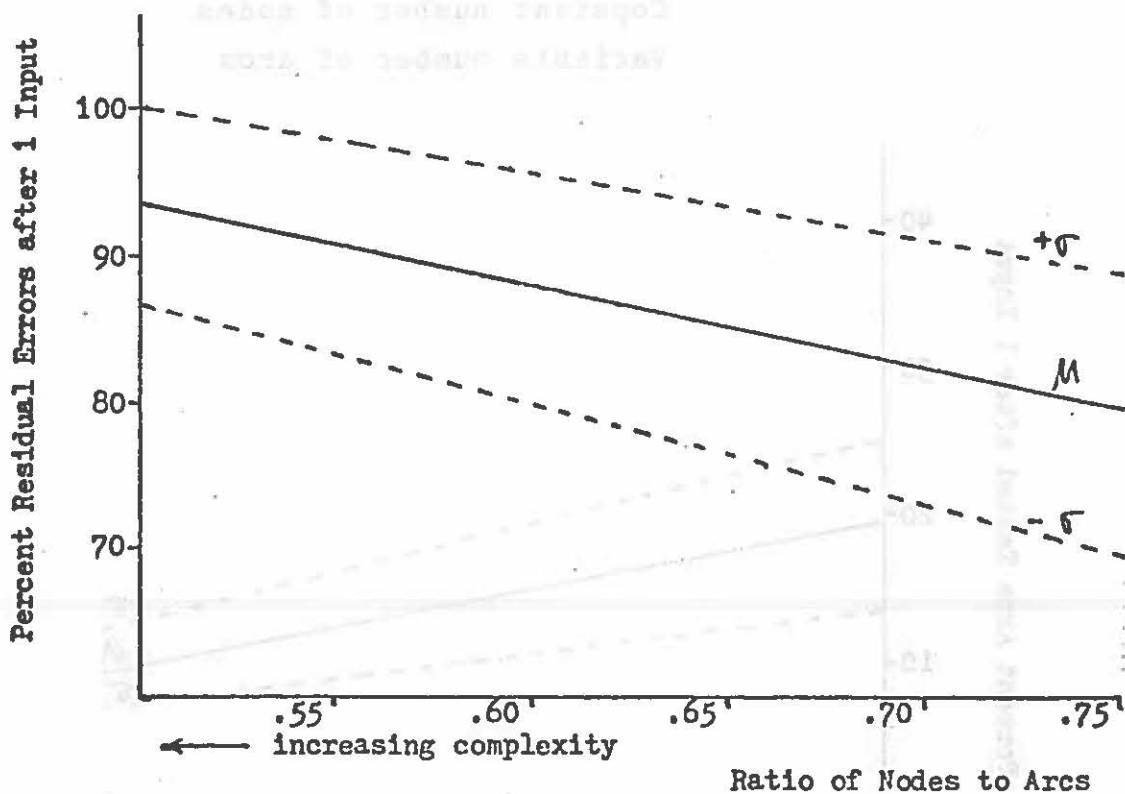


Figure IV-21. The Effect of NA on Residual Errors

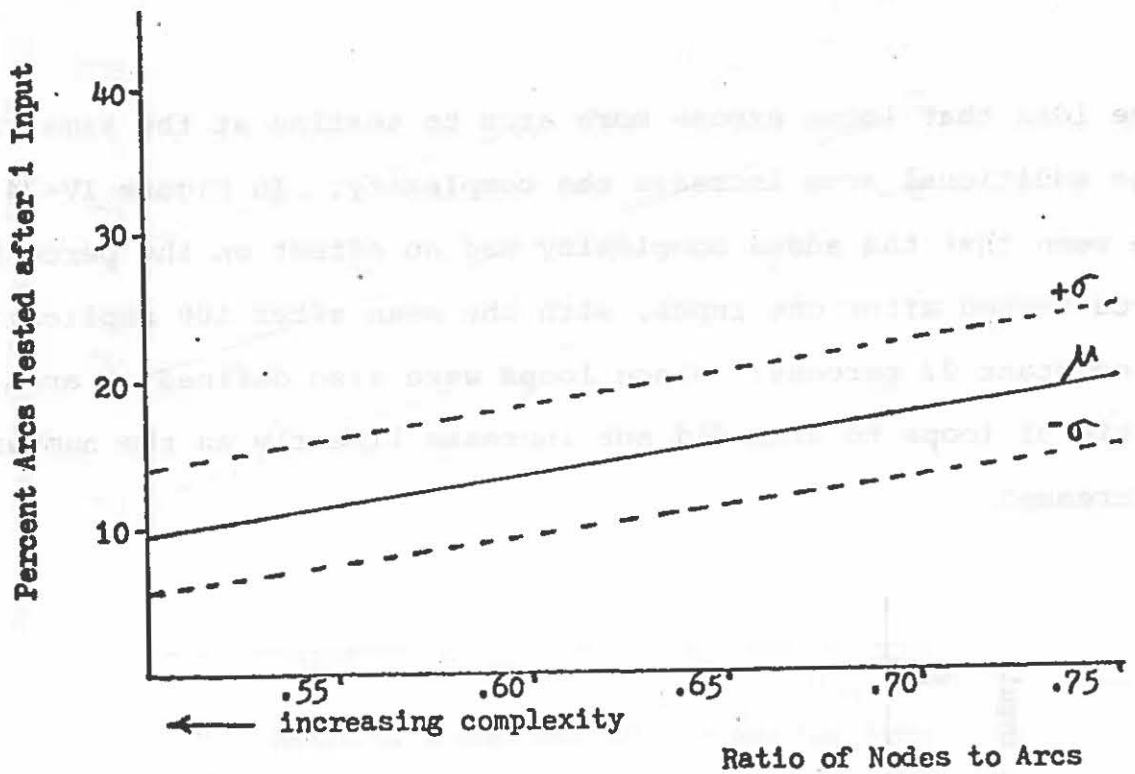


Figure IV-22. The Effect on NA on Arcs Tested.

Constant number of nodes  
Variable number of arcs

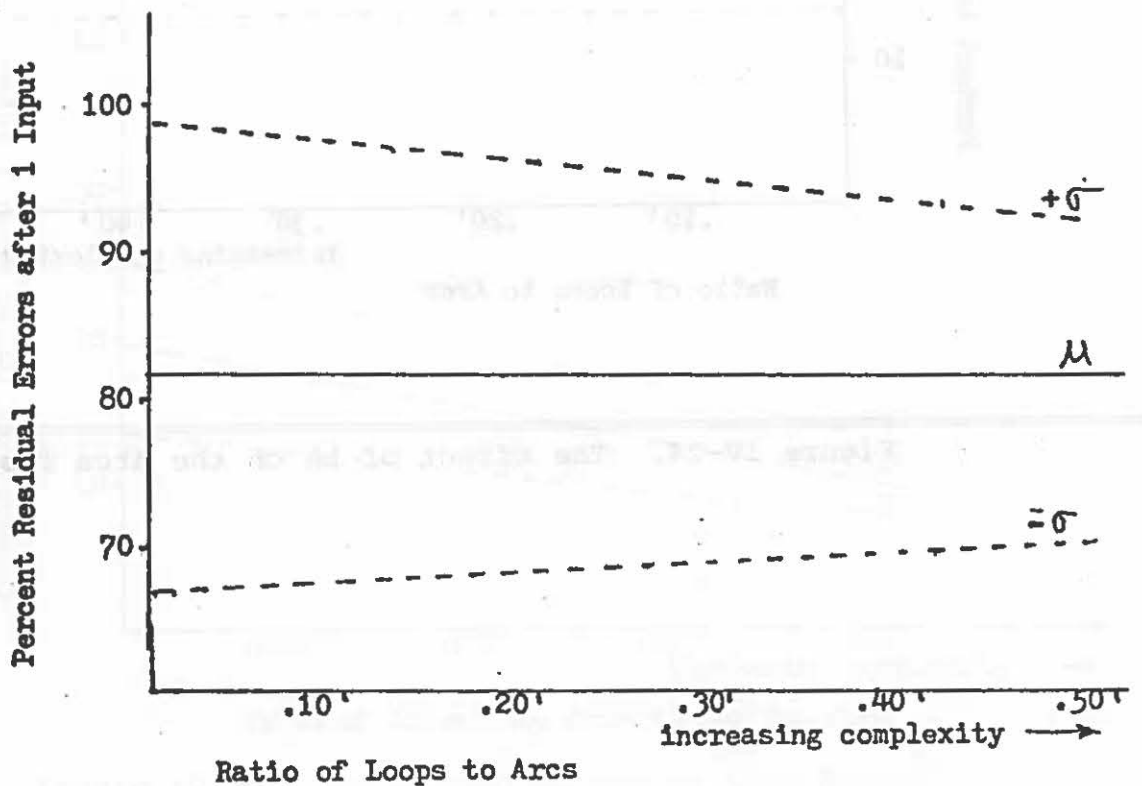


Figure IV-23. The Effect of LA on Residual Errors.



the idea that loops expose more arcs to testing at the same rate as the additional arcs increase the complexity. In Figure IV-24 it can be seen that the added complexity had no effect on the percentage of the arcs tested after one input, with the mean after 100 replications being a constant 22 percent. Since loops were also defined as arcs, the ratio of loops to arcs did not increase linearly as the number of loops increased.

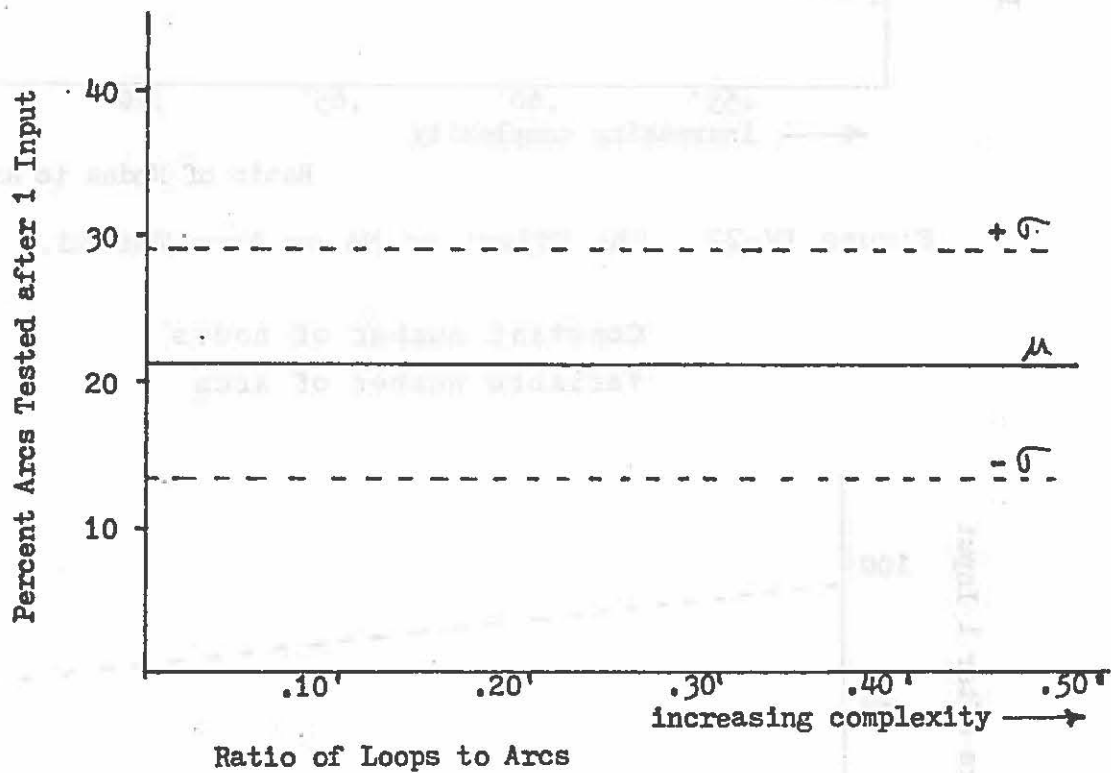


Figure IV-24. The Effect of LA on the Arcs Tested.

## V. ANALYTICAL RESULTS FOR THE ERROR DETECTION MODEL

### A. INTRODUCTION

In the previous section an error detection model was introduced and analyzed by simulation. It is possible to obtain analytical results for the expected number of errors detected by testing. Although the results are more limited than those from the simulation, the analytical results are a relatively inexpensive means to analyze the relationship between structure and the error detection process. The analytical results can also help in the statistical analysis of the simulation and can reduce the number of simulation runs needed.

The error detection model is reviewed. Then the analytical results are developed and the output of computer programs to do the calculation is discussed.

### B. ERROR DETECTION MODEL

Here we investigate how error detection during testing is affected by the structure of a computer program. By structure we mean how the parts of the program are related. It is very difficult to do experimentation with program structure in actual software projects because the cost of duplicate implementations of the same application is very high for all but small projects. For this reason analysis is performed on a model. Structure may be modeled as a set of nodes and arcs as was described in Section IV. An example is shown in Figure V-1.

Program structure affects the error detection process; to study this relationship it is helpful to have measures of each. For the error detection process some measures are: number of errors detected in a fixed time, number of errors detected with a fixed number of inputs, mean time between errors, percent arcs traversed

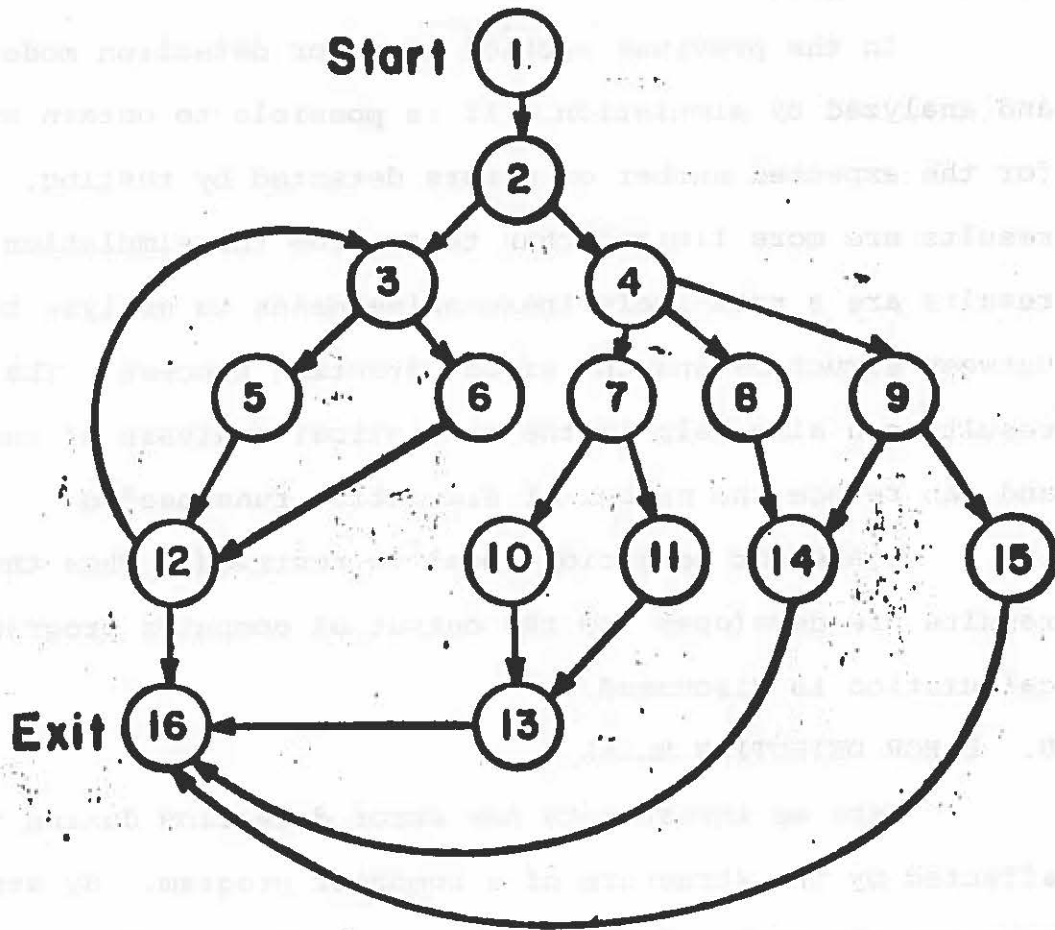


Figure V-1.

by one or more inputs and percent errors remaining. More sophisticated measures involving the shape of the graph of errors detected vs. time are also possible. These measures will be called "error detection characteristics." Good measures of program structure are harder to define. The most simple measure is size as measured by the number of nodes. A measure that expresses the degree of completeness of the graph is the ratio of the actual number of arcs in the graph to the maximum possible number with only one arc between pair of nodes. Since the model allows parallel arcs this number can be greater than one. These measures will be called "complexity measures."

Since measures of complexity are to be used to estimate error detection characteristics, it is important to define measures that adequately express the differences between structures with good and poor error detection characteristics. Since inputs are associated with paths, a measure of complexity is the number of paths. The average number of arcs per path also is a measure that is related to the number of errors detected per input. For moderate size graphs with no directed cycles, it is easy to enumerate all the paths and the number of arcs on each. If there are directed cycles, it is necessary to put an upper limit on the number of arcs in the paths considered in order to eliminate paths with an uncountable number of arcs. The number of ways that an arc can be reached indicates how accessible it is to testing. Measures based on this are the mean and standard deviation of the number of paths that traverse each arc or the number of arcs that are traversed by less than a fixed number of paths.

The complexity measures defined thus far depend only on the topology of the directed graph; it is also possible to use the branch probabilities. Since the paths are not equally likely, the measures involving paths can be weighted by the path probabilities. Given the probability that an arc will be traversed by a single input, a complexity measure of the accessibility of the arcs for testing is the sum of these probabilities for all arcs.

### C. ANALYTICAL RESULTS

Here we describe how to analytically calculate the expected number of errors detected by each of a sequence of inputs. For the purpose of simplifying the analysis, it is assumed that new errors are not created by the correction of errors. Given a computer program represented as a directed graph with branch probabilities, and given the expected number of errors on each arc, the output of the analysis is the expected number of errors detected. For example, Figure V-2 shows the results of the analysis for the graph of Figure V-1 where the expected number of errors in each arc is 0.6 and for each branch node the probability of taking each arc is equal.

The analysis is in two parts. First, it is necessary to calculate for each arc the probability that the arc will be traversed by an input. If there are no loops in the program, the calculation is easy: The probability of visiting the start node is 1, if there are  $k$  branching arcs the probability of traversing each arc is  $1/k$ . The probability of reaching any node is the sum of the probabilities on the arcs coming into that node. For example, for Figure V-1, the probability of reaching node 1 is 1, the probability of traversing

PROGRAM TITLE:      ERROR DETECTION MODEL

INPUTS = 20

NUMBER OF ARCS = 22

TAIL	ARC	HEAD	BRANCH PROB.	TRAVERSAL PROB.	EXPECTED NUMBER OF ERRORS
1		2	1.0000	1.0000	0.60
2		3	C.5000	0.5000	0.60
2		4	C.5000	0.5000	0.60
4		7	C.3333	0.1667	0.60
4		8	C.3333	0.1667	0.60
4		9	C.3333	0.1667	0.60
7		10	C.5000	0.0833	0.60
7		11	C.5000	0.0833	0.60
8		14	1.0000	0.1667	0.60
9		14	C.5000	0.0833	0.60
9		15	C.5000	0.0833	0.60
10		13	1.0000	0.0833	0.60
11		13	1.0000	0.0833	0.60
13		16	1.0000	0.1667	0.60
14		16	1.0000	0.2500	0.60
15		16	1.0000	0.0833	0.60
3		5	C.5000	0.3333	0.60
5		6	C.5000	0.3333	0.60
5		12	C.5000	0.3333	0.60
6		12	C.5000	0.3333	0.60
12		3	C.5000	0.2500	0.60
12		16	C.5000	0.5000	0.60

INITIAL EXPECTED NUMBER OF ERRORS =      13.20

INPUT	EXPECTED NUMBER OF ERRORS DETECTED	CUMMULATIVE EXPECTED NUMBER OF ERRORS DETECTED
1	3.45	3.45
2	1.95	5.40
3	1.39	6.79
4	1.03	7.82
5	C.80	8.62
6	C.63	9.25
7	0.51	9.76
8	0.42	10.19
9	0.36	10.54
10	0.30	10.84
11	C.26	11.10
12	C.22	11.33
13	0.20	11.52
14	0.17	11.69
15	0.15	11.84
16	0.13	11.98
17	0.12	12.10
18	C.11	12.20
19	0.09	12.30
20	0.08	12.38

Figure V-2

arc 1-2 is 1 and the probability of reaching node 2 is 1. There are 2 branches from node 2 so arc 2-3 and 2-4 each have probability 1/2. In this way the probability of reaching nodes 4,7,8,9,10,11,14 and 15 and the probability of traversing the arcs out of these nodes can be calculated; see column 5 of Figure V-3. The loop from node 12 to node 3 makes the analysis for arcs 3-5, 3-6, 5-12, 6-12, and 12-3 complicated because it is possible to return to node 3 more than once. The probability of reaching node 3 directly from node 2 is 1/2 and thus, there is a probability of 1/4 of immediately traversing arc 3-5. However, even if arc 3-6 is traversed there is some probability that after arc 6-12 is traversed arc 12-3 will be traversed and then arc 3-5 will be traversed. Fortunately, it is not necessary to do this calculation by hand. It is possible to do a Markov chain analysis to compute the probabilities and a computer program has been written to do this calculation. Figure V-3 is the output of that program for the graph of Figure V-1. The "R" in the column labeled repeat indicates an arc that may be traversed more than once by a single input. The branch probability column gives the probability of traversing the arc, having reached the tail node.

The second part of the analysis is to compute the expected number of errors detected. For notational convenience, the arcs are numbered  $j=1, \dots, n$ . Let  $p_j$  be the probability of traversing arc  $j$ . Let  $\mu_j$  be the expected number of errors in arc  $j$ . The expected number of errors detected by the first input is  $\sum_{j=1}^n \mu_j p_j$ . After the first input, the expected number of errors in each arc is reduced to  $\mu_{j2} = \mu_j (1-p_j)$  (where the 2 indicates this

PROGRAM TITLE: ERROR DETECTION MODEL

START NODE = 1      EXIT NODE = 16

NUMBER OF ARCS = 22

TAIL	ARC	HEAD	REPEAT	BRANCH PROB.	TRAVERSAL PROB.
6		12	R	1.0000	0.3333
12		3	K	0.5000	0.2500
11		13		1.0000	0.0833
13		16		1.0000	0.1667
14		16		1.0000	0.2500
7		11		0.5000	0.0833
8		14		1.0000	0.1667
1		2		1.0000	1.0000
2		3		0.5000	0.5000
12		16		0.5000	0.5000
2		4		0.5000	0.5000
4		7		0.3333	0.1667
4		8		0.3333	0.1667
4		9		0.3333	0.1667
7		10		0.5000	0.0833
9		14		0.5000	0.0833
9		15		0.5000	0.0833
10		13		1.0000	0.0833
15		16		1.0000	0.0833
3		5	R	0.5000	0.3333
3		6	R	0.5000	0.3333
5		12	R	1.0000	0.3333

FIGURE V-3



is the expected number of errors in arc  $j$  before the second input). The expected number of errors detected by the second input is  $\sum_{j=1}^n \mu_{j2} p_j$ . In general, for the  $k^{\text{th}}$  input the expected number of errors is  $\sum_{j=1}^n \mu_{jk} p_j$  where  $\mu_{jk} = \mu_j (1-p_j)^{k-1}$ . A computer program has been written to do the calculation and to draw the graph; Figures V-2 and V-4 are the outputs of that program.

#### D. LIMITATION OF THE ANALYSIS

The output of the model is an average; there is no information on what the distribution of the number of errors detected might be. It is possible to compute the standard deviation of the number of errors detected by the first input if the number of errors on each arc is independent of the number of errors on every other arc. However, this calculation is impractical for the second and subsequent inputs. The simulation may be used to estimate the distribution of errors detected.

#### E. COMPUTER PROGRAM

Two computer programs have been written to do the analysis. They are written in FORTRAN. The programs and directions for use are included in Appendix E.

EXPECTED NUMBER  
OF ERRORS DETECTED

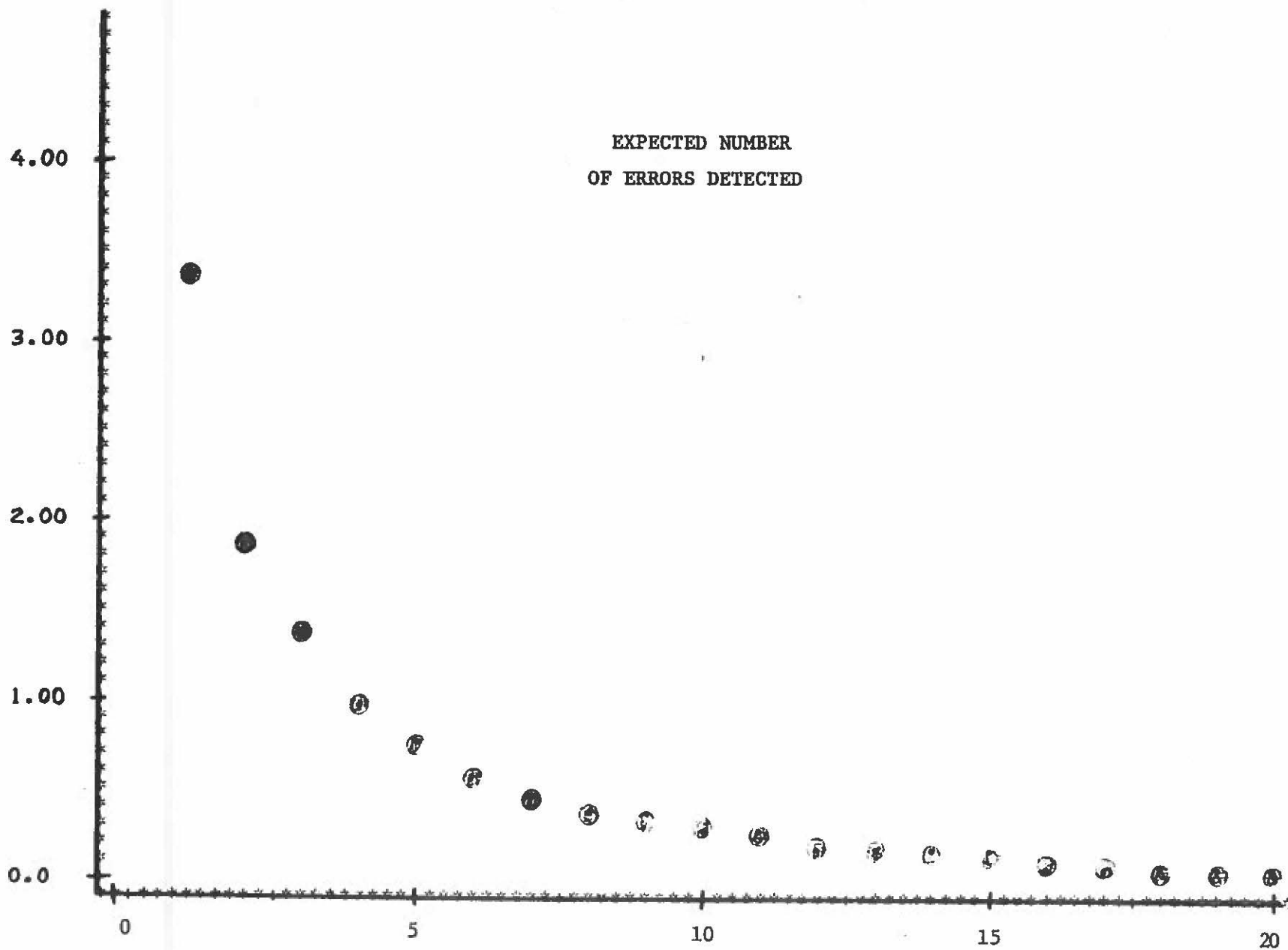


Figure V-4



## VI. ISSUES IN SYSTEM TESTING

There are a number of issues that arise in the development of system tests for avionics. Some of these issues are common to many projects and it is possible to develop guidelines for action without considering the details of the particular avionics system. Here we discuss a number of issues that have been important considerations in the development of system tests; the resolution of these issues has often been critical to the success of the system test effort.

It is helpful to have a scenario to relate the issues. Here we consider a piece of avionics equipment. It is like a computer--it has memory, it inputs and outputs data and it runs programs. It has a test procedure; that is the complete test plan including a computer program and operator manual. The test procedure consists of individual tests. The equipment is given a preflight test to determine if it is in the "go" or "no-go" condition. It has inflight tests to be used routinely and tests to restart in case of failure. It is installed on several platforms, some with repair capability, some with none.

### A. STRESS IN TESTING

Testing as well as operational use weakens components and produces failures. The time for testing can be a significant part of the "on time" for avionics equipment. Testing can involve high stress on the system--often testing purposely subjects the system to higher stress than during operation in order to (1) precipitate failures in weak components so they can be replaced before the mission begins, (2) reduce test time by testing items simultaneously and moving

quickly from one test to another and (3) detect malfunctions that might occur when the system is heavily loaded. One example of the use of high stress testing is the diagnostic tests run by maintenance engineers on general purpose computer systems. Another example is applying low voltages to equipment; this can precipitate failures (this is being used less frequently now since low voltage is particularly detrimental to solid state devices). There is, however, definitely a place for very low stress testing. The deep space missions employ very low stress testing in order to avoid failures due to testing.

For avionics, preflight testing is always higher stress testing than testing during a mission. Platforms with repair capability can use higher stress testing during a mission. During a mission, the decision to use high or low stress testing after a failure or after a power down depends on whether it is more important to have a precise indication of the capability of the equipment (use high stress) or it is more important to complete the mission with the equipment as is (use low stress). The use of the same tests during a mission as were used in preflight is not the best procedure in some situations; for some platforms it will be necessary to design tests with different levels of stress.

It is critical that the design specifications for the preflight and inflight tests indicate clearly whether high stress or low stress is to be used. A critical design variable is the time allowed for testing; a long detailed test plan together with a short test time will result in a high stress design. Avionic systems for different platforms may require a different level of stress for each platform.

## B. MICRODIAGNOSTICS

In avionics systems that have microprogramming, it is possible to implement test programs in software and/or microprograms. There are several reasons why microprograms are an attractive alternative to software: (1) Microprograms require much less hardware than software. Software may require input devices, data channels, main memory, etc., while microprograms only require a data path to a data register. (2) Microinstructions are much closer to actual hardware, so it is possible to get finer resolution in detecting and isolating errors. For example, it is possible to trap a machine language instruction before it is completed. (3) Microprograms are faster and require less storage than software because microinstructions can access basic hardware elements directly.

There are drawbacks to microprograms: (1) They are expensive and difficult to modify. (2) They are less visible to the user than software.

The use of microdiagnostics should be considered in any new system. It is good practice to write specifications that detail how the vendor will demonstrate that the microprograms are correct.

## C. HARDWARE

"Hardware" is defined to be that part of the system that must be fault-free for the test program to run and to output some test result. With a failure in hardware, the operator has no guidance on what is wrong except that it is in hardware. For most avionics systems, power supplies, some memory and arithmetic capability are part of hardware. The equipment designer's decisions determine the hardware; if the

designer optimizes the operation of the equipment with only secondary consideration for testing, the result can be a large hardcore that greatly hinders testing. One of the worst situations is when there is only one way to start the testing; if there is a failure in the hardcore, the operator has no alternate means to begin the test and he must begin a trial and error testing of the complete hardcore.

Hardcore can be classified into three categories. (1) Centralized--all the hardcore components are in the equipment being tested. (2) External--the tests are driven from outside the equipment being tested; it is possible to go through a series of tests even if part of the equipment has failed. (3) Distributed--several hardcores so that the tests can be initialized from any of them. External and distributed are clearly superior to centralized for testing purposes, since there is less chance the operator will be faced with a failure and have no guidance on what caused the failure.

A manufacturer's procedures for testing often make it appear that there is only one way to start the testing even when there may be several. The equipment documentation and test procedures should clearly document any alternate starting procedures in case of hardcore failure.

Microprogramming has allowed hardcore to be reduced; it is possible to initiate microprograms for testing and, thus, external storage is not required to store test routines. Also, input channels are not needed to bring in the test routines.

The decisions that determine hardcore(s) are usually made early in the design. Since hardcore design decisions are hard to change, it is critical that systems test personnel have input to the initial design decisions.

#### D. THE OPERATOR IN TESTING

Testing is a man-machine process where the man and the machine interact. The trend has been to give the machine more of the work because: (1) the machine is faster, (2) the machine is not bored by tedious tasks and (3) the machine can handle large amounts of detailed information. Also, the education and training of the operator has not kept pace with the greatly increased sophistication of the equipment.

Humans have some unique capabilities that machines have not been able to duplicate. Humans have amazing capabilities for "pattern recognition" and they learn from experience much more successfully than any machine built thus far. The ability of operators to detect errors, anticipate breakdowns, and to correct for drift in settings is well known. A talented operator is clearly very valuable in the man-machine system. In order for the man to contribute, the tests must produce information for him to use. Tests that give only go/no-go lights do not best use the unique capabilities of the operator. While still assuming that some operators will be inexperienced and untalented, the designer should have the machine produce information that will allow the motivated operator to fully participate in the test. Research in human factors gives guidelines for how to best use the man; for example, it is possible to provide so much information that the operator is overwhelmed. Also, the information provided must have some meaning, since humans tend to read meaning even into nonsense information.

The personnel responsible for systems testing should consider testing as a man-machine process and should bring human factors considerations into the design. In addition to the usual physical design



decisions, careful consideration should be given to the choice of information to be supplied to the operator during testing.

#### E. REDUNDANCY AND RESTART

The ability to operate after a failure has occurred is important for any avionics system. The object of redundancy is to have the performance of the equipment unaffected by certain failures in the equipment. The object of restart (or rollback) is to minimize the time and information loss after a failure.

In hardware, redundancy is accomplished by having two or more pieces of hardware that perform the same job; the output is a majority vote of the hardware pieces. The effect is to delay the repair of a failed piece of hardware until a more convenient time (e.g., until the aircraft lands). Another hardware approach is to check the output of a part of the equipment for errors and repeat the operation if there is an error or indication of an error. Error detecting and error correcting data channels are examples. Typically, this type of testing takes extra time. The use of standby units is also considered a form of hardware redundancy.

Software redundancy is sometimes implemented by doing a short approximate calculation to test the reasonableness of a long calculation; if there is a significant difference the calculations are done again.

Restart or rollback is accomplished by periodically (or upon signal) outputting critical information to a storage device. If a failure occurs that requires restarting, it is possible to rollback to the restart point or it is possible to restart the operation more quickly than would be possible without the saved information.

## F. DEGRADED MODE

Prototype testing establishes a definition of how the equipment should operate. Maintenance testing determines if the system still meets that definition. If the equipment is not functioning or if it is not functioning as it should, the operator sometimes must determine what part of the mission can be performed with the available equipment.

Although it is widely recognized that equipment must sometimes be operated in a degraded mode and it is widely accepted that the operator, not the test procedure, determines if the equipment is in a go or no-go condition, many test procedures stop if a "severe" error is detected (the test procedure determines what is "severe"). Test procedures should be written so that the operator can, with little effort, override any stop in the test sequence. He should be able to force the testing of any part of the system.

Design decisions affect the degraded mode operation. For example, if the equipment has two arithmetic units, is it possible to operate with just one? Is it possible to bypass or wire out a defective component? Is it possible to drive parts of the equipment externally?

An effective way to make sure that degraded mode issues are properly addressed is to put specific conditions into equipment specifications and acceptance tests. For example, in the acceptance test, faults could be placed in the equipment to observe the test procedure and degraded mode operation.

## G. INDEPENDENCE IN THE TEST PROCEDURE

Since time is so critical in the testing of avionics, the test procedures should be designed so that it is possible to run

some tests independent of the availability of some parts of the equipment and independent of the results of previous tests. The goal of complete independence is not attainable. It is necessary to have some tests that can be run only if certain parts of the equipment are operational. Also, some tests can be interpreted correctly only if several previous tests were successful. Nevertheless, it is very desirable that after a failure has been identified, the testing of other parts of the equipment can continue until the part has been repaired.

A test plan is called combinational if the sequence of tests is fixed. A test plan is called sequential if the sequence of tests depends on the outcome of previous tests. That is, after performing several tests, the next test to be performed is chosen by considering the outcome of some or all of the previous tests. Experience has shown that completely sequential testing is not practical for more than a few tests, because the test program becomes too large, too complex and too slow to justify the benefits of sequential testing. However, it is possible to do some very modest sequential testing by identifying a small number of tests (say 3-5) and then make the test sequence depend on the outcome of these tests. For example, if the test of the arithmetic unit failed, after reporting the failure to the operator, the test sequence could be modified to exclude all tests that needed the arithmetic unit. This would allow the testing to continue while the arithmetic unit was being repaired or replaced. This modest sequential testing offers advantages over the usual testing which is combinational or completely sequential.

#### H. AUTOMATIC ABORT

One reason that hardware failures and software errors are hard to locate is that considerable time may elapse until the failure or

error is detected. Until the error is detected, the contents of memory may be greatly modified by executing data, using incorrect data, etc. It is often very difficult to determine exactly when the error occurred or which instruction was being executed. Therefore, it is useful to be able to stop the equipment immediately after an error has occurred. Some equipment has included a special counter that must be reset periodically (e.g., 1 second real time) or the equipment stops (or turns on a light, or causes a dump of information to a backup storage). On some equipment a memory location is monitored; if it is not changed in a prescribed time, the equipment is known not to be performing correctly. This feature can be helpful in prototype testing when loss of control is frequent and difficult to diagnose.

The action taken when the counter stops the equipment should be nondestructive, since the reason for the stop may be that the equipment is severely overloaded. The operator should always be able to override the effect of the counter.



## VII. CONCLUSIONS AND RECOMMENDATIONS

This project has addressed the areas of prototype testing, maintenance testing, software error detection analysis (simulation and analytic models) and issues in systems testing. The purpose of each research effort has been to provide concepts or tools for improving the testing function. Collectively, these concepts and tools, when augmented by existing techniques, such as structured programming, provide test management with a systems test methodology. The important conclusions and recommendations pertaining to each research area will now be discussed.

### A. PROTOTYPE TESTING

This effort was concerned with the development of procedures and a simulation model to be applied in the planning of prototype testing. The procedural aspects involved the establishment of a terminology, symbology and directed graph representation for describing the module relationships which exist during prototype testing. The simulation model is designed to aid the designer and tester in identifying potential resource usage conflicts which would result in undesirable performance. This model has been successfully used for simulating the execution of a series of tasks, invoked by specified modules, which require the use of designated resources. We recommend that the next step in the model development be an investigation of the ability of the model to detect and diagnose faults which have been purposefully introduced. This would be followed by the application of the model to NADC prototype test planning.

## B. MAINTENANCE TESTING

The maintenance testing methodology which we have described is applicable primarily to those tests which are employed after a system has been delivered to the customer. The tests are invoked prior to or during a mission in order to ascertain the ability of the system to successfully complete the mission. The central idea of the methodology is to use the tests to successively partition the possible faults into subsets, so that the actual fault can be identified. We conclude that this methodology has potential for isolating both hardware and software faults. It appears that this technique could be used to develop test plans for module testing in addition to the maintenance testing application. It is recommended that the next step be the determination of the feasibility of the methodology as applied to the development of maintenance tests for a designated NADC system. This could involve the identification of a set of faults and possible tests such that the number of tests required for fault isolation is minimized.

## C. ERROR SIMULATION MODEL

We conclude from having exercised the error simulation model extensively that certain complex structures do have an adverse effect on the ability to detect errors and to provide adequate test coverage of a program. A next step would be the application of the model to software test planning at NADC. Actual programs which are to be tested would be put into the directed graph format, perhaps by an automated translation process as suggested by NADC, for input to the simulation program. The error detection characteristics of each program would

be simulated. The results would be related to measures of program complexity. The relationship between error detection and complexity would be used to allocate test resources to the programs. In addition, by using the model as described above, the model could be employed at NADC during the software design phase for the purpose of identifying the error detection characteristics of proposed program structures.

#### D. ANALYTIC ERROR DETECTION MODEL

The analytic model has the advantage of providing the expected number of detected errors, as a function of number of inputs, less expensively (CPU time and core) than with simulation. It can also provide a check on the validity of the simulation model. The disadvantage of the model is that it provides limited information concerning the variability of detected number of errors. We recommend that this model be applied in the same manner as the simulation model just discussed. The utility of each approach could be determined in an actual test environment. It is recommended that, initially, the analytic model be used in those situations where it is desired to rapidly obtain a ranking of the error detection characteristics of various programs. The error simulation model could be employed in those instances where greater detail in terms of path traversals, test coverage and error detection variability is desired.

#### E. ISSUES IN TESTING

This section presented a summary of certain key issues in systems testing, primarily those associated with maintenance testing and error recovery capability. Many of these issues are major concerns



of NADC in current test operations. For example, the use of micro-programming for error diagnosis has the obvious advantages of compactness of memory and speed of execution. However, the lack of visibility of diagnostics makes it difficult for NADC to validate vendor supplied products. We recommend that the issues which have been discussed be included as design and test factors during the design phase of future systems. This procedure would ensure the consideration of major test issues sufficiently early in the development cycle to have a beneficial effect on the testability of the delivered system.

## LIST OF REFERENCES

1. Dijkstra, E. W., "Structured Programming," Report of NATO Conference on Software Engineering Techniques, p. 84-87, 1970.
2. Campbell, D. J. and Heffner, W. J., "Measurement and Analysis of Large Operating Systems During System Developments," Fall Joint Computer Conference, p. 903-914, vol. 33 part 1, 1968.
3. Fishman, G. S., Concepts and Methods in Digital Simulation, p. 262-310, John Wiley & Sons, 1973.
4. Boehm, B. W., McClean, R. K., and Urfrig, D. B., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," Proceeding International Conference on Reliable Software, p. 105-113, 1975.
5. Vyssotsky, V. A., "Common Sense in Designing Testable Software," Program Test Methods, p. 41-48, Prentice-Hall, 1973.
6. Poole, P. C., "Debugging and Testing," Advanced Course on Software Engineering, p. 278-318, 1972.
7. Beizer, B., The Architecture and Engineering of Digital Computer Complexes, p. 641-665, Plenum Press, 1971.
8. The Polytechnic Institute of New York, Meaning of Exhaustive Software Testing, by Shooman, M. L., p. 10, January 2, 1974.
9. Schneidewind, N. F., "Analysis of Error Processes in Computer Software," Proceedings International Conference on Reliable Software, p. 337-346, 1975.
10. The Rand Corporation, Software and Its Impact: A Quantitative Assessment, by Boehm, B. W., p. 51, December 1972.
11. The Rand Corporation, Some Information Processing Implications of Air Force Space Missions: 1970-1980, by Boehm, B. W., p. 45, January 1970.
12. Bradley, G. H., Green, T., Howard, G. T. and Schneidewind, N. F., "Structure and Error Detection in Computer Software," Proceedings AIIE National Conference, p. 54-59, 1975.
13. Hetzel, W., "A Definitional Framework," Program Test Methods, p. 7-10 Prentice-Hall, 1973.
14. Dijkstra, E. W., "Notes on Structured Programming," Structured Programming, p. 1-82, Academic Press, 1972.

15. Mills, H., "Top Down Programming in Large Systems," Debugging Techniques in Large Scale Systems, p. 41-53, Prentice-Hall, 1971.
16. Gruenberger, F., "Program Testing: The Historical Perspective," Program Test Methods, Prentice-Hall, p. 11-15.
17. Schneidewind, N.F. and Green, T.F., "Simulation of Error Detection in Computer Programs," Proceedings of the Symposium on the Simulation of Computer Systems, National Bureau of Standards, 1975, six pages.
18. Herbert Y. Chang, et al., Fault Diagnosis of Digital Systems, John Wiley and Sons, 1970.
19. Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, v. , p. 48-59, May 1973.
20. Boehm, B. W., "The High Cost of Software," Proceedings of a Symposium on the High Cost of Software, Jack Goldberg (ed), Stanford Research Institute, p. 27-40, 1973.
21. Schneidewind, N.F., "An Approach to Software Reliability Prediction and Quality Control," AFIPS Conference Proceedings, v. 41, Part II, Fall Joint Computer Conference, p. 837-838, 1972.
22. Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, v. 11, No. 1, p. 65-66, 1972.
23. Rizza, J.B., and Hacker, D., "Quality Assurance Inspection and Test Tools - An Application," Proceedings of a Workshop on Currently Available Program Testing Tools, v. 1, p. 9-10, April 1975.
24. Howden, W. E., "Systems for Automating the Generation of Program Test Data," Proceedings of a Workshop on Currently Available Program Testing Tools, v. 1, p. 37-39, April 1975.
25. Stucki, L.G., "Automatic Generation of Self-Metric Software," Record of 1973 IEEE Symposium on Computer Software Reliability, New York City, p. 94-100, April 30 - May 2, 1973.
26. Von Alven, W. H., (ed), Reliability Engineering, p. 155-156, ARINC Research Corporation, Prentice-Hall, 1964.
27. McCormick, J. M. and Salvadori, M.G., Numerical Methods in FORTRAN, p. 290-295, Prentice-Hall, 1964.

INITIAL DISTRIBUTION LIST

	Copies
Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
Dean of Research Naval Postgraduate School Monterey, California 93940	4
W. R. Church Library Code 0211 Naval Postgraduate School Monterey, California 93940	2
Philip J. Kiviat Technical Director Department of the Air Force AFDA Center Washington, D. C. 20330	1
Professors G. Howard G. Bradley N. Schneidewind Code 55 Naval Postgraduate School Monterey, California 93940	2 each
Professors J. Esary D. Gaver R. Richards Code 55 Naval Postgraduate School Monterey, California 93940	1 each
Professor U. Kodres Code 72 Naval Postgraduate School Monterey, California 93940	1
Professor M. Powers Code 52 Naval Postgraduate School Monterey, California 93940	1
Professor G. Barksdale Code 72 Naval Postgraduate School Monterey, California 93940	1

Library Code 55  
Naval Postgraduate School  
Monterey, California 93940

1

Knox Library  
Naval Postgraduate School  
Monterey, California 93940

2

Prof. D. Williams Code 0211  
Naval Postgraduate School  
Monterey, California 93940

1

T. Green  
3246 Fenelson Street  
San Diego, California 92105

1

G. Montgomery  
510 Forest Heights Drive  
Knoxville, Tennessee 37919

1

Naval Air Development Center  
Warminster Pennsylvania 18974  
Attn: Mr. H. Steubing

1

Naval Air Development Center  
Warminster, Pennsylvania 18974  
Attn: Mr. R. Pariseau

3

1 each

Defense Documentation Center  
Cameron Station  
Alexandria, Virginia 22304  
1  
Head of Research  
Naval Postgraduate School  
Monterey, California 93940  
1  
W. K. Church Library  
Code 0211  
Naval Postgraduate School  
Monterey, California 93940  
1  
Philip J. Fowler  
Head Director  
Department of the Air Force  
AFMA Center  
Washington, D. C. 20330  
3  
Professor J. H. ...  
Code 57  
Naval Postgraduate School  
Monterey, California 93940  
1  
Professor J. H. ...  
Code 57  
Naval Postgraduate School  
Monterey, California 93940  
1  
Professor U. ...  
Code 15  
Naval Postgraduate School  
Monterey, California 93940  
1  
Professor M. ...  
Code 57  
Naval Postgraduate School  
Monterey, California 93940  
1  
Professor G. ...  
Code 15  
Naval Postgraduate School  
Monterey, California 93940  
1

