COMPLETE SAFETY SOFTWARE TESTING: A FORMAL METHOD

by

JON R. LUNGLHOFER

B.S., United States Naval Academy, Annapolis, Md. (May 1994)

SUBMITTED TO THE DEPARTMENT OF NUCLEAR ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February, 1996

Signature of Author ______________________________
Department of Nuclear Engineering
January, 1996

Certified by: ______________________________
Professor Michael W. Golay
Thesis Supervisor

Certified by: ______________________________
Professor David D. Lanning
Thesis Reader

Accepted by: ______________________________
Professor Jeffrey P. Freidberg
Chairman, Department Committee on Graduate Students

COMPLETE SAFETY SOFTWARE TESTING:

A FORMAL METHOD

by

JON R. LUNGHLHOFER

Submitted to the Department of Nuclear Engineering
on January 18, 1996 in partial fulfillment of the
requirements for the Degree of Master of Science in
Nuclear Engineering.

Abstract

In order to allow the introduction of safety-related digital control in nuclear power reactors, the software used by the systems must be demonstrated to be highly reliable. One method of improving software reliability is testing. A study of the complete testing of software was performed based upon past research. The literature search found two complete testing methods, only one of which, complete path testing, was feasible for use. The literature contained a practical and nearly complete testing method known as Structured Path Testing, developed by Howden (Howd77). Structured Path Testing was adapted here as the basis for a new testing method known as Feasible Structured Path Testing (FSPT). The FSPT involves a five step formal method which examines the code's data flow structure and develops test cases based upon the analysis. The FSPT eliminates unfeasible paths from the test case which greatly speeds the testing process.

The FSPT was applied to several simple programs. Many of the test programs are written in the OO1 case tool language developed by Hamilton Technology (Hami92.). The ultimate example test bed program, known as the Reactor Protection System (RPS) was written by M. Ouyang (Ouya95).

The work reported here was restricted to the examination of software code testing. No attempt was made to examine the reliability of corresponding hardware or human interactions. In order to improve reliability of safety-critical software, one must assure that human interactions are simple and that hardware is sufficiently redundant.

The results of the study presented here indicate that for simple programs, complete testing is possible. If looping structures are included in a program, in most cases, the tester must sacrifice complete testing in order to complete the testing process within the programs lifetime. While the scope of this study was limited, there is no reason to believe the FSPT could not be used for much larger tasks. The FSPT greatly enhances the reliability of software code.

Thesis Supervisor:	Michael W. Golay
Title:	Professor of Nuclear Engineering

Acknowledgments

I wish to thank my thesis supervisor, Professor Michael W. Golay for his continued direction and support throughout this study. Without his financial, technical, and personal assistance, this work could not have gone forward.

I wish to thank ABB-Cumbustion Engineering for their financial support which allowed this work to go on. Special thanks go to Michael Novak, the primary contact at ABB-CE. His feedback and clear focus helped to focus the goals of this research.

Hamilton Technology Inc. also deserves thanks for supplying the OO1 system for use in the project. Special thanks goes to Margaret Hamilton for her generous offer of the OO1 evaluation system package; to Ron Hackler, Nils, and Zander for their continued technical assistance, both on-line and off.

Thanks also go to Meng Ouyang for his patience, helpfulness, and companionship throughout my stay at MIT. I enjoyed working with Meng and wish him the best of luck in his future endeavors. Thanks also go to Chang Woo Kang for being a true friend, (Kamsa Hamnida). Thanks to Rachel Morton for her continued support in drafting this thesis.

Finally, I wish to thank my family at home and away from home. Thanks Mom for all your love, I couldn't have made it without you. Thanks go to David Fink for begin an excellent room mate while at MIT. We shared some excellent time together. To all my friends at the Boston Rock Gym, whom are too numerous to list, I thank you all for your companionship. To Tina Burke, my true inspiration, I thank her for her constant love and support. Climb On.

Table of Content

List of Figures

In Order of Appearance

List of Tables
In Order of Appearance

1. Introduction

1.1 General

Due to the difficulties of verification and validation (V& V), the nuclear power industry has lagged in the introduction of digital control software to regulate the safety systems of nuclear power plants. The (V& V) process includes testing the software against design specification performance requirements. Software designed for use in nuclear power plant safety systems should be completely tested to prove its correctness. Without a method of verifying a software system, the nuclear industry continues development without the advantages of modern digital control technologies common to other industries.

Extensive methods have been developed to determine the reliability of the analog safety systems currently being utilized in nuclear power plants. The debate over the introduction of digital control software systems stems from the question of reliability. In order for digital control software to replace the current analog systems, the software must be proven to be as reliable. Given the complexity of the (V& V) process, this is a difficult task.

The development of highly reliable software systems has been of great interest to those operating large, complicated, and expensive systems that are traditionally prone to human error. Such systems include fly-by-wire aircraft and telecommunication systems. Due to the large size of these programs, it is considered impossible to completely test them. These programs are tested until the expected frequency of error detection is longer than the program's expected lifetime. This method of testing does not offer an all together acceptable standard of reliability. The U.S. Nuclear Regulatory Commission (NRC) holds the nuclear industry to a higher standard.

This work seeks to develop, or find, an existing method for completely testing safety-critical software. Also, the work will, in the absence of a complete testing method, demonstrate the best method available. The method of choice will be tested on simple pieces of safety software. The art of software development will also be examined. The complexity of a piece of software corresponds with its testability. It is possible to design software that it is easily testable, or nearly impossible to test. This report examines methods for developing software that improve the ease of testability. This work leads to an improvement in nuclear power plant safety by paving the way for the use of modern digital technology.

1.2 *Background*

There are many ways in which software can contain errors, and many ways to test for those errors. The ultimate goal is to locate every error in a piece of safety-related software to allow for its use in a nuclear power plant. If safety-related software can be demonstrated as highly reliable with very little uncertainty, it would likely become an integral part of the nuclear power plants of the future. However, due to the nature of software systems, partial proofs with high uncertainty will be insufficient. Software errors, unlike hardware faults, are always present and waiting to be unearthed. Where hardware can be inspected to reveal faults, software must be tested against the specification document to demonstrate errors. Since undetected software errors often lie in rarely used sections of code, such as accident sequence responses, it is imperative that all the errors be detected in the (V& V) phase of software development.

In order to test a piece of software for errors, one must first understand the types of errors that can occur. Software faults can result from the following sources of error:

- Incorrect formulation of the program specifications.
- Operating system errors.
- Environmental errors.
- Input / output errors.
- Programming errors, such as:
    - syntax errors
    - logical errors
    - numerical errors (division by zero, etc.)

There are many documented ways of searching for errors. The methods can be broken down into two main categories, black box and white box testing. Black Box testing treats the program as if it were a closed box. There is no examination of the code during the testing process. Test cases are generated purely from the specification document. The results of these test cases are compared with a database of correct results to determine if the correct output was obtained. This database is commonly referred to as an oracle. The existence of an oracle is necessary for all types of testing. The test cases generated during white box testing result from an examination of the code structure. White box testing also requires an oracle to assure that the test results are correct.

Many other methods exist for improving software reliability. Some of these methods are not based on testing and include:

- Developing two different programs on two or more different operating environments from the same specifications. The two software systems are then compared to each other. Every differences between the two codes is a potential sources of error and must be addressed.
- The code is checked independently line by line for syntax and conceptual errors.
- Graphic oriented formal development languages are used to develop the code which is internally consistent and complete in capture of the specifications.
- Programs are modularized to decrease the complexity of the code. Small pieces of code are less prone to errors.

There are many sources of error in the software development process. Even using the most careful of development methods, errors pass into the final software code. It is necessary to test for these errors. Only when a complete test of the software against its specification document is completed can one be assured of correct performance.

1.3 Method of Investigation

The effort reflected in this work is the result of a literature search of all documents pertaining to software testing. The search covers a broad range of software testing methods and styles. The aim of the search is to identify, test, and possibly improve upon the most efficient and complete testing method currently documented in the literature. Methods for developing reliable software are also examined. The literature is discussed and summarized.

An examination of several simple safety-related programs is utilized to illustrate the effectiveness of chosen testing methods. Other examples are used to demonstrate the performance of other testing methods described in the literature.

1.4 Scope of the Work

There are many sources of software errors external to the software development process. This work focuses upon the testing of the software code for programming errors. The testing of the hardware, operating system, and environment are considered beyond the scope of this work and will not be included. While these types of errors are quite common in complicated software systems, it is assumed that the environment in which the software operates is in working order.

The benefits of designing simple and easily testable modular software is examined here. Poorman's work (Poor94) demonstrates the usefulness of the software complexity metric. The complexity metric offers a formal method of gauging the complexity of a piece of software. Using a metric, one can assure that sections of software code do not become overly cumbersome and difficult to test.

Although it is generally considered impossible to completely test a piece of complex software, this work demonstrates that if software is written simply and correctly it can be thoroughly tested. If the software specifications document is written incorrectly or incompletely, one can not assure correct software performance. The testing method described in this report requires the presence of an oracle. The development of an oracle is a topic deserving of a study such as the one reported here. The presence of a proper specification and oracle is assumed.

2. Literature Search

2.1 *General*

This section summarizes the literature search of software testing methods used as the starting block for this study. The literature search for the work reported here was broad in scope, yielding many possible methods for testing software code. The goal of the search was to identify all of the available testing methods, and assess the possibilities based upon completeness, practicality, and overall usefulness. The research lead to many articles discussing reliable software development methods. Even though they do not directly relate to testing, these articles were given attention. A more reliable development process can only lead to less errors in the code which simplifies the testing process. Experience shows that in many cases software can not be completely tested, because of the over whelming number of test cases necessary. There are also hardware and user interaction concerns. Finally, there is the concern that the specifications document may include errors which can be passed on to the software. Since this study focuses only upon the complete testing of software code, other concerns, such as hardware and human interaction, have been eliminated from the search.

Thus, the search focuses on methods for testing software code and the development of coding practices which ease the testing process. The results of the search include two complete testing methods. One of these, exhaustive input testing, is applicable only in very simple theoretical cases making it incompatible with this study's requirements. The other method, complete path testing, is applicable in some simple cases which do not included looping structures.

In programs which involve loops, the Strucutred Path Testing technique sacrifices completness for timeliness, but is an excellent compromise which greatly improves program reliability.

The search begins with a review of previous research work done at MIT, the initial focus being the work of Poorman (Poor94) and Arno (Arno94). Both use a cyclomatic metric based testing schemes to yield complete testing results. The cyclomatic metric, also known as the McCabe metric for its developer T. J. McCabe, is explained in detail later in this chapter. The use of the cyclomatic metric is very helpful in gauging the difficulty of testing a piece of code. While helpful in determining the minimum number of tests needed for certain types of code testing, the McCabe metric based testing technique used by Poorman and Arno does not reliably provide a complete code test. The McCabe metric testing method does not test every possible execution path through the code. Using the standard control flowgraph, the McCabe metric testing technique used by Poorman and Arno guarantees branch coverage. It is possible that untested paths contain errors. The concepts of branch and path coverage as well as the control flowgraph are covered later in **Section 2**. The McCabe metric based technique is explained and its limitations fully demonstrated later in this chapter. Recall that the focus of our search is to locate a complete testing method. The results of Poorman and Arno do not yield a complete testing method. Thus, our search expands to examine all types of code testing methods in hopes of locating a complete method.

2.2 Software Development Methods

In recent years, the use of formal methods to develop software has become widespread. The ad hoc development methods of the past have become obsolete as programs have increased in size. For small and simple programs the ad hoc approach will never be replaced. However, for safety critical software systems, it is necessary to have a highly reliable form of quality control.

The software development process begins with the establishment of a requirements or specification document and continues through the design, testing, and documentation stages (Poor94). While the main focus of the work reported here is upon the testing phase, the design phase is also examined in **Section 2.2.2**. The software design process has been studied here in hopes of identifying ways for decreasing the introduction of errors into the final testable piece of software, thus easing the testing process. A broad range of topics can be classified as software development methods and they are examined here.

In a report prepared for the Nuclear Regulatory Committee (NRC) concerning the preparation of a severe reactor accident computer code, Sandia National Laboratories described a method for the reliable management of a large software development project (Qual91). Described within the report, is the management plan for treating situations which may be encountered during software development. Such procedures are essential with large projects dealing with many individuals. The main goal of the management plan is to deal with changes in the software as it is being developed. The document says little about the specific development of the software system, and focuses upon human interactions within a large programming team. While it is necessary to have methods for the administration of large software projects, such managerial details are beyond the scope of this study. The goal here is to develop methods which are specific to the coding process itself.

2.2.1 **Structural Requirements**

It has been demonstrated that the number of errors in a piece of code is related to the complexity of the section of code in question. A study was conducted by Lipow that demonstrated the existence of a non-linear relationship between the size of the program and the number of errors or bugs, found within the code (Lipo82). He demonstrated that smaller programs had a much lower error rate than had larger programs. He found that as the number of lines of code increased to around 100 the number of errors per line grew at a linear rate. However, in larger programs (> 100 lines), the number of errors per line increased in a non linear fashion. He concluded that shorter programs have an advantage over larger ones in terms of the amount of errors found per line. The practice of writing modular code takes advantage of these findings. However, in the literature there are many different ideas addressing when a program should be broken into modules. The differences stem from a general disagreement over how to measure the complexity of a piece of code. The question is: "How . . . [does one] . . . modularize a software system so the resulting modules are both testable and maintainable?" (McCa76)

The complexity of a piece of code is often measured by a software metric. A software metric is a number generated by a method or technique that supplies information about the complexity of the code in question. For his study, Lipow used the lines of code metric, which is equal to the number of lines of written code. The lines of code metric is utilized by several companies to govern the maximum allowable module size (Beiz90). This metric is the simplest, but can often underestimate the true complexity of the code in question. Consider a twenty five line program that consists of twenty five consecutive if . . then statements. This program contains 2^{25} possible paths of execution. Merely measuring the number of lines of code is not enough to judge the complexity. Many other metrics have been developed to aid the programmer in deciding when to modularize.

These metrics are numerous and can be categorized as either linguistic metrics or structural metrics (Beiz90). A linguistic metric is based upon measuring various aspects of the program's text. For example, the basis of a linguistic metric could be the number of lines of written source code. The linguistic metric is based upon any aspect or combination of aspects of the physical text layout. The structural metric is based upon the structural relationship between program objects. The structural metric is usually based upon understanding the programs control flow graph. For example, a structural treatment may gauge the extent of recursive structures in the code by examining the flow paths shown in the control flow graph. Two of the most useful metrics Halstead's (a linguistic metric) and McCabe's (a structural metric) are discussed here. The explanation of McCabe's metric is prefaced by a review of the control flowgraph.

2.2.1.1 *Halstead's Metric*

Halstead's metric is the best established of all linguistic metrics (Beiz90). Halstead developed the metric first by examining the patterns of errors in a large and diverse set of programs (Hals77). Halstead then took two simple linguistic measurements from each program, the number of program operators (e.g. keywords such as **Repeat..Until and if..then)**, and the number of program operands (e.g. variables and data types).

n1 = the number of distinct operators in the program.

n2 = the number of distinct operands in the program.

These two measurements were then used in a formula derived from an empirical fit to the actual error data. The hope was that the number of errors in a program could be predicted with relative accuracy using the metric. The metric gives a value designated as H

$$H = n_1 \log_2 n_1 + n_2 \log_2 n_2 \qquad \textbf{(2.1)}$$

which Halstead called the "program length." The "program length" H should not be confused with the number of lines of code in the program.

Halstead also defines two move parameters based on the total number of operator and operand appearances in the program text.

N_1 = Total operator count

N_2 = Total operand count

Halstead also defines a new metric

$$N = N_1 + N_2 \tag{2.2}$$

known as the actual Halstead length (Beiz90). Halstead's claim is that the value of H will be very close to the value of N. This has been confirmed by several studies. Thus, if the programs makeup is known it is possible to calculate an estimated length of the finished program, before it is written.

Halstead also developed a calculation to predict the number of errors present in a program (Beiz90). The formula

$$Errors = \frac{(N_1 + N_2)\log_2(n_1 + n_2)}{3000} \tag{2.3}$$

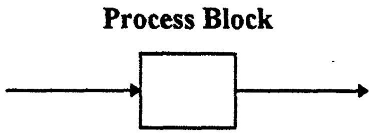
is based on the four values: n_1, n_2, N_1, and N_2.

For example, if a program has 75 data object and uses them 1300 times and 150 operators which are used 1200 times, then according to **Equation 2.3** there should be $(1300+1200)\log_2(75+150)/3000 = 6.5$ errors in the code.

Halstead's metric is very useful for the management of a large programming project, but is not such a useful tool with regard to completely testing that software. The metric does not supply the user with knowledge of the true number of errors, only an estimate. There is of course no way to have 6.5 errors in a piece or software. Thus the Halstead metric is a useful tool, but is unable to help adequately with the complete testing process. One could use Halstead's error estimation metric as a gauge of when to stop testing the code (Poor94) (e.g. stop testing after the estimated number of errors are found). However, such a technique seems far from complete. Halstead's metric is a useful tool for measuring and comparing the complexity of software codes being considered for the same job. The simplest code should be used. Although Halstead's metrics are useful, they do not facilitate the complete testing of a piece of software code, and is not used in this study.

2.2.1.2 The Control Flowgraph

Understanding the control flowgraph is central to understanding much of the literature on software testing. In general, the control flowgraph is a graphical representation of the control structure of a program. Before the control flowgraph can be discussed it is first necessary to introduce some standard terminology. The following terms are used to describe various aspects of computer code:

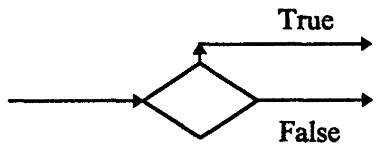


A section of code in which the statements are executed in order from the entrance to the exit is known as a Process Block.

Figure 2-1 Process Block Diagram

The process block is composed of a group of statements which must be executed in the order presented once the first statement is entered. In a less formal sense, a process block is also

Decisions



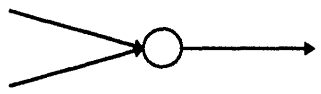
A decision is a section of the code where flow diverges, and is ussually expressed in the following manor: If (condition) then (execute the True path) else (execute the False path).

Figure 2-2 Decision Block Diagram

known as 'straight line code.' The length of the process can vary from two to hundreds of statements. There is no way to enter into a process block other than from the entrance.

The Decision represents a section of the code where flow can diverge into different paths. The decision is often binary in nature, but can be multi directional. (e.g. a case statement) The standard decision construct is the if..then..else statement which relies on conditions of certain variables to decide which path will be executed.

Junctions



When program paths merge together, a Junction is formed.

Figure 2-3 Junction Diagram

The Junction is a piece of code where flow paths can merge together. For example, when the separate true and false paths end after an if..then statement, the flow can merge together again. These merges are known as Junctions.

All programs can be broken down into some combination of these three elements. In some cases statements can be both decisions and junctions, such as loop control statements. The terminology used above is taken from the flowchart concept. The flowchart has become outdated and is being replaced with the control flowgraph. The control flowgraph greatly simplifies the flowchart concept. There are only two types of components to the flowgraph: circles and lines. A circle is called a node and line a link. A node with more than one link entering is a junction and one with more than one leaving a decision. The node with only one link entering and one link exiting is a process node. All code structures can be expressed using the control flowgraph.

An example is used to illustrate the control flowgraph. For this example a written program must be examined. To facilitate this, a generic and simple "programming language" is adopted. The language used is based on that developed by Rapps and Weyuker in 1985 (Rapps85). Using this language should break down any language barriers other high level languages might cause. The following statement types are included in the language:

The simple program detailed in **Figure 2-5**, written in the language illustrated in **Figure 2-**

Elementary programming language for exemplary use.

- Begin statement: **Begin**
- Input statement: **Read** $x_1, . . . , x_n$
  ($x_1, . . . , x_n$) are variables.
- Assignment statement: $y <= f(x_1, . . . , x_n)$
  ($y, x_1, . . . , x_n$ are variables) and f in a function.
- Output Statement: **Print** $z_1, . . . , z_n$
  ($z_1, . . . , z_n$)are either littorals or variables.
- Unconditional Transfer Statement: **Goto m**
  m is a label.
- Conditional Statement: **If** $p(x_1, . . . , x_n)$ **Then**

  ```
  If p(x1, . . . , xn) Then
        statements executed if predicate p is true
  Else
        statements executed if predicate p is false
  End If
  ```

  p is a predicate on the variables ($x_1, . . . , x_n$). Conditional Statements can be nested.
- End Statement: **End**

Figure 2-4 Simplified Example Programming Language

4, is used to illustrate the control flowgraph concept. The program reads the customers bank account balance (b) and then queries the customer for an amount to be withdrawn (w). The program then checks to see if the withdrawal amount is larger than the account balance. If so, no withdrawal is made. If the withdrawal amount is equal to the account balance, then the user is notified that the final account balance will be zero. If the withdrawal not more than the balance, then the balance is reduced by the amount (w). The illustrative program is listed in **Figure 2-5**.

```
Begin
Read (b)
Print ("Enter amount to withdraw.")
Read (w)
withdraw <= true
If (w > b) Then
   Print ("Amount to withdraw exceeds balance.")
   withdraw <= false
Else
   If (w < b) Then
      Print ("Thank You.")
   Else
      Print ("Thank You, Your final balance will be $0.00")
   End If
End If
If (withdraw)=true Then
   b <= b - w
Else
   b <= b
End
```

This code is written in the language specified in **Figure 2-4** and is used as an example in order to further the understanding of control flowgraphs.

Figure 2-5 Automatic Teller Machine (ATM) Example Code

The simple program above can be broken down into its control flowgraph. **Figure 2-6** illustrates the control flowgraph.

ATM Code Control Flowgraph Breakdown

Statements

```
Read (b)                                             A
Print ("Enter amount to withdraw.")
Read (w)
withdraw <= true
If (w > b) Then

  Print ("Amount to withdraw exceeds balance.")      B
  withdraw <= false

Else

  If (w < b) Then                                    C

    Print ("Thank You.")                             D

  Else

    Print ("Thank You, Your final balance will be $0.00")   E

  End If
End If

If (withdraw)=true Then                              F

  b <= b - w                                         G

Else

  b <= b                                             H

End                                                  I
```

Flowgraph

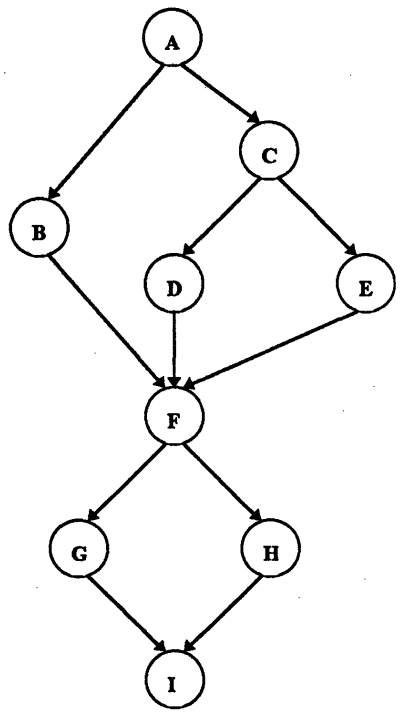


Figure 2-6 ATM Example Code and Flowgraph

The statements which are represented by each node are contained in the labeled boxes on the left. For example the node labeled **C** corresponds to the statement "If (w < b) Then." The simple procedure for creating control flowgraphs from code should be clear. With practice, generating control flowgraphs from written code becomes second nature.

Many terms used in software engineering relate to the control flowgraph (Rapps85). The links on the control flowgraph are also known as *edges*. An edge is a connection between nodes. An edge from node *j* to node *k* is given the notation (*j* , *k*). Node *j* designated as the *predecessor* of node *k*, and node *k* the *successor* to *j*. There can be only one edge between any two distinct nodes. The first node of a program is known as the *start node*, and has no predecessors while the last node is known as the *exit node* and has no successors.

A *path* or *flowpath* is a finite sequence of nodes connected by edges. A path is designated by a sequence of nodes. For instance, a possible path through the ATM Code is Path 1 = (A+B+F+G+I). Path 1 is a *complete path* because the first node is the start node and the last node the exit node. Path 1 is also a *loop-free path* because none of its nodes are repeated. If any node in a path is repeated, the path contains a looping structure. It is possible that such a path could contain an infinite amount of nodes. Such a path never reaches an exit node and is the result of a *syntactically endless loop*. There is no possible escape from such loops, and they often result from program errors.

The control flowgraph, or flowgraph as it is termed hereafter, is the foundation of structural metric analysis. This section should be reviewed if questions remain on the use and construction of flowgraphs.

2.2.1.3 McCabe's Metric

The McCabe's metric is a structure-based metric developed by Thomas J. McCabe (McCa76). McCabe noted that many software engineering companies had trouble deciding how to modularize there software systems. Many used the length-of-code metric, which seemed illogical to McCabe. Some programs are much more complex that others of the same length. McCabe hoped that his metric would give a more accurate measure of program complexity.

The calculation of McCabe's metric is a simple process. The McCabe metric is derived from the flowgraph of the software in question. McCabe defines his metric **v** as

$$\mathbf{v} = \mathbf{e} - \mathbf{n} + 2\mathbf{p} \tag{2.4}$$

where

e is the number of edges or links,

n is the number of nodes,

p is the number of connected flowgraph components.

Edges and nodes have been explained previously, but the value p may be unclear. Consider the following example (**Figure 2-7** consisting of a main program (M) and two subroutines (A and B). The flowgraph has three detached sections and thus the value of p is equal to three.

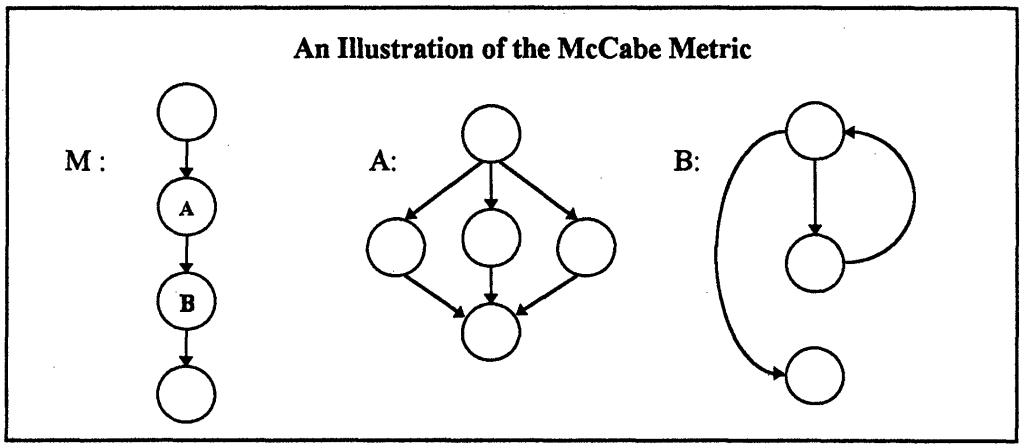


Figure 2-7 McCabe Metric Example Flowgraphs

Let us calculate the value of the McCabe's metric for each individual section in the illustration. The main program M contains three edges and four nodes in only one section. The metric for M takes the value

$$v(M) = 3 - 4 + 2(1) = 1 \ . \tag{2.5}$$

The McCabe metric values for sections A and B separately are

$$v(A) = 6 - 5 + 2(1) = 3 \quad \text{, and} \tag{2.6}$$

$$v(B) = 3 - 3 + 2(1) = 2 \quad . \tag{2.7}$$

If A and B are considered sub operations of the main program M, then the value of the McCabe metric of M including A and B is

$$v(M \cup A \cup B) = 12 - 12 + 2(3) = 6 \ . \tag{2.9}$$

Notice that in this case $v(M \cup A \cup B)$ is equal to the sum of v(M), v(A), and v(B). In general, the value of the complexity metric of a number of connected flowgraphs is the same as the sum of their individual complexity metric values (McCa76).

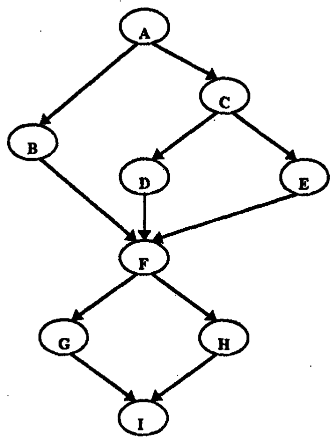
McCabe's metric is based upon a calculation of the number of linearly independent paths through the flowgraph in question. The complexity value 'v' is equivalent to that number. We can see this in the example above. For instance, there are clearly only three paths from the start node to the exit node in subroutine A, and the value of v(A) is equal to three. Suppose that we examine the flowgraph from the ATM code in **Figure 2-4**. This flowgraph has a McCabe metric value of four, as shown by the calculation

$$v(ATM) = 11 - 9 + 2 = 4 \ . \tag{2.9}$$

Inspection of the corresponding flowgraph reveals six different paths leading from the start node A to the exit node I. Recall that the McCabe metric gives the number of independent paths from start to exit. In the case of the ATM code, two of the six paths are linearly dependent upon the other four. Although any four of the six paths can be selected as the independent ones, it is useful to have a standard procedure.

A procedure for selecting the independent paths is documented by Poorman (Poor94). First a basepath must be selected. In order to facilitate the ease of selecting subsequent paths, it is convenient to select the path the with most nodes as the basepath. However, any path can serve this purpose. In the ATM example, the path (A+C+D+E+G+I) is selected as the basepath. The next step is to locate the first node along the basepath which contains a conditional split, and to follow the alternate path, rejoining the basepath as soon as possible. This path is said to be independent. If the first alternate path contains a conditional split before rejoining the basepath, this second path is followed rejoining the first alternate or the basepath as quickly as possible. This process is repeated until all of the branches of the first alternate path have been exercised, and have rejoined the base path. In our example, by following the alternate route from the conditional split at A, the next independent path would (A+B+F+G+I).

After the set of alternate paths has been exhausted, the basepath is followed to the second conditional split where the process is repeated. The second conditional split is exhausted of alternate paths and the process continues until there are no longer any new paths to follow. The resulting set of paths should be equal in number to the McCabe metric. The set of paths is also linearly independent. Thus, any other path through the flowgraph can be equated to some linear combination of the independent paths. Let us continue the process of finding the independent paths in the ATM example. At node C the alternate path (A+C+E+F+G+I) is found and at node F, path (A+C+D+F+H+I) is found. Thus the complete set of independent paths is presented in Table



The lineraly independent paths of the flowgraph illustrated in **Figure 2.6** are presented in **Table 2-1**. The flowgraph is printed to the left for the readers convinience.

Path Number	Flow Path
1	A+C+D+F+G+I
2	A+B+F+G+I
3	A+C+E+F+G+I
4	A+C+D+F+H+I

Table 2-1 The Linearly Independent Paths of the ATM Code

2-1.

Note that there are two other paths through the flowgraph which are not included in the list above. These paths, path 5 (A+B+F+H+I) and path 6 (A+C+E+F+H+I) are dependent paths. They can be represented as a linear combination of the other four paths. Path 5 = Path 2 + Path 4 - Path 1:

Path 5 = (A+B+F+H+I) = (A+B+F+G+I) + (A+C+D+F+H+I) - (A+C+D+F+G+I) .

Likewise, path 6 can be represented as a linear combination of paths one through four.

Path 6 = (A+C+E+F+H+I) = (A+C+E+F+G+I) - (A+C+D+F+G+I) + (A+C+D+F+H+I) .

McCabe demonstrated that the value of 'v' closely correlates to the number of errors present in a piece of code. As the value of the complexity metric v increases, the number of errors in the code increases as well. McCabe also noted that, in general, longer programs have low complexity and smaller ones have high complexity levels. This, he felt made program length modularization techniques highly inadequate. Instead, McCabe suggested that every module that had a metric value higher than ten should be further modularized. This he concluded would limit programming errors, and would result in code which is more easily tested.

The basics of the McCabe metric should now be understood by the reader. For a more in depth study of the topic see McCabe's paper "A Complexity Measure" (McCa76). The McCabe metric gives a method of measuring the structural complexity of a piece of code in a standard way. This allows one to set criteria for code modularization. The McCabe metric also gives the user a method for locating the independent paths through a flowgraph. Work has been done which utilizes the McCabe metric as the basis of a structural testing strategy. This testing method is discussed in **Section 2.3.2**. Used as either a testing aid or a gauge for program modularization, the McCabe metric is a useful tool for a software engineer.

2.2.2 Formal Development Methods

The use of formal development methods has recently become very prevalent in software development. Formal development methods are replacing the ad hoc methods previously used in software development. With the use of formal development methods, strong feedback is incorporated between the different elements of the design process. The use of formal methods improves the development process in two ways, according to Ouyang (Ouya95):

- A complete and unambiguous specification is developed in a form which facilitates mathematical validation.
- Verification of the software during the production process is much more effective.

The formal development method provides a mathematically based specification which describes the concept of the program. This mathematical basis provides an excellent foundation for verification and validation of the code. The use of a formal method has been shown greatly to improve the software development process. The formal method requires an increase in time spent on the early stages of a project, as the specifications are the main focus of the formal method. This extra time is later recovered over in the lifetime of the software. When using informal methods, much of the time spent developing software is devoted during the final stages of validation and verification. Because of this, much of the software must be reengineered. The use of a formal method greatly reduces the number of errors which are passed to the final testable code.

Ouyang's work includes a deep description of the many formal development methods currently in use. By comparing the various development methods, Ouyang concluded that the DBTF (Development Before The Fact) formal method was the most effective among those available currently. DBTF is marketed by Hamilton Technology Inc. of Cambridge, Ma. (Hami92a) The application of the DBTF method is aided by a CASE Tool Suite known as the

OO1 System. The OO1 System is explained in this report. For a closer examination of the subject of the formal development methods, one should review Ouyang's work.

2.3 Software Code Testing Methods

The main emphasis of the literature search of this report is upon code testing methods. The goal is to find the most efficient, and complete method possible. In order to choose this method, the literature is extensively reviewed, and each method is rated for completeness and general usefulness. In the end, the most complete methods are incorporated into a formal testing method. This formal method is then tested on several software codes.

A search of the literature of code testing reveals many different testing strategies. Some of these methods are specific to certain program types or programming languages and are ignored due to their lack of generality. The goal of the study reported here is to find a method which is useful in general, and not in specific cases only. Many of the methods found, fit this criteria. Although the methods are quite diverse, they can be broken down into two categories: *black box testing* and *white box testing*. Black box methods are based purely upon the specification document of the program in question. The code is treated as a closed box and is never examined. In white box testing, the code is examined, and tests are formulated based on the some aspect of the code itself. Black box testing methods are examined first.

2.3.1 Black Box methods

Black box testing methods, also called *functional testing* methods, derive test cases from the specifications and not the code itself. The tester is not concerned with the mechanism which generates an output, only that the output is correct for the given input set (Myers79.) Simple programs are often tested with black box methods. For example, a simple program's specification document might indicate that two numbers (A and B) be taken from user input and their sum (A+B) be displayed on the screen. To test this simple piece of code functionally, an input set is chosen which represents the typical use of the code. The tester may wish to input several combinations of A and B. For instance the cases where A is equal to B, A is greater than B ,and A is less than B could constitute a test set. The tester may wish to use input data values which are more likely to cause errors. For instance letters of the alphabet could be entered instead of numbers. If all of these cases execute properly, the code is found to be satisfactorily tested functionally. However, if one of the functional test cases yields an incorrect result, the code must be examined for the error. Due to the nature of the strategy, black box testing is also known as *input / output* testing.

Recall that the focus of this search is to locate a complete, yet practical testing method. The only complete black box method is known as *exhaustive input testing*. For the example above, every possible combination of the two numbers would be tested, resulting in an infinitely large test set. Exhaustive input testing is very impractical. If any one of the input variables is a rational number, an infinite amount of test cases must be considered. Despite the apparent impracticality of complete black box testing, these methods should be mentioned. They are examined here as a possible compliment to another more useful testing method. It may be possible to use them in conjuncture with other testing methods to achieve a more complete result.

Myers identifies a method for decreasing the number of tests needed in the input / output testing process. This method is called *equivalence partitioning*, and deals directly with the specification document to decrease the number of test cases needed. Because of the unfeasibility of exhaustive input testing, one is limited to a small subset of the allowable input values. With the equivalence partitioning method, this small subset is put to the greatest use possible. There are two considerations when developing test cases. The first is that the test case reduces the total number of tests needed by more than one. The second is that the test covers a large range of other test cases; i.e. it should test for more than one error type simultaneously (Myers79). Test cases are developed from each condition given by the specification document. Myers gives several rules for choosing test cases from the specifications. The process is a very subjective one with only a few loose guidlines. The testing of erroneous input values is stressed. Equivalence partitioning testing is a heuristic testing method, and is far from complete. For a more complete reference, one should review Myers.

Black box testing methods can be very useful, as they simulate the actual usage of the program. The program's targeted user will not examine the internal workings of the code during each usage. Black box methods will always be useful in testing small pieces of code designed for a simple and specific tasks. Exhaustive input testing is the only known complete black box testing method, and is very impractical. Functional testing methods may not offer a viable complete testing alternative, but could be used in conjunction with other methods of testing to achieve a useful result, and thus they are mentioned here.

Black box testing methods represent just one of the many testing schemes in the literature. All of the alternatives must be considered before a decision can be made as to which one is the most effective. While, it is possible that black box methods can be used in parallel with other methods to achieve an effective yet incomplete result, the heuristic approach which is so fundamental to black box methods makes them a poor candidate for use in the work reported here.

2.3.2 White Box Methods

White box testing methods generate test sets by examining the inner workings of the program in question. White box testing is also known as *structural testing* due to the basis of the method upon the code structure. Many types of structural testing methods exist. This survey reviews these numerous methods in order to decide which one is the most satisfactory for the job of complete testing. Since we can not be certain of finding a complete testing method, many incomplete testing methods are examined here. These incomplete methods may be used in conjuncture with other methods or as the basis of a new method to improve software reliability.

2.3.2.1 Basic Structural Testing Methods

To understand the concept of structural testing, three testing methods must become familiar. These techniques are *statement*, *branch*, and *path* testing methods, and they form the base of understanding most other forms of structural testing methods. These testing techniques are said to result in a certain *coverage* of the code.

- *Statement coverage* (known as C_1) is a testing method which assures that every statement of the code has been tested. In other words, the test set selected results in the execution of every statement in the code. (Ntaf88)
- *Branch coverage* (known as C_2) is achieved by testing every transfer of control from one program statement to the next. To attain complete branch coverage, every direction at every decision node must be executed at least once by a test case (Howd81). Statement coverage is included in C_2.
- *Path coverage* (known as C_∞) is attained by executing every possible control path through the code. Path coverage includes both C_1 and C_2 (Ntaf88).

Statement coverage is the least rigorous structural testing method. It is often possible to achieve statement coverage with relatively few test cases, even for a large piece of code. At the end of the test campaign, the tester knows that all of the statements in the code are executable, and that the program has yielded some correct results. However, the program tester can not concluded that the code is error free. Complete statement coverage is a far too limited a testing criterion to be used to indicate successfully complete testing.

Branch and path testing are very fundamental concepts to the study of structural testing metrics. Paths have been introduced previously in the description of control flowgraphs **(Section 2.2.1.3)**. Recall that a complete path is a sequence of nodes starting with the entrance node and finishing with the exit node of a program. Path testing involves testing every possible path through the flowgraph. This method can result in many tests. In the case of the ATM example code, recall that there are six paths through the code. Recall **Figure 2-6**, the flowgraph of the ATM code. To achieve path coverage, the six paths are those summarized in **Table 2-2**.

Path Number	Flow Path
1	A+C+D+F+G+I
2	A+B+F+G+I
3	A+C+E+F+G+I
4	A+C+D+F+H+I
5	A+C+D+F+H+I
6	A+B+F+H+I

A test set resulting in Path coverage of the ATM example.

Table 2-2 Path Coverage of the ATM code

. However, only four tests would be necessary to achieve complete branch coverage on that same code.

Path Number	Flow Path
1	A+C+D+F+G+I
2	A+B+F+G+I
3	A+C+E+F+G+I
4	A+C+D+F+H+I

A test set resulting in branch coverage of the ATM example code.

Table 2-3 Branch Coverage of the ATM Code

Path coverage is the more rigorous method. In fact, path coverage is the only complete testing method. A piece of code which is path-tested is, by definition, completely tested (Myers79).

The path coverage method may be rigorous, but it also has many practical drawbacks. Path coverage is often nearly impossible to attain in practice due to the enormous number of tests required to achieve it. This problem can be illustrated by a short program which contains twenty five consecutive if..then..else statements. To achieve path coverage one would require 2^{25} or 33.5 million test cases. Branch coverage, however, could be achieved with as little as two tests, one for all of the true branches and one for all of the false branches. It may not be possible to formulate such perfect test cases in practice, as some paths may be *infeasible*.

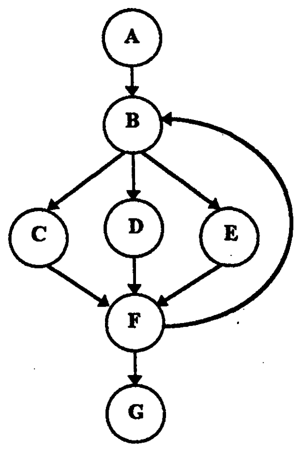
An infeasible test path can result from either branch or path coverage analysis. The reader may have noticed already that of the six paths in **Table 2-2**, only three of them are executable. Those paths are shown in **Table 2-4**.

Path Number	Flow Path
1	A+C+D+F+G+I
2	A+B+F+H+I
3	A+C+E+F+G+I

Table 2-4 Feasable test Paths of ATM Code

For every test case, a set of input variables must be determined which will cause that flowpath to be followed. In the ATM code example, the input variables are **"b"** and **"w"**. An analysis of the code reveals that the edge of node **A** divides the input variables into two distinct regions, $(w > b)$ and $(w \leq b)$. The edge of node **C** further divides the input variables into the three regions $(w > b)$, $(w < b)$, and $(w = b)$. One can also notice that if either node **D** or node **E** is reached, then the exit sequence will be (F+G+I). If node **B** is reached, then the exit sequence will be (F+H+I). Thus, three of the possible paths through the code are infeasible, meaning that no input variable set can be identified which will cause their execution. The possibility of infeasible paths adds to the time consuming nature of path testing. Infeasible paths can also be a nuisance when branch testing.

Looping path structures add great complexity to the process of software testing. Looping structures are the most difficult to test completely. Any type of recursion can result in an exponential growth in the number of paths in a piece of code. For example, consider **Figure 2-8** depicting a section of code containing a looping structure which executes "n" times through. There are three possible paths for each pass, or *iteration* through the loop.



Example Program to illustrate the relationship between looping structures and structural testing techniques.

Figure 2-8 Example Looping Program Flowgraph

For one iteration (n=1) there are three paths, a manageable number. However, the number of test paths increases very quickly as the value of n increases, as 3^n. Ten iterations result in 59049 paths, and 100 iterations lead to $5.15*10^{47}$ paths. Path testing becomes very infeasible as the number of iterations increases. In fact, for even 20 iterations a tester would have to execute one test every second for 100 years to complete the testing process. Since software is often outdated in a matter of years, it is necessary to find a more accommodating testing process.

"Branch testing . . . is usually considered to be a minimal testing requirement (Ntaf88)." This statement implies that errors can and do pass through branch coverage. Howden cited three empirical studies which showed that many errors are undetected by branch coverage (Howd81). A compromise must be found between the minimal approach of the branch coverage method and the infeasible, but complete path coverage method. The goal of this report is to locate a complete, and accommodating testing method. Path coverage is a complete testing method, but is not accommodating in most cases. Let us examine the other types of structural methods in the literature in hopes of finding a compromise.

2.3.2.2 Data-Flow Methods

Data-Flow testing methods are common in the literature and represent a possible compromise to unfeasibility of complete path testing. There are many types of data-flow methods. Only the most useful methods are summarized in this report. To understand data-flow testing methods, one must be familiar with several concepts which are explained below.

Data-flow testing works with the flowgraph to support a search for data-flow anomalies. Data-flow testing can lead to a selection of tests which represent a compromise between the rigor of path testing and the minimal approach of branch testing (Biez90). An extensive literature exists concerning the subject of data-flow testing. Much of this work was lead by Rapps and Weyuker (Rapps85). Their premise behind the idea of data-flow testing is a simple one. Their statement, as quoted by Biezure (Biez90), that ". . . one should not feel confident about a program without having seen the effect of using the value produced by each and every computation," has truth to it.

Data-flow testing has its origins in the black box testing technique of exhaustive input testing. In theory, every possible input value could be tested in order to satisfy the complete test criteria, but as we have seen, this is impossible for all but the simplest of codes. Data-flow testing is an intelligent method for selecting a small subset of the input domain for testing purposes (ie. for equivalence partition testing). One hopes that the selected subset reveals all of the errors in the program. In practice, such a subset is difficult, if not impossible to locate (Rapps85). However, systematically locating as complete a test set as possible is the goal.

For every testing method other than complete path testing, one must assume that the errors located in the code, also known as bugs, can be detected by the testing method of choice. In data-flow testing, one must assume that the program bugs are a result of the incorrect use of data objects. For instance, if the program's control structure is incorrect, data flow testing assumes that this error is refected in the data objects. This is a key assumption.

Comprehending the terminology of data-flow testing is very important to understanding the literature. Data-flow testing examines the process through which input data is transformed and processed in order to achieve the end result of the program. A *data object* is any type or variable which can be used by the program to alter either other program variables or the program's control flow. Data objects can exist and function in a number of ways as defined by the following symbols (Biez90).

- *d* or *def*- defined or initialized.
- *k* - killed or destroyed.
- *u* - used in some manner.
    - *c* - used in a calculation.
    - *p* - used in a predicate. A variable used as a predicate controls the execution path of the program. (ie. X is a predicate in the statement "IF X=true THEN Goto A ELSE Goto B")

As data objects flow through a program, their states and uses can be traced using the symbols above. For instance, a data object could be defined, used in a calculation, and then destroyed. Such a combination of events would be denoted as *dck*. Some combinations of symbols represent normal data-flow situations, while others represent bugs. For instance the combination *du* is allowed and frequently encountered. The data object is first defined, and then used. However, if the combination *ku* ever were to occur in a program, an error would be indicated, since *ku* represents the destruction of a data object followed by its use. This sequence is logically impossible. With these symbols in mind, data-flow testing techniques can be thoroughly understood.

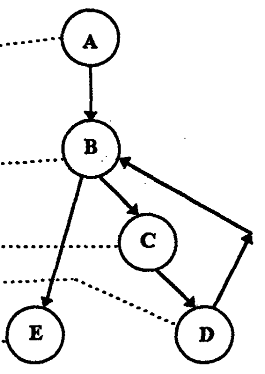
Rapps and Weyuker introduce a family of path selection criteria, based upon data flow analysis (Rapps85). In order to understand the notation used to describe a family of paths, one must first grasp the concept of a *def-clear path*. A def-clear path, which stands for definition clear path, is a path which does not contain a definition of a particular variable (call it x) in any of its interior statements. The following notation is given in further explanation of the concept of def-clear paths. A path $(i, n_1, \ldots, n_m, j)$ where $m \geq 0$, is def-clear with regard to x from node i to node j, if there are no definition uses of x in nodes $n_1, \ldots, n_m$. A path is a *simple* path if at most one of its nodes is visited twice (Beiz90). A simple path occurs when a looping structure is involved. For example the path (A+B+D+F+E+B) in **Figure 2.8** is a simple path.

Many types of test path selection criteria are introduced by Rapps and Weyuker, however, the most complete set of selection criteria corresponds to the *all du-paths* method. A du-path $(n_1, \ldots, n_j, n_k)$ is defined, with respect to a variable x, as meeting one of the following criteria.

- n_k is a c-use of x and path $(n_1, \ldots, n_j, n_k)$ is a simple and def-clear path with respect to x.
- (n_j, n_k) contains a p-use of x, and the path $(n_1, \ldots, n_j)$ is both def-clear with respect to x and loop free.

As well, the node n_1 must contain a definition use of the variable x. With the criteria in mind, one can define the test strategy governed by the all du-paths method. Simply put, the all du-paths method includes all paths which contain a du-path for every variable in the code. This is best illustrated by an example. The following example program is written in the language illustrated in **Figure 2-4**.

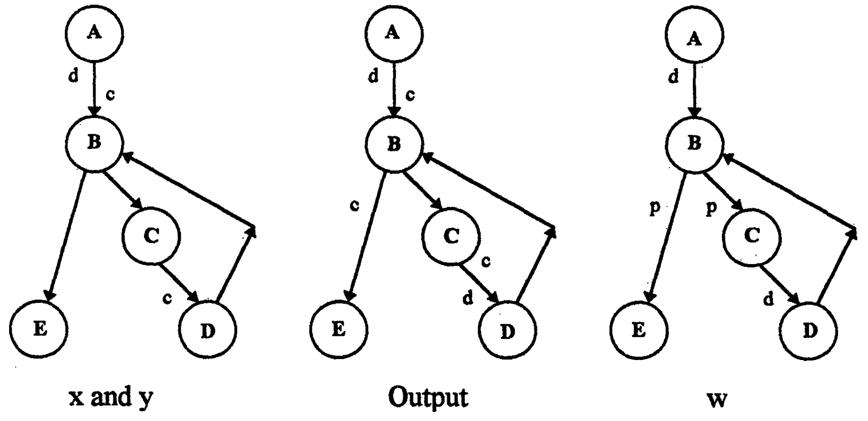
```
Begin
output <= 0
Print ("Enter an integer between 1 and 10")
read (x)
Print ("Enter an integer between 1 and 10")
read (y)
w <= x - y
[a]  if w≥1 then
        output <= f(output, x, y)
        w <= w - 1
        goto [a]
    else
        Print output
End
```



An Example Looping Program for the illustration of the all du-paths testing procedure.

Figure 2-9 Looping Example Program for Illustrating Data Flow Testing

The program's flowgraph is annotated to show the uses of each of the variables in the code on certain links. A separate flowgraph is constructed for each variables in the code as shown in **Figure 2-10.**



A flowgraph of the example program in Figure 2-9, annotated with data flow actions for the four variables used in the program: x, y, Output, and w.

Figure 2-10 A Flowgraph of the Looping Example with Data Flow Annotation

In order to demonstrate the all du-paths concept, the example program in **Figure 2-9** is described in flowgraph form in **Figure 2-10.** **Figure 2-10** indicates the data flow uses for each of the variables in the simple program. By examining the flowgraphs above, one can create a list of du-paths for each of the variables in the example program. Variables x and y are used in the exact same way, and thus share the same flow graph. The du-paths for x and y are (A+B) and (A+B+C+D). The du-paths of output are (A+B), (A+B+E), (A+B+C+D) and, (C+D+B+E). The du-paths for w are (A+B+E), (A+B+C) and, (C+D+B+E). It is important to note that a single test case can satisfy multiple du-paths. In this simple case, all of the du-paths can be tested with just two tests. The first test will be (A+B+E) and the second test (A+B+C+D+B+E). All nine of the individual du-path segments are included in these two tests. This simple test case demonstrates an important concept of the all du-paths testing scheme. Recall that a du-path must either be loop-free or simple. Thus, looping structures need only to be iterated once through. The possibility of an endless test path is eliminated. The test set generated by the all du-paths method is always finite, an advantage over the complete path testing method.

The literature mentions many other types of data-flow testing methods (Rapps85). These other methods are based on other types of path selection criteria. For instance, one of the testing strategies includes all of the predicate uses of every variable, and is known as *all p-uses*. Many other variations of data-flow testing exist. However, the other methods are less complete than the all du-paths method, and are not summerized in this report. For an in depth look at data flow methods see Rapps, Weyuker, and Frankl (Rapps85, Weyu93, Frank88, Frank93).

The all du-paths method is a rigorous one, which represents a compromise between complete path testing and branch testing. Several studies have been completed which compare the effectiveness of the all du-paths testing method with other testing methods. These studies indicate that while the all du-paths method is one of the most time consuming data-flow analysis methods, it yields the best experimental results (Weyu93) measured in terms of percentage of errors located for each test program.

The use of data-flow testing can not lead to a complete testing result. However, the most effective data-flow testing method, all du-path testing, is a viable compromise to complete path testing. Complete path testing is a complete and rigorous method, which in the presence of loops can take an unreasonable amount of time to achieve. All du-path testing circumvents the problem with loops by limiting them to one iteration. However, an error could be introduced which only surfaces after n loop iterations, and thus would not be found by the all du-paths method. All du-path testing is a very complex and time consuming technique which may not be feasible for large programs.

2.3.2.3 McCabe's Metric Based Methods

The works of Poorman (Poor94) and Arno (Arno94), explain the use of the McCabe metric as the basis for a testing method. McCabe explains that his metric is an excellent measure of the number of tests needed to achieve branch coverage. However, he also indicates that the metric should only be used as what it is, a guideline. If one has a program G with a McCabe's metric value of v, the number of test cases t should be greater than or equal to v. If t is less than v, either the flowgraph is in error, or more test cases are needed to achieve branch coverage. Poorman concluded that the McCabe metric "... is essentially a measure of the number of tests required to completely test the structure of the code. (Poor94)" This is false, even in simple cases. Consider the following example specification document and resulting program.

The following program is utilized to accomplish this task.

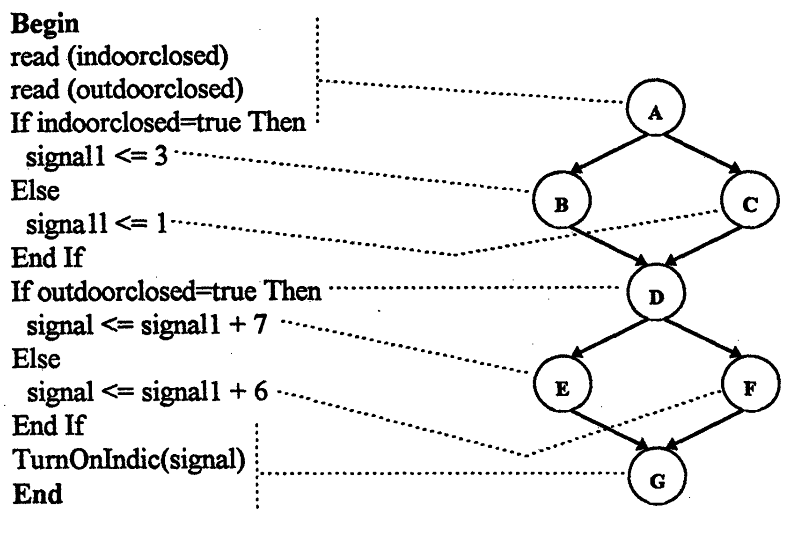
- The program will illuminate certain indicator lights by calling the function TurnOnIndic (indicator number), depending upon the status of a submarine's torpedo tube. The tube has two doors, an outer and an inner door. The table below demonstrates which indicators are to be lit during each of the four possible door configurations:

Inner Door	Outer Door	Indication Light Number
Closed	Closed	10
Closed	Open	9
Open	Closed	8
Open	Open	6

- The boolean variables innerdoorclosed and outerdoorclosed will be supplied. These variables are true if the respective door is closed and otherwise false.
- Indicator Light Six is also attached to a warning siren, as opening both the inner and outer doors could have grave consequences.

The specification document for McCabe metric based testing example program.

Figure 2-11 Submarine Door Example Program Specification Document



```
Begin
read (indoorclosed)
read (outdoorclosed)
If indoorclosed=true Then
  signal1 <= 3
Else
  signal1 <= 1
End If
If outdoorclosed=true Then
  signal <= signal1 + 7
Else
  signal <= signal1 + 6
End If
TurnOnIndic(signal)
End
```

The resulting program, from the specifications document in Figure 2-11 with its flowgraph.

Figure 2-12 Submarine Door Example Program Listing and Flowgraph

The McCabe metric of the program is (v=8-7+2=3) three. Thus, under the assumption that the McCabe metric equals to the number of test cases necessary for complete testing, only three test cases are needed. The chosen test paths and the test results are listed below.

Path	Inner Door Status	Outer Door Status	Resulting Indicator Number
A+B+D+E+G	Closed	Closed	10
A+C+D+E+G	Open	Closed	8
A+B+D+F+G	Closed	Open	9

Table 2-5 Test Cases and Results for Sub Door Example

The tests for the three chosen paths were successful. Therefore, if the McCabe metric represents the number of tests required to completely test the code, the program is assumed to operate correctly.

The reader may have noticed that the program is incorrect, and does not meet the specification requirements. This is not discovered, because the error lies in the untested path

(A+C+D+F+G), the case where both doors are open. Failure to locate such a grievous error demonstrates the shortcomings of Poorman's method.

2.3.2.4 Variations on Path Testing

Thus far, two complete testing methods have been found, exhaustive input testing, and complete path testing. Of these two, complete path testing is the most feasible. In the case of simple non-looping programs, complete path testing is attainable. Exhaustive input testing is rarely achievable in any situation. It is apparent that complete software testing of reasonable sized programs is only possible in theory. Path testing seems to be the best theoretical method for attaining a complete test. In hopes of decreasing the sizable number of test cases in the most beneficial way, an examination of the subtle variations of path testing follows. The goal is to locate a testing coverage that can realistically be achieved, while retaining the path testing approach.

Structured path testing and *boundary-interior path testing* seem to have the most promise as a compromise to complete path testing. Both methods were pioneered by Howden (Howd75). The methods only differ from path testing in the number of times that looping paths are iterated. Both strategies limit the number of iterations. The more rigorous of the two methods is Structured path testing, and it is discussed first.

Structured path testing is equivalent to complete path testing, except that the number of loop iterations is limited to some number. To be more specific, all test paths P, are followed which do not contain more than k repetitions of any subpath *sp* (Ntaf88). Thus, every loop is limited to k iterations. As k approaches infinity, structured path testing becomes equivalent to complete path testing.

Boundary-interior testing is closely related to structured path testing. Boundary-interior testing involve separating the possible testing paths into two groups. One group, called the boundary group, consists of all paths which do not enter loops. The other, known as the interior group, consists of all of the paths which enter the program's recursive structures. The interior group contains many paths which differ only in the number of times that a loop is iterated. From the interior group, only the paths which contain one iteration of the loop are added to the test set. Every path in the boundary group is included in the test set. Thus the flowgraph in **Figure 2-13**, requires three tests using the boundary-interior method. The first path (A+B+G the boundary path) does not enter the loop. The other two paths (A+B+C+D+F+B+G and A+B+C+E+F+B+G the interior paths) follow the two alternate paths inside the loop. The loop is not iterated multiple times. Boundary-interior testing, is equivalent to structured path testing when k is equal to one.

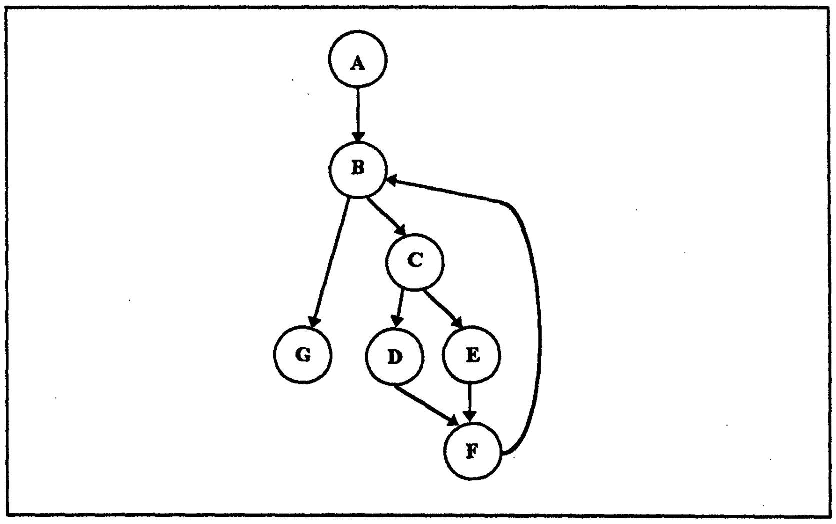


Figure 2-13 Example Flowgraph to Demonstrate Altered Path Testing Methods

2.3.2.5 Mutation Based Testing Techniques

A technique called mutation testing is also mentioned in the literature. Mutation based testing is structurally based, but is unlike any technique which has thus far been discussed.

Mutation testing requires that a class of mutation transformations be applied to the program in question, that will introduce errors of a certain kind into the code. The typical transformation might involve altering variables in the control structure, altering calculations, and swapping variables. A test set is said to be complete if it can distinguish between all of the mutated programs and the original program (Howd81).

Mutation testing is very computationally intense and time consuming. No sure way of determining the proper mutation transformations exists. It is estimated that there are on the order of n^2 mutations of an n line program. This necessitates a large number of tests for even a small program. In fact, if there are t tests in the test set, the number of test cases necessary is somewhere between n^2 and $t*n^2$ (Howd81). The use of mutation testing in the literature is limited. In the work reported here, mutation testing is not used. Mutation testing is very time consuming as does not attain a complete test of the software.

2.3.3 An Overview of Testing Technique

To this point, the emphasis of this report has been on the particular method of testing. There are some general issues which should be discussed with regard to testing strategy. Once one has decided upon a specific code testing method, for instance boundary-interior testing, there are still several issues to be resolved. Perhaps the biggest of these issues is the question of how one should attack the testing process. Modern software is often very large and modularized. Given a very large software system, one may not want to attempt to run test cases for the entire code. It may be more efficient to test each module individually. Such an approach is known as incremental testing. An approach which tackles the entire code simultaneously is known an nonincremental testing.

Each method has its advantages and disadvantages. Nonincremental testing is very difficult for large programs. McCabe recommended that if a module had a metric value of greater than ten, it should be further broken up. His motivation is as follows. It is very difficult for a human to analyze and develop test cases for an entire piece of complex software. It is much easier to divide and conquer. Incremental testing allow one to do just that. However, there is the possibility of interface errors occurring between the modules. The incremental method does not test for interface errors as rigorously as the nonincremental method does.

One must also be aware of a mechanism for determining the correct answer for each test case. The result of a test case is worthless if one can not determine the correctness of the result. A data base which contains the correct answer for every test case is known as an *oracle*. The presence of an oracle is often assumed in the literature of software testing. In practice, this assumption can be a costly one. It is often very difficult to determine if every test case achieves the correct result. Often the tester will also act as the oracle. This practice is not recommended, as the tester will be biased from having made an in depth examination of the code. Determining the proper source and type of oracle to be used is a major problem in any software testing project.

2.4 Literature Search Conclusion

The goal of the literature search reported here was to locate a complete testing method, with a focus on software development methods which make this feasible. The search was broad in scope, with a focus on testing methods which are applicable in general.

Methods for developing reliable software have been examined. Both McCabe and Halstead concluded that a shorter program is less prone to error. McCabe has developed a metric for measuring the complexity of a piece of software code. Using this complexity measure as a guide, McCabe recommends breaking a program up into modules with a metric value of no greater than ten each. Experimentally, it has been shown that this procedure reduces the number of errors in each module. Formal development methods, such as DBTF (Development Before The Fact) have also been examined. These methods provide a mathematical basis for the program specification, and assure an aided verification and validation process.

Two complete testing methods were found. One method, a black box method, is known as exhaustive input testing. Implementation of exhaustive input testing results in an unreasonable amount of test cases. The other complete testing method, a structural one, is known as complete path testing. The literature regards complete path testing as being impossible for all but the simplest loop free programs. The presence of a loop can result in an infinite number of test paths, resulting in an infinite testing time. Because of the unfeasibility of these methods, other methods are examined. Other forms of black box testing are inadequate for the task, and do not measure up against the alternative structural methods. Of the structural methods, variations on path testing known as structured path testing and boundary-interior testing, appear to have the most merit. However, improvements can be made to either of these methods. Thus, it is evident that a practical method for completely testing all except very small simple programs does not exist.

3. Selection of a Testing Method

3.1 General

Due to the lack of a feasible method for completely testing safety related software, a new method of testing has been developed. This method is known as the *Feasible Structured Path Testing* method and is designed to incorporate many of the favorable aspects of the testing methods discussed in chapter two. The Feasible Structured Path Testing (FSPT) is a structurally based testing method resulting from a mixture of data flow and path testing methods. The method differs from those in **Section 2**, in several important ways. The FSPT systematically guides the tester through the process of formulating test cases. The method is complete in many cases, but is not overly time consuming. In the case of a looping program where complete path testing yields an unfeasibly large amount of test cases, the FSPT limits the number of tests to a manageable number. One important aspect of the FSPT is the removal of unfeasible test paths, which speeds the testing process. The results attained from using the FSPT in several examples programs are excellent. All errors within the example codes are located. Due to its structured nature the FSPT could be automated, to greatly increase the efficiency of the testing process.

3.2 Feasible Structured Path Testing

The feasible structured path testing method is based upon the concept of structured path testing. As stated in **Section 2**, structured path testing is equivalent to complete path testing aside from limiting the number of loop iterations to a value, k. Every test path is limited to k iterations of looping structures. The FSPT duplicates this strategy. If a looping structure exists in a section of code, that loop is only executed k times through at most. This strategy prevents the occurrence of an unreasonable and conceivably infinite number of test cases. This criterion also sacrifices completeness, but as we have seen, this sacrifice must be made for testing to be practical. Also, the FSPT prevents the testing of unfeasible test paths, by examining the program's data flow and control structure to determine which paths are achievable. The FSPT consists of five steps. An overview of the entire process is presented, followed by a detailed description of each step. This chapter concludes with two example uses of feasible structured path testing.

3.2.1 Method Overview

Feasible structured path testing consists of a five step formal process. The goal of the process is to take a program listing, and through analysis to, systematically produce test cases which achieve structured path testing coverage involving no unfeasible test cases. This process is as follows:

1. The first step is to draw the flowgraph of the program. This step is completed as demonstrated in **Section 2**. The flowgraph is then altered to include only those paths which include k iterations of any looping structures which may be present.
2. The second step of the process is to locate the data flow paths which lead from the input variables to the output variables. Each variable that exists in the operational sequence of the flow path from the input variables to the output variables is recorded in a list. These variables are known as *transitional* variables.
3. The third step of the process utilizes the list of transitional variables generated in the second step. The tester must reorganize the flowgraph to reflect every unique state of the transitional variables.

4. The fourth step in the process is to measure the McCabe metric value of the altered flowgraph to determine the complete set of linearly independent paths for the piece of code. The McCabe metric can be used to achieve a complete result only because the flowgraph has been altered. Poorman's method utilized the classic control flowgraph with no alterations and achieved only branch coverage.
5. The fifth step in the process is to trace each test path to determine which input values correspond to each path. The tester also systematically eliminates unfeasible paths. Unfeasible paths can not be reached by any combination of input variables. The input values for each feasible test path are recorded.

The tester then has all the information needed to proceed with the FSPT. Each of the steps listed above is examined in detail below.

3.2.2 Feasible Structured Path Testing: The Details

1.1 The first step of the FSPT examines the program being tested and constructs the program's flowgraph. This process is documented in **Section 2**. During the first step, the tester should become familiar with the control structure of the program being tested. In the case that looping structures are present in the code, one expands the flowgraph to include only k iterations of any looping structure. There are several implications of limiting the value of k. A complete test occurs when k reaches infinity or the maximum value of loop iterations possible. In many instances, the tester does not have the facilities to test

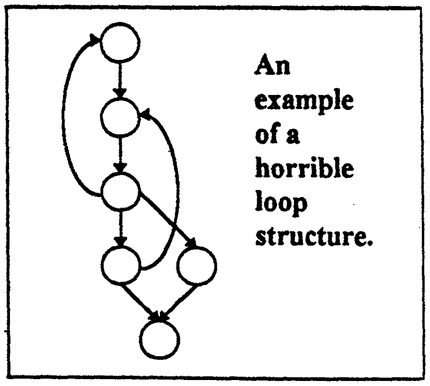


Figure 3-1 Horrible Loops

such a tremendous number of test cases. The tester should be familiar with the time needed for each test. Comparing this time with the total time alloted for testing, the tester can arrive at a practical estimate of k. k should be choosen as to limit the number of test cases, so the testing can be completed in a timely manner.

1.2 The first step assumes that no *horrible loops* exist in the code. Horrible loops are described by Beizer as "... code that jumps into and out of loops, intersecting loops, hidden loops, and cross-connected loops . . .," (Beiz90). The FSPT assumes that the code contains no horrible loops. **Figure 3-1** contains an example of a horrible loop structure. The flowgraph developed in the first step serves as a guide for steps two and three.

2.1 The second step involves an examination of the data flow of the program, and results in a list of variable transitions. These transitions relate the input variables to the output variables. The first step of the process is recognizing the input and output variables of the code. Any variable or data object that attains its value from outside the piece of code being examined is considered an input variable. Those variables which facilitate the ultimate use of the program are known as output variables. To recognize the output variables, one must have some understanding of the program's function. Output variables are produced in several ways. A function can be performed and its products presented on an output device (e.g. a monitor), data can be stored in a memory device (e.g. RAM) for later use, or in the case of testing a program module, variables can attain values for use in other modules. Once the tester recognizes both input and output variables of the code, the remainder of the second step can begin.

2.2 A list of every variable which directly affects the flow of data from the input to the output variables is generated. This list contains the transitional variables. For a variable to be included in the list of transitional variables, it must meet one of two requirements. The variable can directly attain its value as a function of an input variable, or of another transitional variable. Also, other variables are designated transitional variables by directly affecting the value of the output variables, or by affecting the values of other transitional variables which affect the output variables. Recall from the exemplary language, detailed in **Figure 2-4**, the assignment statement: y <=f(x_1, ... , x_n) where (y, x_1, ... , x_n) are variables. In order for a variable to be included in the transitional variable list, it must be created from an assignment statement involving either input variables or other transitional variables already assigned to the list. The relationship also works in reverse. If y is the output variable or another transitional variable, then x_1, ... , x_n are added to the list of transitional variables. When all flow branches have been examined including as many as k iterations of any looping structure, one should have a complete list of every transitional variable leading from the input to the output variables. When this list is complete, one can move on to the third step of the process.

Expanding a Flowgraph to Show k Iterations

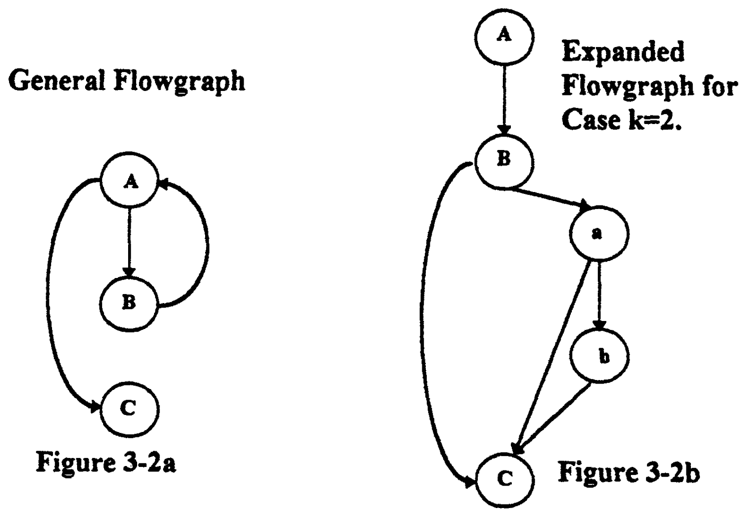


Figure 3-2a **Figure 3-2b**

The flowgraph in (Figure 3-2a) is exanded into two iterations of the loop represented in the path A+B+A in (Figure 3-2a). In the case above the value of k is two.

Figure 3-2 Fowgraph Expanded to k Iterations

3.1 The third step of the FSPT results in an altered form of the original flowgraph. From the altered flowgraph, one designs the necessary test cases to complete the method. In the first step of the FSPT, the flowgraph is altered to reflect only k iterations of each looping structure. See **Figure 3-2**.

3.2 Starting from the top node, one systematically examines the state of the transitional variables that exist in each node. If a node contains a transitional variable that has not reached the end of its development, the transitional variable affects other transitional variables or the output variables at some point further down the execution path. It is necessary to expand the flowgraph into a parallel structure reflecting each possible unique state of that transitional variable. The number of states produced depends upon the number of possible paths reaching the node in question. For instance, in **Figure 3-2**, if at node **C** a transitional variable x exists that affects the program's output variables, node **C** is expanded. If a unique value of the variable x arises from each of the three paths leading into node **C**, node **C** expands into three separate nodes and a new exit node is added. See **Figure 3-3** for an illustration.

Once the flowgraph has been altered, the fourth step can be completed.

4 The fourth step in the FSPT finds the value of the McCabe metric of the flowgraph altered in step three. This value is used to determine the number of test cases needed to completely test the code. As described in Section 2, a base test path is selected, and a set of linearly independent paths is chosen from the code. These paths represent the complete set of the FSPT test cases.

Expanding a Flowgraph to Show Unique Transitional Variable States.

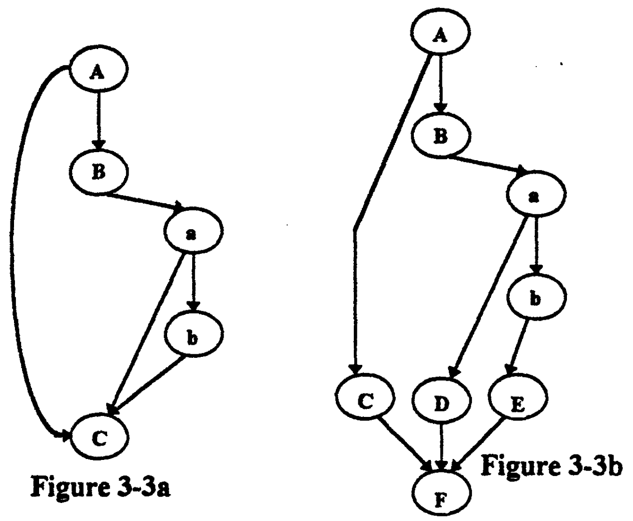


Figure 3-3a **Figure 3-3b**

The flowgraph on the left is expanded at node C to reflect the three unique states of the transitional variable x which leads to the program output at node F.

Figure 3-3 Flowgraph Expansion: Transitional Variables

5. The fifth step in the method determines the values of the input variables that cause each test path to execute. This is done by deducing those values from the test flowpath. In some cases, a broad range of input values lead down a certain path. For example, consider that a predicate at node a in **Figure 3-3** states the function (if (x>30) then A, else B), then the variable x could take all values ≤ 30 in the false path B and all values >30 in the true path A. In this case, the tester chooses a value at random from within the range and, records the exact values of the input variables that are used for each test case. The tester must also determine which test paths, if any, are unfeasible. Unfeasible paths usually appear in a flowgraph because reaching a certain node in the flowgraph predetermines the exit path segment. For instance, in **Figure 3-3b**, if at node A the

variables that control the predicate are always fixed to choose the finishing path **D+F**, then the finishing path **b+E+F** can be eliminated as a test case. This entire process is demonstrated later in this chapter. Proper record keeping is necessary to facilitate the location of any errors in the code.

3.2.3 Feasible Structured Path Testing: General Testing Strategy

The use of the FSPT is not limited to any type of program, and must therefore be adapted for use in many complex situations. A simple ten line program is tested quickly and efficiently by the FSPT. However, most programs are more lengthy with many code modules. The FSPT can also efficiently test larger programs. A general testing strategy applicable to all types and sizes of programs is given below.

Given a section of code to be tested using the FSPT, there are several important procedures to be followed. One must be very rigorous in recording data associated with the test cases. One should have a table indicating the exact values of the input variables associated with each test case. There may be a case where input variables are passed to a section of code, but not used in the tests. The values of these variables should still be recorded for each test case. In the presence of an error, complete record keeping is vital. The tester records the value of every output variable after each test case. These values can be compared with the oracle database developed by the tester. The oracle data base contains the correct value of the output variables for every test case. A comparison with the results from the oracle reveals any errors which are present in the code. Should an oracle not exist, which is often the case, the record of the test results can be used at a later time. The true goal of the FSPT is to demonstrate nearly every way the program could possibly be used. This demonstration record is compared with the program's specification document to locate any flaws in the code.

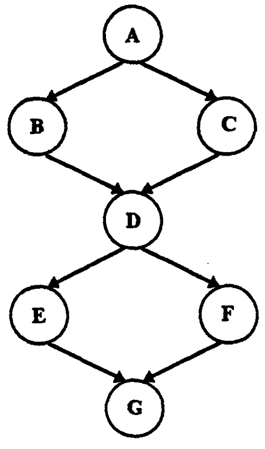
Feasible structured path testing should be utilized as an incremental testing method. As stated in **Section 2**, incremental testing consists of the testing of each modular section of the code separately. Doing this eases the burden upon the tester since smaller program is less cumbersome, and often less apt to contain a unreasonable number of test paths. In fact, it is recommended that the value of the McCabe metric of any module be no greater than ten. Experience indicates that testing becomes increasingly difficult as the McCabe metric grows, and ten is found to be a reasonable metric maximum. Given a program in modular form, the tester should examine the program and organize it into a hierarchy of levels. The highest level code contains all the modules in the first level. The first set of sub-modules could contain a second set of sub-modules. The second set could contain a third, etc. Given a modules on the first level, testing should be conducted as follows. The deepest sub-modules should be tested first. Once these are tested, the next set can be tested using the FSPT, and so on until every module has been tested. At this point, the high level code is tested, and the testing is complete.

Incremental testing methods can result in several problems. Often a specification document is not written in the modular form reflected in the structure of the program spawned by it. In such a case, it is difficult to compare the results of a modular test to the specifications, if the specifications do not indicate clearly what a particular module is to accomplish. The FSPT is effective in demonstrating the performance of a module, but relies upon the tester to determine if errors occur. This problem is found throughout the literature of software testing, and is solved by assuming the presence of an oracle.

3.3 FSPT: An Example

1. This first example of the FSPT involves the submarine door example program used in

```
Begin
read (indoorclosed)
read (outdoorclosed)
If indoorclosed=true Then
  signal1 <= 3
Else
  signal1 <= 1
End If
If outdoorclosed=true Then
  signal <= signal1 + 7
Else
  signal <= signal1 + 6
End If
TurnOnIndic(signal)
End
```



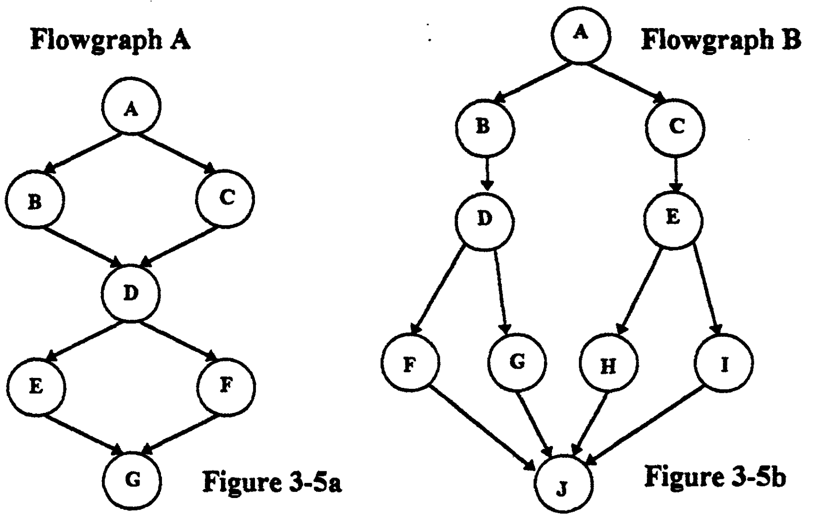
Submarine door example program and control flow graph for use in explaining Feasible Structured Path Testing.

Figure 3-4 Submarine Door Program and Flowgraph for the FSPT

Section 2.3.2.3. This simple program effectively illustrates the use of the FSPT. **Figure 3-4** is included for convenience. The diagram results from the first step of the FSPT in the five step process. The first step is to draw the flowgraph and expand it in the presence of looping structures. Since there are no loops in this case, the flowgraph in **Figure 3-4** results from step one of the process.

2. The second step of the process is listing of the transitional variables. This can be done quickly for this simple program. The input variable set includes the variables, indoorclosed and outdoorclosed. The output variable is signal. There are no transitional variables which result from the input variables. However, the output variable, signal, gives rise to the transitional variable, signal1. With signal1 identified as the transitional variable, the third step can occur.

3. The third step of the FSPT is the most complex. The flowgraph is altered to reflect unique states of the transitional variables. If one examines the flowgraph, one notices that at node **D** the variable signal1 is in a non-unique state. That node must be expanded to include the non unique states of the transitional variable, signal1. The value of signal1 can have a value of either 3 or 7, depending on which flowgraph branch is taken at node **A**. **Figure 3-5** illustrates this change.



Example of step three of the FSPT process. Node D on the left contains two states of the transitional variable signal1 and is expanded into nodes D and E to the right.

Figure 3-5 Step Three Expanded Flowgraph: Submarine Door Example

Step three is complete.

4. The fourth step of the process requires one to compute the value of the McCabe metric of the altered flowgraph. This results in a value of four (v=12-10+2=4). The next step is to choose the linearly independent paths through the flowgraph. The result of this step is given in **Table 3-1**. The

Path Number	Flow Path
1	A+B+D+G+J
2	A+C+E+H+J
3	A+C+E+I+J
4	A+B+D+F+J

The linearly independent paths of flowgraph Figure 3-5b.

Table 3-1 Lineraly Independent Paths of the Submarine Door Example

given paths represent the complete test set for the example program.

5. The final step is to determine the value of the input variables, indoorclosed and outdoorclosed. The values of these variables determine the flow path which is executed. In every two way predicate branch the left branch represents a true path and the right branch represents a false path. Thus, the repective values of indoorclosed and outdoorclosed are given in **Table 3-2** for each test path. The are no unfeasible paths in this example. The results of the test

Path Number	Flow Path	indoorclosed	outdoorclosed
1	A+B+D+G+J	True	False
2	A+C+E+H+J	False	True
3	A+C+E+I+J	Flase	False
4	A+B+D+F+J	True	True

The value of the input variables indoorclosed and outdoorclosed for each test path.

Table 3-2 Tested Cases: Submarine Door Example

show that an error exists in path number 3 when the test outcome is compared with the oracle.

3.4 FSPT: A Second Example

```
        Begin
        output <= 0
        Print ("Enter an integer between 1 and 10")
        read (x)
        Print ("Enter an integer between 1 and 10")
        read (y)
        w <= x - y
[a]     if w≥1 then
                output <= f(output, x, y)
                w <= w - 1
                goto [a]
        else
        Print output
        End
```

Example Program for the illustration of FSPT with looping structures.

Table 3-3 Looping Program for Example the FSPT

The example program of **Table 3-3** is used to illustrate the FSPT with looping structures. The following example, contains a looping structure, and may answer any remaining questions about implementing the FSPT. The example program of **Table 3-3** is also listed in **Figure 2-6.** This code however, has no specification document attached to it. It is useful, none the less. It is interesting to examine the testing process when loops are involved.

1. The first step in the FSPT process is to construct the flowgraph with a limit of k iterations of any looping structures present in the code. In this case, the value of k is selected to be two. In the case where k is equal to two, the flowgraph is illustrated in **Figure 3-6b**.

2. The second step of the process is to capture the transitional variables. One recognizes that the input variables are output, x and y. The transitional variables in this case are *w* and *output*. It is true that *output* is also the output variable, it is not considered to be the output variable until it has reached its final state. With the flowgraph and list of transitional variables, the tester is ready for the third step.

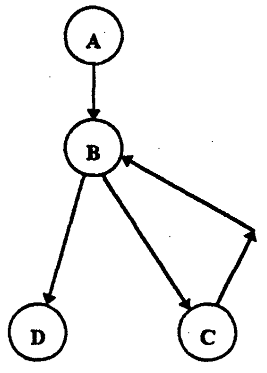


Figure 3-6a

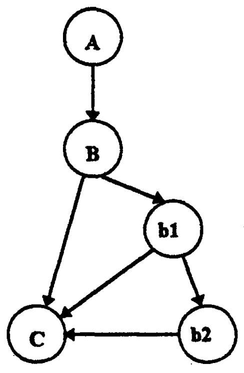


Figure 3-6b

An example of the FSPT step one when loops are involved. The flowgraph on the right represents the flowgraph on the left with a limit of two loop B+C+B iterations.

Figure 3-6 Third Step Expanded Flowgraph: Looping Example

3. In this case, the third step of the FSPT is an interesting one. In this step, the flowgraph is altered to reflect the unique states of the transitional variables. Also, the flowgraph is relieved of any unfeasible flow paths. The transitional variables are output and w, as mentioned above. The flowgraph in **Figure 3-6b** reflects every unique state of the input variables. Step three is quickly completed.

4. In step four, the McCabe metric of the structure is calculated, and the test paths are chosen. The value of the McCabe metric of the flowgraph is three (v=6-5+2=3). The linearly independent paths are listed in **Table 3-4**.

Path Number	Flow Path
1	A+B+C
2	A+B+b1+C
3	A+B+b1+b2+C

The linearly independent paths of Figure 3-6b.

Table 3-4 Linearly Independent Paths of the Looping Example

5. The final step of the process is to determine the values of the input variables x and y which correspond to each of the paths. Each of the paths contains a unique set of input variables that cause its execution. There are no unfeasible paths listed **Table 3-4**. The first path is executed when w is less than unity. Equivalently, path one is executed when w≤0 because integers are being used. w is equal to x-y, and thus path one is followed when y≥x. Path two is followed when y=x-1 and path three when y=x-2. Because the number of loop iterations was limited to two, these are the only cases which are tested. Those cases between y=x-3 and y=x-9 are not tested. The results of the tests are recorded, and compared with those of the oracle. Notice that it makes no difference what the function f(output, x, y) consists of. It is only important to recognize that the transitional variable output is directly affected by this function of the input variables x, y, and output.

3.5 FSPT: Conclusion

Feasible structured path testing is designed to attain complete testing coverage whenever possible, in a systematic way. The method combines the structured path testing method with data flow examination to achieve the desired result. The structured path testing method allows the tester to eliminate data flow paths containing more than k iterations of an looping structure. This greatly speeds the testing process of programs containing loops. Data flow analysis is used to eliminate any unfeasible paths from the testing scheme. Data flow analysis also streamlines the testing strategy by removing redundant test cases. The FSPT gives the tester a list of feasible and fairly complete test cases to use in testing the program at hand. The goal of these tests is to locate every error in the code. In most cases, this goal is achieved.

4. The OO1 Case Tool

4.1 General

Programs written in the OO1 case tool language are used to demonstrate feasible structured path testing methods. The OO1 tool is briefly discussed in **Section 2** of this report. The OO1 case tool is used to facilitate the Development Before the Fact (DBF) formal method. The reader must have a general understanding of the OO1 code structure to fully appreciate the examples presented in the next chapter of this report. The goal of **Section 4** is to give the reader that understanding.

The OO1 system is highly reliable. This reliability comes from the use of pre-defined reliable systems. Only reliable pre-tested building blocks are used as mechanisms to integrate systems. These building blocks can be classified into two catagories, *types* of objects and *functions* which operate upon the types. Types are essential to the OO1 language and can be arranged into a data structure known as a *Type Map* or TMap (pronounced T-map). Functions are also arranged into maps known as *Functional Maps* or FMaps (pronounced F-map). Understanding FMaps and TMaps is essential to understanding the OO1 language.

4.2 Type Maps (TMaps)

Type maps are constructed of reliable and predefined data objects known as primitive data types. These predefined data structures are built into TMaps. The relationship between types and types maps is illustrated in **Figure 4-1**.

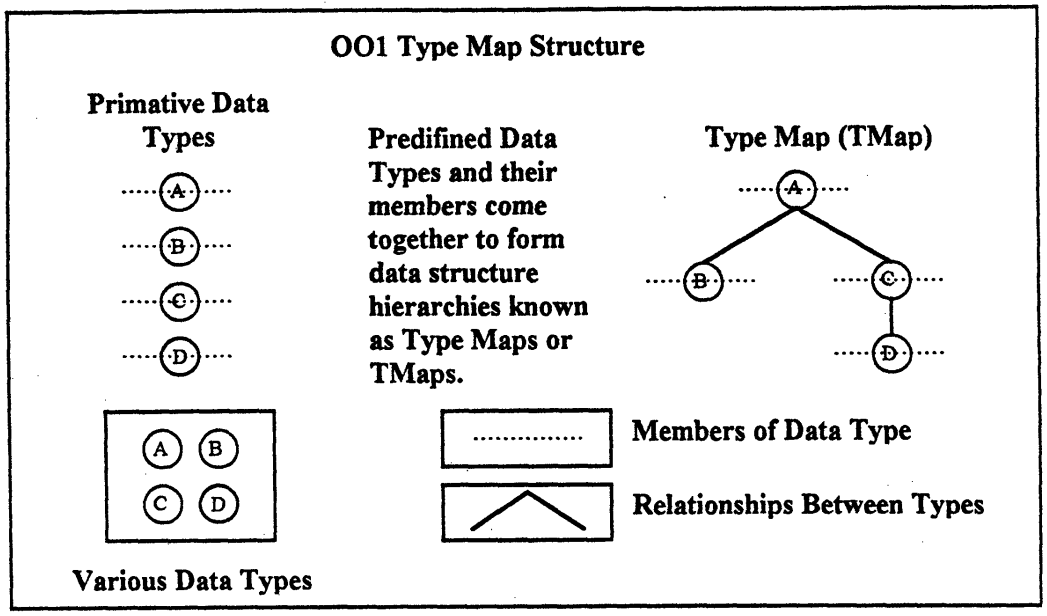

OO1 Type Map Structure
Primative Data Types
A
B
C
D
Predifined Data Types and their members come together to form data structure hierarchies known as Type Maps or TMaps.
Type Map (TMap)
A
B
C
D
Various Data Types
Members of Data Type
Relationships Between Types

Figure 4-1 Type Map (TMap) Explanation

Type maps can be used to create very complex data structures by specifying the relationship between primitive types, or other TMaps previously defined by the user. Primitive data types include strings, integers, Booleans, etc. Another data type exists, and is known as the *parameterized data type*. The parameterized data type provides a mechanism to define a TMap without a specific definition of the relationships between primitive data types. Examples of parameterized data types are listed below.

The data type *TupleOf* is defined as a collection of different data objects which is fixed in size. In **Figure 4-2** (Ouya95) one can see an illustration of the TupleOf data type. The TupleOf called RobotA describes a data structure that could be used by a program controlling a robot. RobotA, located at the top of **Figure 4-2**, contains six members. Five of the members are natural numbers (Nat), and the member called Ports is another TupleOf. Ports contains two members. Both of the members are defined in sub-maps. The period at the end of the member IOPort. and APortIds. indicate this fact. The variable IOPort., which describes the values given to the input /

output ports is an OSetOf type. The OSetOf type is explained next. The variable APortIds. is a OneOf type which describes the the type of IOPort being used. The OneOf type is also explained below.

The second parametrized type is known as a *OSetOf* (pronounced ordered set of). The OSetOf is a collection of variables of the same type, similar to an array. The data type IOPorts is and OSetOf the type IOPort. This means that IOPorts contains a list of every IOPort used in robot being represented in this example.

Each individual IOPort is of the parameterized type *OneOf*. This type indicates that only one of the values listed below the IOPort type can be allowed at any one time. The type APortIds is also a OneOf type. The identification of the IO port in use can only be one of the five strings listed below APortIds in **Figure 4-2**.

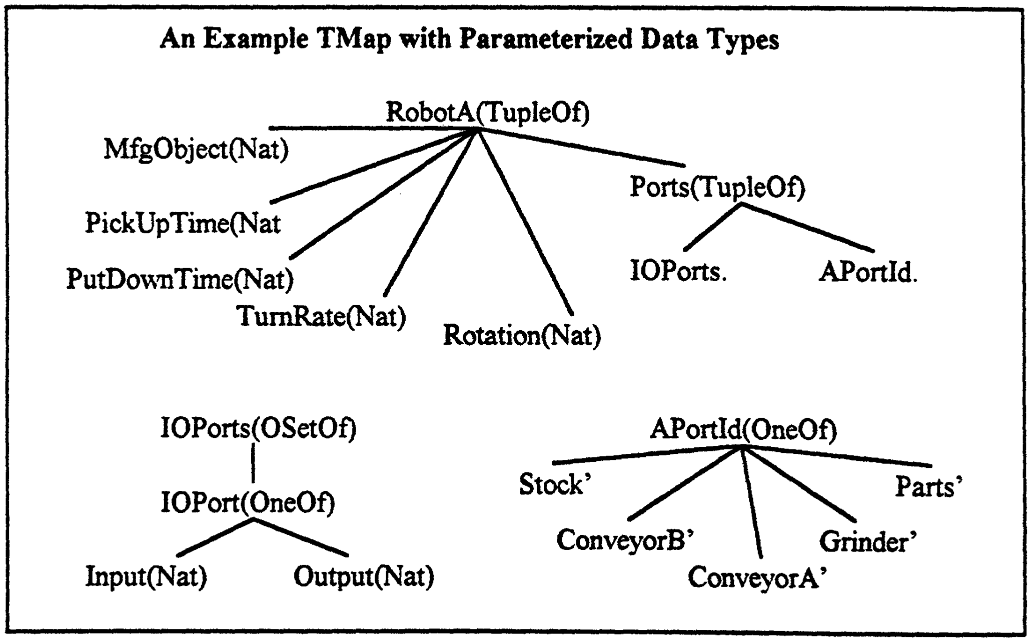


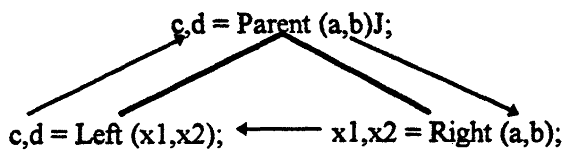
Figure 4-2 Example TMap

4.3 Function Maps (FMaps)

FMaps control the manipulation of data types within the TMap. The FMap has the same graphical structure as a type map. However, types are replaced with functions and the lines which link them represent the relationships between the functions. Ultimately three primitive control structures are used to manipulate TMaps and achieve the goal of the program. Those primitive control structures are illustrated in **Figure 4-3**. They are *Join* (J), *Include* (I), and *Or* (O). These reliable pre-tested primitive control structures remove the interface errors which can occur in other program languages. Data flow for each of the primitive operations is indicated by arrows in

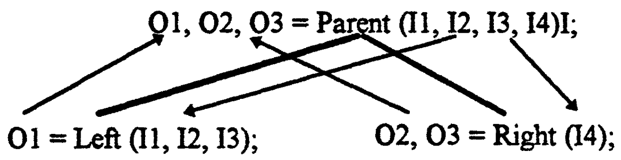
Figure 4-3.

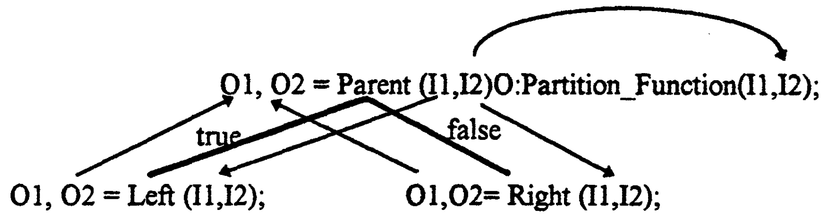
Primative OO1 Control Structures



Join (J) Control Structure
The Join control structure is used for dependent relationships. The flow of data is indicated to the left.

Include (I) Control Structure
The Include control structure is used for independent relationships. The flow of data is indicated to the right.





Or (O) Control Structure
The Or control structure is used for decision making. The Partition Function is used to decide upon which branch is taken. The left branch is taken if the Partition Function is true and the right branch if it is false. The Or statement is similar to the familiar if..then..else statement.

Figure 4-3 Function Map (FMap) Explanation

More complex control structures are also possible, and are shown in the **Figure 4-4**. With these six control structures, and a corresponding TMap, the programmer is able to construct any code which is needed.

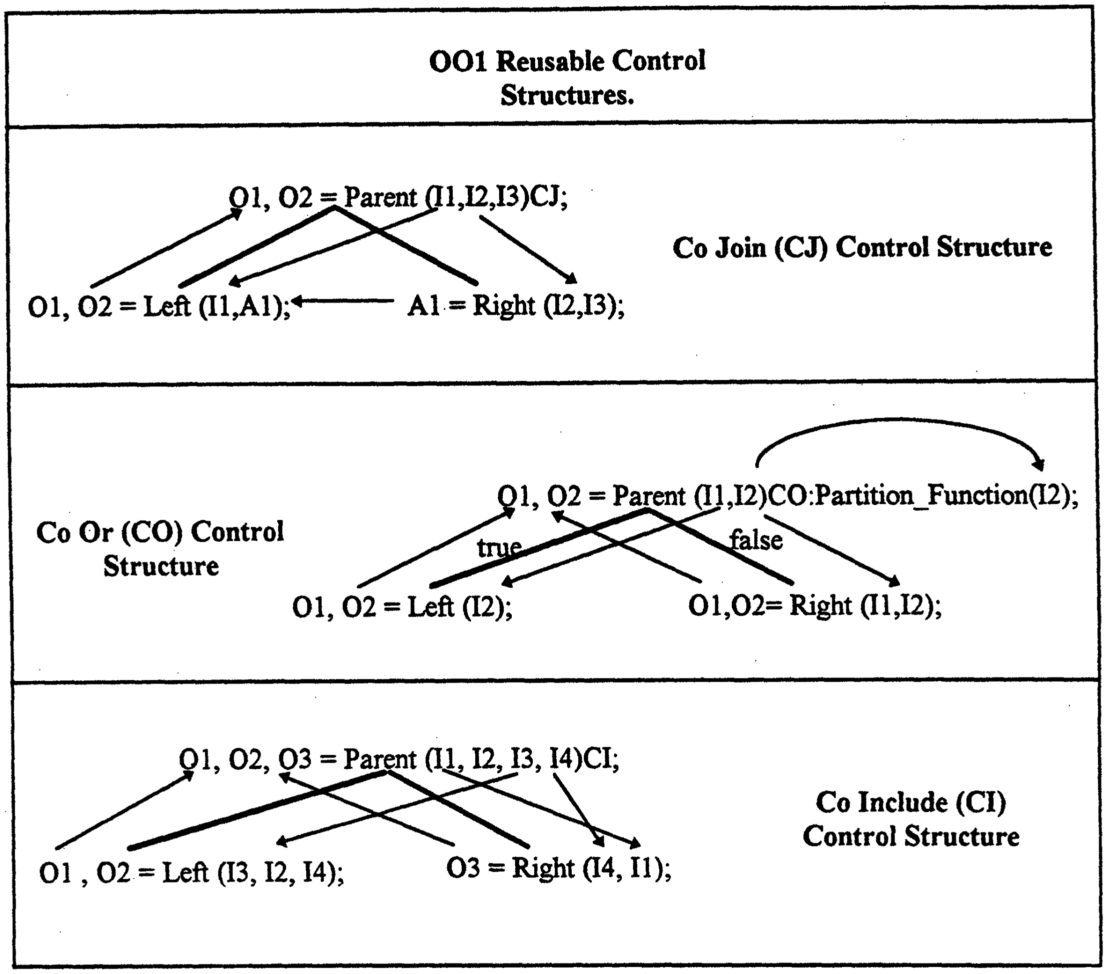

OO1 Reusable Control Structures.
O1, O2 = Parent (I1,I2,I3)CJ;
Co Join (CJ) Control Structure
O1, O2 = Left (I1,A1);
A1 = Right (I2,I3);
O1, O2 = Parent (I1,I2)CO:Partition_Function(I2);
Co Or (CO) Control Structure
true
false
O1, O2 = Left (I2);
O1,O2= Right (I1,I2);
O1, O2, O3 = Parent (I1, I2, I3, I4)CI;
Co Include (CI) Control Structure
O1 , O2 = Left (I3, I2, I4);
O3 = Right (I4, I1);

Figure 4-4 OO1 Reusable Control Structures

The simple example in **Figure 4-5** is perhaps the best way to illustrate the OO1 coding language. The FMap uses the TMap shown in **Figure 4-2**. The program checks to see if the IO port identification [*APortIds(OneOf)*] has a value of Stock'. If it does, the rotation rate [*Rotation(Nat)*] of the robot is updated. Otherwise, the robot's turnrate [*TurnRate(Nat)*] is updated. The update is effected by the *operations* UpdateTurnrate, or UpdateRotation. The use of external operations allows the programmer to modularize the program. Operations are designated by the letters -op- at the end of the line. The name of the FMap in **Figure 2-5** is Update. This can be seen in line one of the FMap. Line one also contains the symbol CJ*3. This indicates that the data flow relationship between the function Update and each of the three primitive operations below in lines two through four is CoJoin.

A step by step examination of each line of the code facilitates a better understanding.

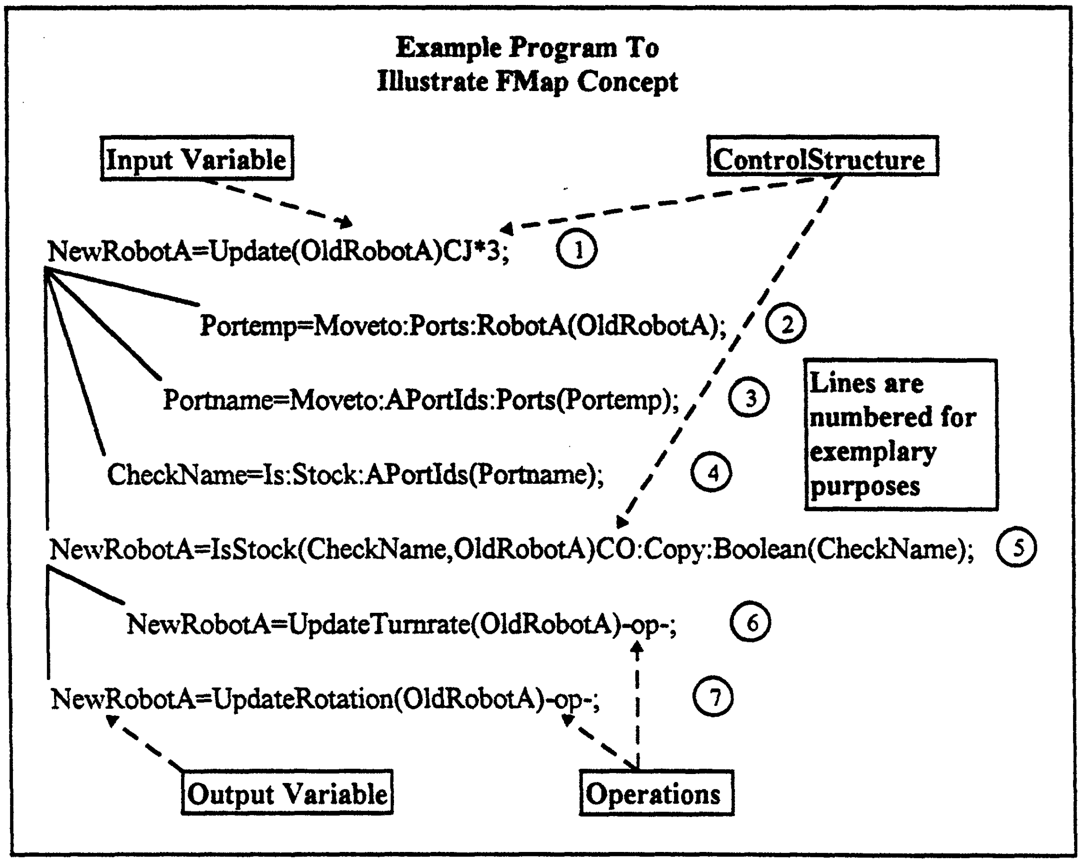

Example Program To
Illustrate FMap Concept
Input Variable
ControlStructure
NewRobotA=Update(OldRobotA)CJ*3; 1
Portemp=Moveto:Ports:RobotA(OldRobotA); 2
Portname=Moveto:APortIds:Ports(Portemp); 3
CheckName=Is:Stock:APortIds(Portname); 4
Lines are numbered for exemplary purposes
NewRobotA=IsStock(CheckName,OldRobotA)CO:Copy:Boolean(CheckName); 5
NewRobotA=UpdateTurnrate(OldRobotA)-op-; 6
NewRobotA=UpdateRotation(OldRobotA)-op-; 7
Output Variable
Operations

Figure 4-5 Example FMap

The first line includes the input variable OldRobotA. OldRobotA is of the type RobotA. This data type is shown in **Figure 2-2**. In line two, the temporary variable Portemp is assigned to the value of the Ports type included in OldRobotA. In line three, the variable PortName is assigned to the APortIds section of OldRobotA. In the fourth line, the Boolean variable CheckName is set to true if the value of APortIds is 'Stock and false if it is not that value. In line five (a Co Or function named IsStock) the value of CheckName is used to determine whether the operations in line six or in line seven is executed. If CheckName is true, then the operation UpdateRotation is executed. If

CheckName is false, then UpdateTurnRate is executed. In each case, the output variable is NewRobotA which is of type RobotA.

4.4 001 System Review

The 001 system is used to illustrate the testing techniques previously discussed in this report. The reader requires a basic understanding of the language. The descriptions above are intended to give the reader that understanding. The FMaps and TMaps used in the next chapter are much more complicated than the ones in **Figures 4-2 and 4-5**, and may initially be difficult to understand. A brief review of the example presented in this **Section** should be of use. For more detailed information upon the subject of FMaps and TMaps one should refer to Ref. (Hami93).

5. FSPT: A Trial Use

5.1 Introduction

In this section, a relatively complex program, known as the Reactor Protection System (RPS), written in the OO1 code system, is explained and then verified using the FSPT. The testing process is described in detail, and the results for each test are listed. The code, unaltered in the first testing phase, is then modified to included several errors, and the testing process is repeated. Excellent results are achieved in both phases. The first phase is described in detail, while the second phase is discussed only briefly. In both phases, the testing strategy remains the same: FSPT. This section demonstrates the effectiveness of Feasible Structured Path Testing on a relatively large modular system which contains looping structures. The FSPT proves to be excellent for detecting errors within the code, as well as aiding in correcting the errors.

5.2 The Reactor Protection System

The modular system used in this section is known as the Reactor Protection System (RPS) is based upon an algorithm printed by Bloomfield (Bloo86), and was developed in prior research by Ouyang (Ouya95). Ouyang developed the RPS in the OO1 specification language which is reviewed in section four of this report.

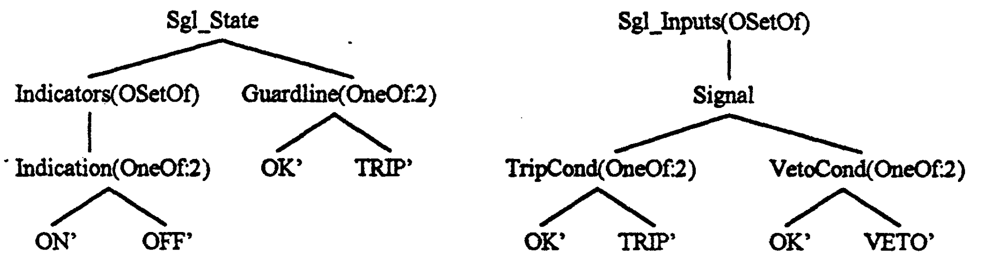
The following is a summary of the specification document for the RPS.

- The system recieves up to 40 pairs of input signals. Each pair represents the trip condition of a plant parameter (TRIP' or OK') and the veto condition (VETO' or OK') of that same parameter.
- The system provides a guardline signal as output (TRIP' or OK'). If a non-vetoed trip condition is present (when a signal is trip condition TRIP' and veto condition OK'), the guardline signal is set to TRIP'. The guardline remains tripped until it is reset.
- A number of output signals (equal to the number of input signals) are attached to indicator lights which provide information on the status of the input signals which correspond to trip inputs. Regardless of the veto status, the lights are either (ON' or OFF') corresponding to the status of the trip condition. If the light is ON', then the corresponding input signal is in the TRIP' condition. The output signals are initiated with all signals in the OFF' state.

The specifications above are used to construct the OO1 software system presented below in **Section 5.2.1**.

5.2.1 RPS OO1 TMap and FMaps

Recall that any system written using the OO1 case tool requires a TMap and an FMap. The RPS is no exception to this rule. The TMap of the code is presented below in **Figure 5-1**.



Type Map of the RPS Program. The Sgl_State type represents the output signals, and the Sgl_Inputs type the input signals.

Figure 5-1 RPS Type Map

One can see that the data structure matches the specifications given above. The Sgl_State type represents the output signal, and is broken into two parts. The first part is the Indicators, an OSetOf type. The list of Indicators contains the type Indication(OneOf:2) which is either ON' or OFF'. The output type Sgl_State also contains the guardline which has the values of either OK' or TRIP'. The input type is called Sgl_Inputs and is an OSetOf. The Sgl_Inputs list contains both the Trip and the Veto conditions (TripCond and VetoCond). The trip condition can be either OK' or TRIP' and the veto condition is either OK' or VETO'. This TMap is used to satisfy the specification requirements along with the FMaps listed below.

The FMaps of the RPS are organized in a modular fashion. The first of the FMaps in **Figure 5-2** is the highest level FMap that binds all of the modules together. The FMaps are annotated to aid the reader unfamiliar with the 001 system.

```
NewState=Update(PreState,NewSignals)CJ;
        ChkGL=CheckGLOK(PreState)-op-;
NewState=Real_Update(PreState,NewSignals,ChkGL)CO:Copy:Boolean(ChkGL);
        NewState=Update_Upon_Trip(PreState,NewSignals)-op-;
NewState=Update_Upon_Ok(PreState,NewSignals)CJ;
        State1=Update_GL(PreState, NewSignals)-op-;
NewState=Update_Indi(State1, NewSignals)-op-;
```

The RPS highest level FMap. It calls all of the operations listed in the FMaps below.

Figure 5-2 High Level FMap Update_State

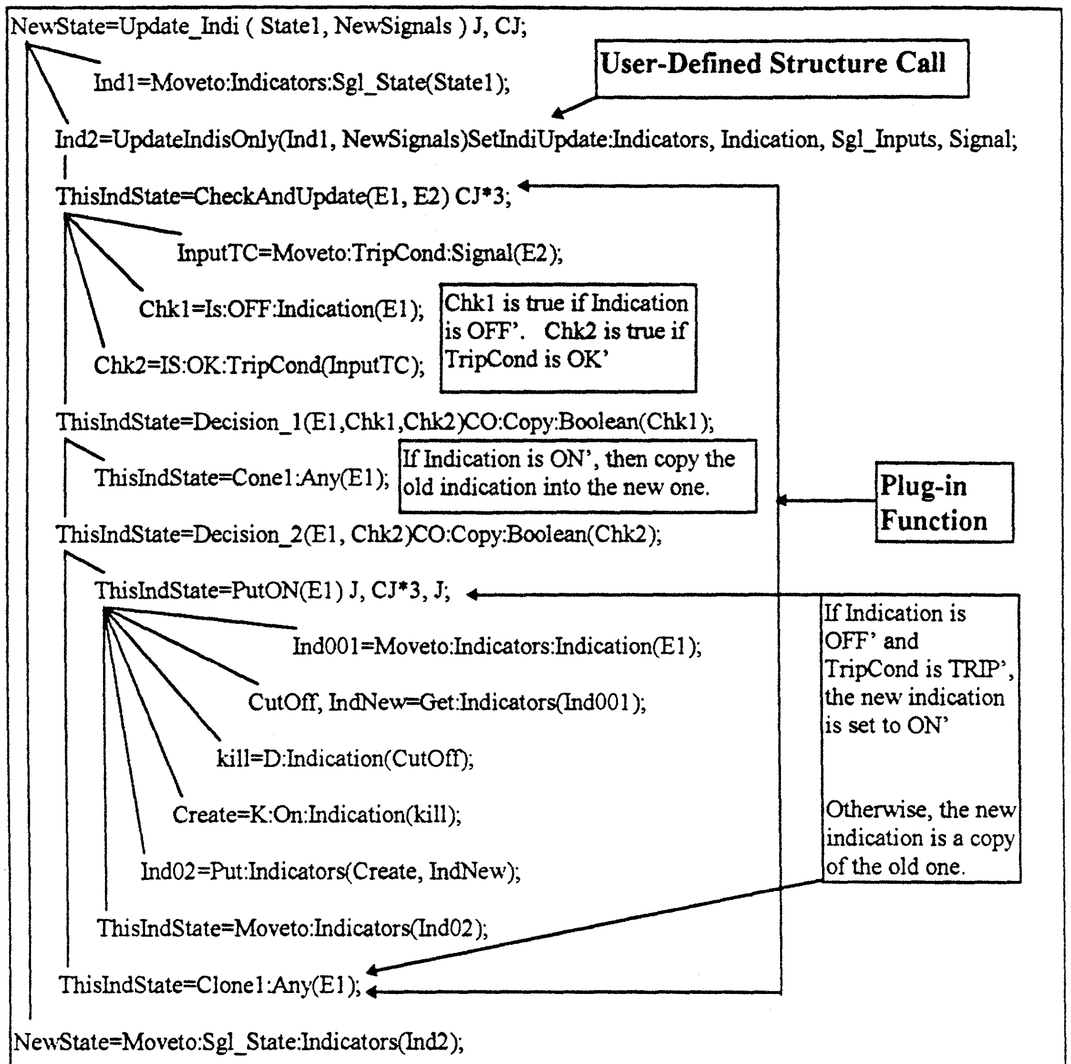
The remaining FMaps are pieces of the RPS that all link to the high level code in **Figure 5-2**.

```
Chk1=CheckGLOK(PreState)J;
        OldGL=Moveto:Guardline:Sgl_State(PreState);
Chk1=is:OK:Guardline(OldGL);
```

<u>Operation CheckGLOK</u>

This operation check the status of the guardline and assigns a true to the varaible Chk1 if it is OK' and a false if it is TRIP'.

Figure 5-3 Operation CheckGLOK FMap



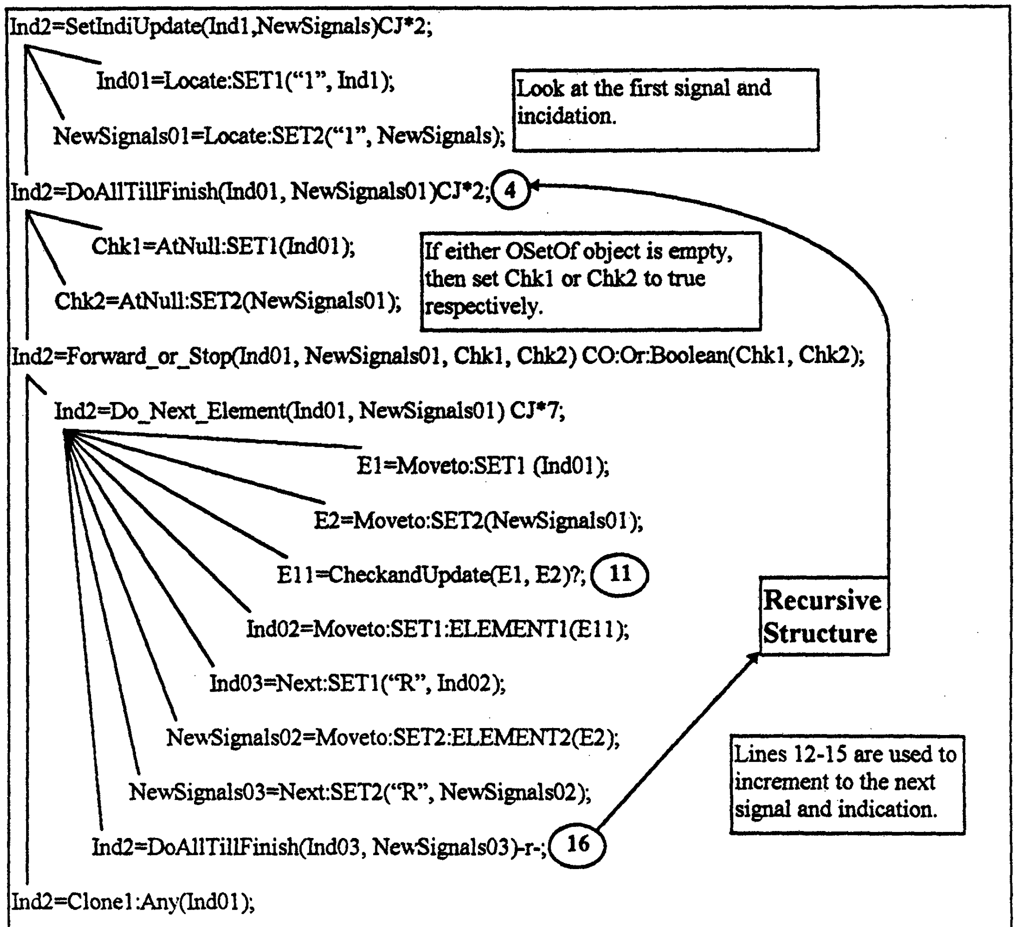
```
NewState=Update_Indi ( State1, NewSignals ) J, CJ;
    Ind1=Moveto:Indicators:Sgl_State(State1);
    Ind2=UpdateIndisOnly(Ind1, NewSignals)SetIndiUpdate:Indicators, Indication, Sgl_Inputs, Signal;
    ThisIndState=CheckAndUpdate(E1, E2) CJ*3;
        InputTC=Moveto:TripCond:Signal(E2);
        Chk1=Is:OFF:Indication(E1);
        Chk2=IS:OK:TripCond(InputTC);
    ThisIndState=Decision_1(E1,Chk1,Chk2)CO:Copy:Boolean(Chk1);
        ThisIndState=Cone1:Any(E1);
    ThisIndState=Decision_2(E1, Chk2)CO:Copy:Boolean(Chk2);
        ThisIndState=PutON(E1) J, CJ*3, J;
            Ind001=Moveto:Indicators:Indication(E1);
            CutOff, IndNew=Get:Indicators(Ind001);
            kill=D:Indication(CutOff);
            Create=K:On:Indication(kill);
            Ind02=Put:Indicators(Create, IndNew);
        ThisIndState=Moveto:Indicators(Ind02);
    ThisIndState=Clone1:Any(E1);
NewState=Moveto:Sgl_State:Indicators(Ind2);
```

User-Defined Structure Call
Chk1 is true if Indication is OFF'. Chk2 is true if TripCond is OK'
If Indication is ON', then copy the old indication into the new one.
Plug-in Function
If Indication is OFF' and TripCond is TRIP', the new indication is set to ON'
Otherwise, the new indication is a copy of the old one.

Operation Update Indi

This operation updates the output indicators to relect their proper state as indicated by the specifications. This operation is only executed if the guardline isn't tripped. Note the user-defined structure call and the plug-in function. This is further explained in the Figure 5-5.

Figure 5-4 Operation Update_Indi FMap



```
Ind2=SetIndiUpdate(Ind1,NewSignals)CJ*2;
    Ind01=Locate:SET1("1", Ind1);
    NewSignals01=Locate:SET2("1", NewSignals);
Ind2=DoAllTillFinish(Ind01, NewSignals01)CJ*2; (4)
    Chk1=AtNull:SET1(Ind01);
    Chk2=AtNull:SET2(NewSignals01);
Ind2=Forward_or_Stop(Ind01, NewSignals01, Chk1, Chk2) CO:Or:Boolean(Chk1, Chk2);
    Ind2=Do_Next_Element(Ind01, NewSignals01) CJ*7;
        E1=Moveto:SET1 (Ind01);
        E2=Moveto:SET2(NewSignals01);
        E11=CheckandUpdate(E1, E2)?; (11)
        Ind02=Moveto:SET1:ELEMENT1(E11);
        Ind03=Next:SET1("R", Ind02);
        NewSignals02=Moveto:SET2:ELEMENT2(E2);
        NewSignals03=Next:SET2("R", NewSignals02);
        Ind2=DoAllTillFinish(Ind03, NewSignals03)-r-; (16)
Ind2=Clone1:Any(Ind01);
```

Look at the first signal and incidation.

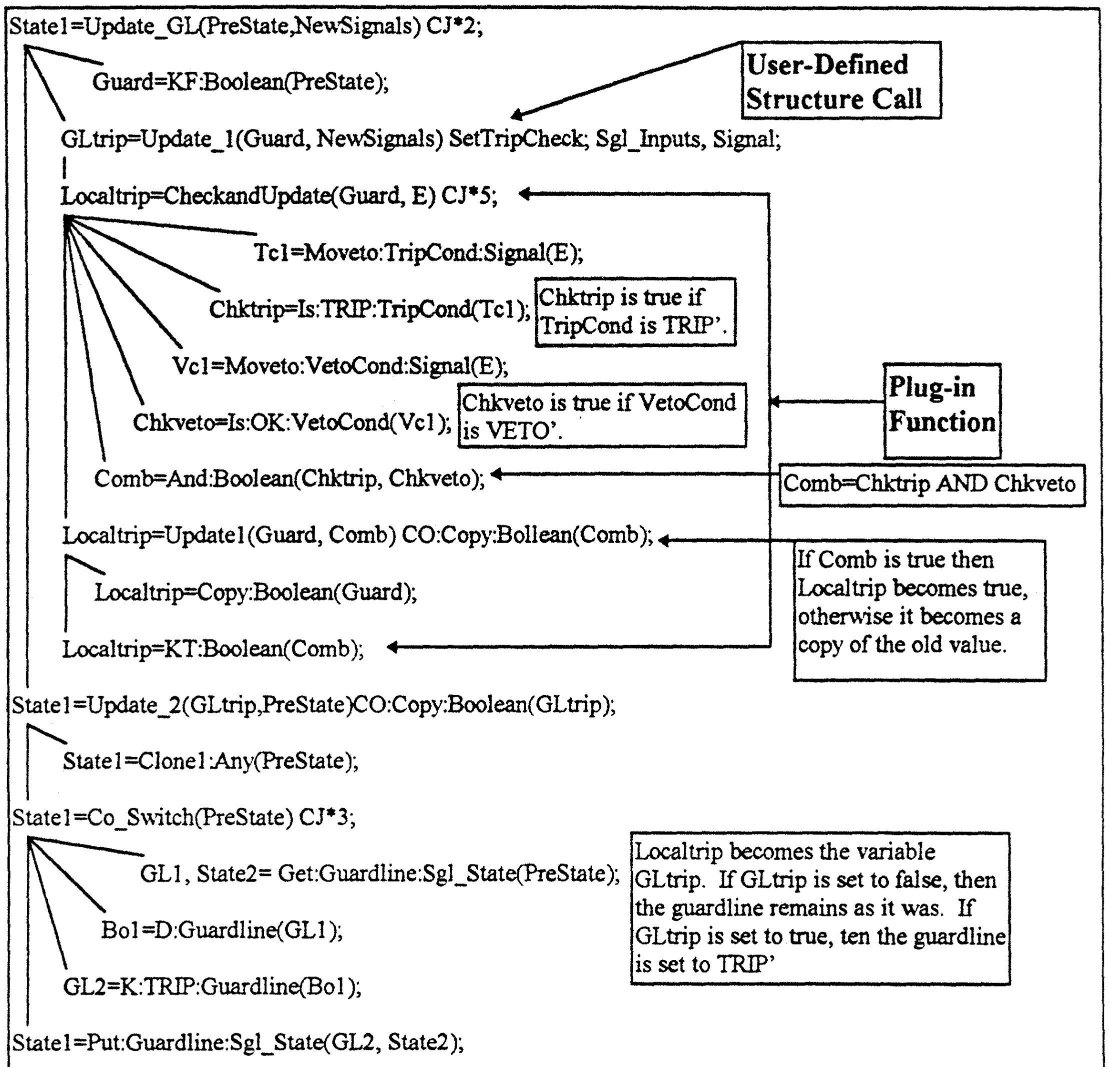
If either OSetOf object is empty, then set Chk1 or Chk2 to true respectively.

Lines 12-15 are used to increment to the next signal and indication.

User-Defined Structure <u>SetIndiUpdate</u>

This structure is used to traverse the entire OSetOf input signals and output indications. It is used by both operations Update_Upon_Trip and Update_Indi. The structure contains a "-r-" structure in line 16. This indicates a recursive operation or a loop. The values of the variables Ind03 and NewSignals03 in line 16 are passed to the variables Ind01 and NewSignals01 respectively for the next pass. The recursion allows the structure to examine every field in the ordered set. The "?" function in line 11 allows the operation using the structure to insert unique functions into the structure at that point. These unique functions are located below the structure calls in operations Update_Upon_Trip and Update_Indi and are known as plug-in functions.

Figure 5-5 Structure SetIndiUpdate FMap

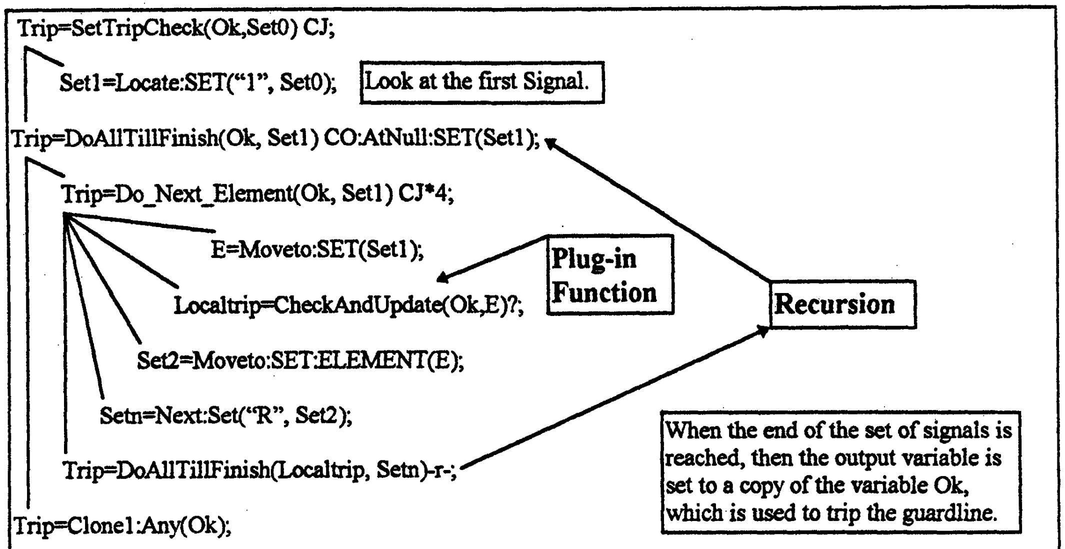

User-Defined Structure Call
Chktrip is true if TripCond is TRIP'.
Chkveto is true if VetoCond is VETO'.
Plug-in Function
Comb=Chktrip AND Chkveto
If Comb is true then Localtrip becomes true, otherwise it becomes a copy of the old value.
Localtrip becomes the variable GLtrip. If GLtrip is set to false, then the guardline remains as it was. If GLtrip is set to true, ten the guardline is set to TRIP'

```
State1=Update_GL(PreState,NewSignals) CJ*2;
    Guard=KF:Boolean(PreState);
  GLtrip=Update_1(Guard, NewSignals) SetTripCheck; Sgl_Inputs, Signal;
  Localtrip=CheckandUpdate(Guard, E) CJ*5;
                Tc1=Moveto:TripCond:Signal(E);
            Chktrip=Is:TRIP:TripCond(Tc1);
          Vc1=Moveto:VetoCond:Signal(E);
        Chkveto=Is:OK:VetoCond(Vc1);
      Comb=And:Boolean(Chktrip, Chkveto);
  Localtrip=Update1(Guard, Comb) CO:Copy:Bollean(Comb);
      Localtrip=Copy:Boolean(Guard);
  Localtrip=KT:Boolean(Comb);
State1=Update_2(GLtrip,PreState)CO:Copy:Boolean(GLtrip);
  State1=Clone1:Any(PreState);
State1=Co_Switch(PreState) CJ*3;
        GL1, State2= Get:Guardline:Sgl_State(PreState);
      Bo1=D:Guardline(GL1);
  GL2=K:TRIP:Guardline(Bo1);
State1=Put:Guardline:Sgl_State(GL2, State2);
```

Operation Update_GL

This operation updates the guardline signal. It uses the user-defined structure SetTripCheck. This structure is called in line 3 of the operation. The plug-in function is contained in lines 4-12.

Figure 5-6 Operation Update_GL FMap



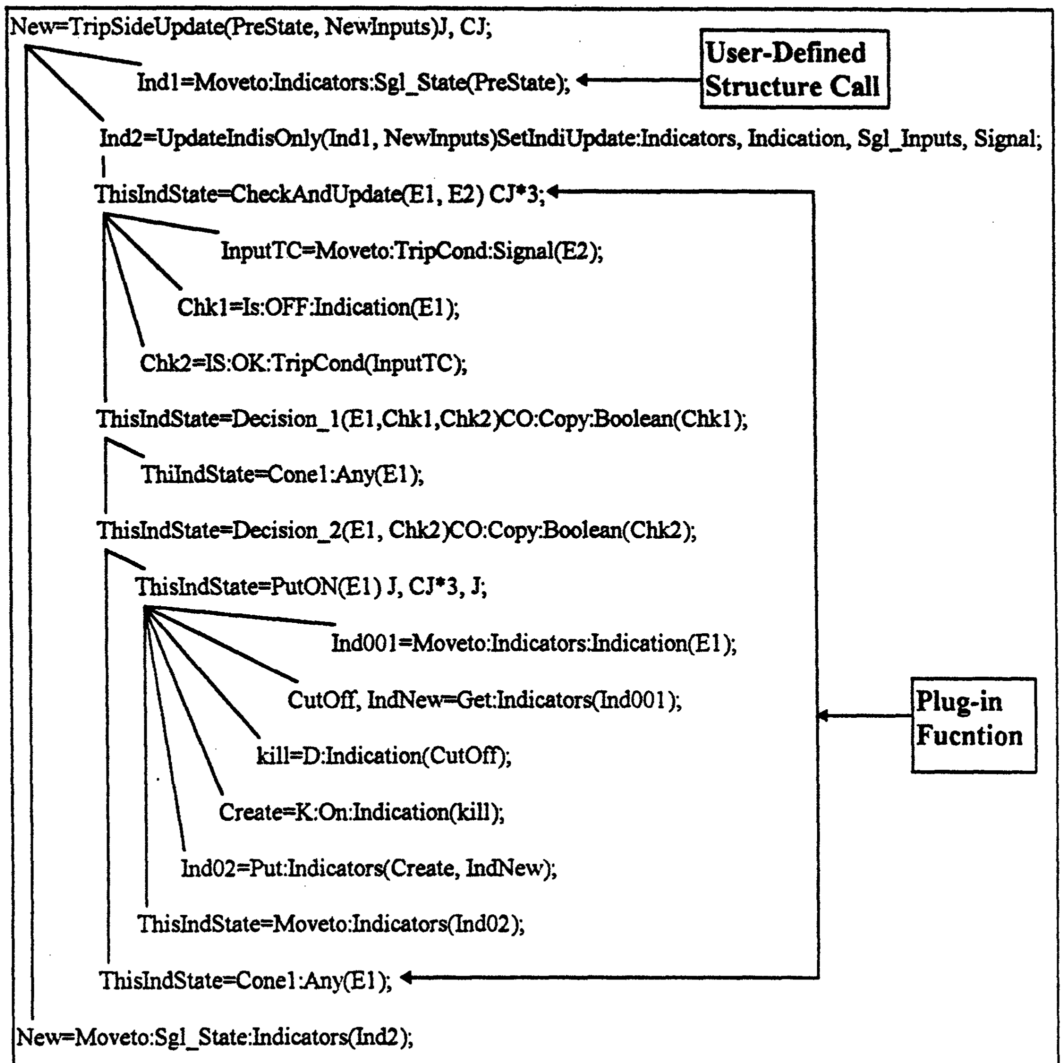
```
Trip=SetTripCheck(Ok,Set0) CJ;
    Set1=Locate:SET("1", Set0);
Trip=DoAllTillFinish(Ok, Set1) CO:AtNull:SET(Set1);
    Trip=Do_Next_Element(Ok, Set1) CJ*4;
                    E=Moveto:SET(Set1);
                Localtrip=CheckAndUpdate(Ok,E)?;
            Set2=Moveto:SET:ELEMENT(E);
        Setn=Next:Set("R", Set2);
    Trip=DoAllTillFinish(Localtrip, Setn)-r-;
Trip=Clone1:Any(Ok);
```

Look at the first Signal.
Plug-in Function
Recursion
When the end of the set of signals is reached, then the output variable is set to a copy of the variable Ok, which is used to trip the guardline.

User-Defined Structure SetTripCheck

This structure is used by operation Update_GL to traverse the set of input signals to check for the condition of trip condition TRIP' and veto condition OK'. Should this occur, the guardline is tripped. This structure contains recursion and a plug-in function. They are noted above.

Figure 5-7 Structure SetTripCheck FMap


```
New=TripSideUpdate(PreState, NewInputs)J, CJ;
        Ind1=Moveto:Indicators:Sgl_State(PreState);
    Ind2=UpdateIndisOnly(Ind1, NewInputs)SetIndiUpdate:Indicators, Indication, Sgl_Inputs, Signal;
    ThisIndState=CheckAndUpdate(E1, E2) CJ*3;
            InputTC=Moveto:TripCond:Signal(E2);
          Chk1=Is:OFF:Indication(E1);
        Chk2=IS:OK:TripCond(InputTC);
    ThisIndState=Decision_1(E1,Chk1,Chk2)CO:Copy:Boolean(Chk1);
        ThiIndState=Cone1:Any(E1);
    ThisIndState=Decision_2(E1, Chk2)CO:Copy:Boolean(Chk2);
        ThisIndState=PutON(E1) J, CJ*3, J;
                  Ind001=Moveto:Indicators:Indication(E1);
                CutOff, IndNew=Get:Indicators(Ind001);
              kill=D:Indication(CutOff);
            Create=K:On:Indication(kill);
          Ind02=Put:Indicators(Create, IndNew);
        ThisIndState=Moveto:Indicators(Ind02);
    ThisIndState=Cone1:Any(E1);
New=Moveto:Sgl_State:Indicators(Ind2);
```

Operation Update Upon Trip

This operation updates the the output signals according to the specification document. This operation also uses the user-defined structure SetIndiUpdate as indicated above. The operation is nearly identical to Update_Indi.

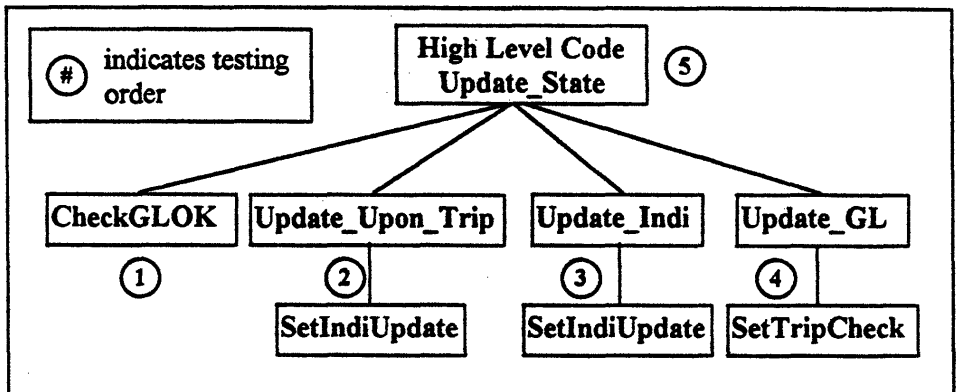
Figure 5-8 Operation Update_Upon_Trip FMap

The presence of *user-defined structures* and their corresponding *plug-in functions* may be confusing as they were not explained in **Section 4**. User-defined structures are similar to procedures or functions in other languages. They are reusable, and in the case of OO1, contain plug-in functions. A plug-in functions allow the user to make the user-defined structure into a more versatile tool. The user-defined structures in the RPS, SetIndiUpdate, and SetTripCheck are designed to examine each item in an OSetOf type. The Structures loop through each of the items until there are none left, or a condition to exit early is met. The plug-in functions do the work by examining the content of each item in the ordered set and acting accordingly.

A complete understanding of the RPS is not necessary in order to understand the process of the FSPT. The FMaps are broken down into flowgraphs in **Section 5.3**. Examination of the flowgraphs for each FMap should clear up any questions the user has about the control flow of the FMaps.

5.3 The FSPT of the RPS

Using the FSPT for the RPS begins by determining the overall testing strategy. As discussed in **Section 3**, this strategy is an incremental one. Examining the high level FMap, one sees that the RPS is made up of the high level code, four operations and two user-defined structures. Since the user-defined structures are executed differently for each operation, they are tested with the operations. The testing strategy is as follows.



The RPS incremental testing strategy is indicated on the diagram above. CheckGLOK is tested first. The remaining operations are then tested with their user defined structures. Finally, the high level code Update_State is tested.

Figure 5-9 Order Of Testing the RPS

First, the tester tests operations CheckGLOK, Update_Upon_Trip, Update_GL, and Update_Indi independently. Once these operations have been tested, the high level code Update_State can be tested.

5.3.1 Testing the Operations

Using the FSPT, the operations noted above are tested separately. This section is broken down into five subsections: one subsection for each of the four operations. In each subsection, the five steps of the FSPT are performed. Test cases are formatted, and the results of those tests are given. In order to limit testing time, the k value for every test is equal to two. Thus, only two looping iterations, at most, are tested for each loop present.

5.3.1.1 Operation CheckGLOK

(See **Figure 5-3**)

The first operation to be tested is CheckGLOK. This operation's code is in-line, meaning there are no control branches. There are no functional variables, and the flowgraph is a single node. It is not shown here. Only one test is needed to test CheckGLOK. Using the OO1 testing suite, the tester uses as input data the PreState input object of type Sgl_State which has the guardline either in the TRIP' or OK' position. The value of Chk1, the output variable, is then recorded for later use. The value of PreState is recorded also. In case of an error, it is helpful to know the value of every input variable. In this case there is only one indication which is set to OFF' and the value of the guardline is OK'. The resulting value of the output variable Chk1 is true.

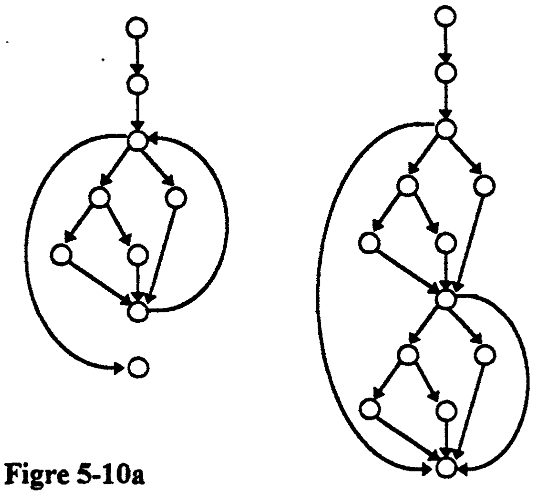
5.3.1.2 Operation Update_Upon_Trip

(see **Figure 5-8**)

Operation Update_Upon_Trip is also tested using the FSPT. The steps of the process are listed below with explanations given where necessary.

Step One: Construct the flowgraph.

The flowgraph is shown in **Figure 5-10**. Since k has been chosen to be two, the flowgraph in **Figure 5-10** has been altered to reflect this fact.



Figre 5-10a

Figre 5-10b

The flowgraph of Operation Update_Upon_Trip in its original form above (Figure 5-10a) and altered to included only two loop iterations to the right (Figure 5-10b).

Figure 5-10 Update_Upon_Trip Flowgraph

Step Two:

The second step requires that the tester recognize the input, output, and transitional variables. The following list specifies the input variables:

Input Variables: PreState (Ind1, Ind01, E1, Ind001, Cutoff, IndNew)

NewInputs (NewSignals, NewSignals01, E2, InputTC, NewSignals02, NewSignals03)

The input variables are listed in the first line of the operation. They are PreState and NewInputs. PreState is of type Sgl_State, and NewInputs is also of type Sgl_Inputs. Throughout the operation, several variables are assigned to pieces of these variables by using Moveto statements. For instance, in the second line of the operation, the variable Ind1 is assigned to the Indicators type of the variable PreState. Thus, Ind1 is designated as an input variable and included in this list. In the list of input variables given below, the variables listed in parentheses are variables which are unaltered pieces of the original input variables: PreState and NewInputs.

The output variable for any operation is easily located in the OO1 system. The left hand side of the equality in the first line of code lists them. In this case, the variable is known as New and is of type Sgl_State.

Output Variables: New

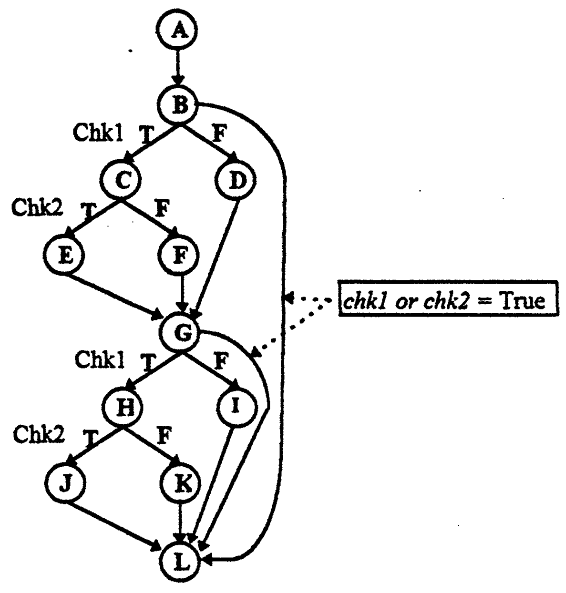
Location of the transitional variables is a bit more complex due to the presence of recursion and user-defined structures. The transitional variables are recorded as follows:

Transitional Variables: *Chk1*, *Chk2*, Chk1, Chk2, (ThisIndState, *E11*), kill, Create, Ind02, Ind02, (Ind03), Ind2

These variables are chosen through use of the criteria listed in **Section 3**. The variables shown in *italics* are variables used in the user-defined structure SetIndiUpdate. The variables shown inside parentheses are variables passed between the user-defined structure to the plug-in function using a different name. In this instance, *E11* is passed to the plug-in function CheckAndUpdate as the variable ThisIndState.

Step 3:

In this step, one develops the altered flowgraph which depicts the unique states of those transitional variables which are not yet at the end of their development. This task is completed and the results are displayed in **Figure 5-11**.



Operation Update_Upon_Trip
Step Three Flowgraph

The flowgraph above indicates the value of the decision variables Chkl and Chk2 and *Chkl* and *Chk2* (user-defined structure variables) for certain branches. Use of this flowgraph aids in the development of testing cases.

Figure 5-11 Update_Upon_Trip Step 3 Flowgraph

<u>Step 4</u>

The fourth step involves calculating the value of the McCabe metric of the flowgraph in **Figure 5-11**, and then locating the complete set of linearly independent paths. The McCabe metric of the structure shown above has a value of seven (v=17-12+2=7). Thus, there are seven test paths necessary to test the operation. Those paths are listed in **Table 5-1**. It is determined that there are no unfeasible test paths. This is done by tracing the six paths through the code to determine if they are executable. This conclusion is further verified in the fifth step of the FSPT.

Test Paths for Operation Update_Upon_Trip

Path Number	Sequence
0	A+B+C+F+G+H+K+L
1	A+B+L
2	A+B+D+G+H+K+L
3	A+B+C+E+G+H+K+L
4	A+B+C+F+G+L
5	A+B+C+F+G+I+L
6	A+B+C+F+G+H+J+L

Table 5-1 Test Paths: Update_Upon_Trip

Step 5

In the fifth step of the FSPT, the values of the input variables are determined for each of the selected test paths and unfeasible test paths are elimintated. The results are indicated in the Table 5-2. Question marks denote that the test is indifferent to the value of the vaiable indicated. However the values of such variables are recorded in order to aid in the location of errors.

Full Set of Test Paths for Operation Update Upon Trip

Path	Indication1	Trip Condition 1	Indication2	Trip Condition 2
0	OFF'	TRIP'	OFF'	TRIP'
1	* ON'	* NULL	* OFF'	* NULL
2	ON'	? TRIP'	ON'	? OK'
3	OFF'	OK'	OFF'	TRIP'
4	OFF'	TRIP'	* OFF'	* NULL
5	OFF'	TRIP'	ON'	? TRIP'
6	OFF'	TRIP'	OFF'	OK'

? - indicates that the value of the object is irrelevant for test case indicated.
* - indicates that one or both of the pair of indication and trip condition is null.

Table 5-2 Test Cases: Update_Upon_Trip

It is necessary to record every value of the input variables, even if they are not relevant to the execution of the path of immediate interest. Should an error be located, the tester must know the value of every input variable used in each test case. In this case, the value of the Guardline is always TRIP', and the value of the veto condition is always OK' except when the signal is null.

The results are listed in the **Table 5-3**. They are identical to the expected results obtained from the specifications. If the indicator is OFF' and the trip condition is TRIP', then the indicator should be turned ON'. In any other case, the indicator remains where it is set. Each test case satisfies this specification.

Test Results From Update_Upon_Trip

Path	Indication 1	Indication 2
0	ON'	ON'
1	ON'	OFF'
2	ON'	ON'
3	OFF'	ON'
4	ON'	OFF'
5	ON'	ON'
6	ON'	ON'

Table 5-3 Update_Upon_Trip Test Results

The five step FSPT process has been successful in testing operation Update_Upon_Trip. It appears that the module works as specified. However, the specification document is not written in modular form. One can not conclude that the operation is correct until the final test has been done of the high level code. Only then can the tester conclude that the operation behaves according to the specification document. Were the specification written in a more specific modular form, the tester could consult the document to determine the correctness of the operation.

The remaining operations are not given such a detailed treatment. Much of the work is presented without comment.

5.3.1.3 Operation Update_GL

The purpose of operation Update_GL (see **Figure 5-6**) is to update the guardline signal. The operation checks to see if any of the input signals have a trip condition of TRIP' and a veto condition of OK'. If this case occurs, then the guardline signal is tripped. Otherwise, the guardline is left in its current state.

Step 1 Construct the flowgraph and alter it to include only two loop iterations.

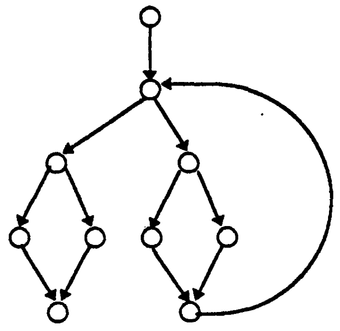


Figure 5-12a

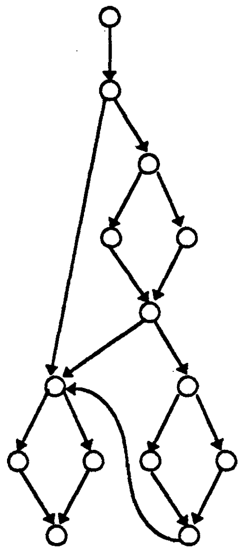


Figure 5-12b

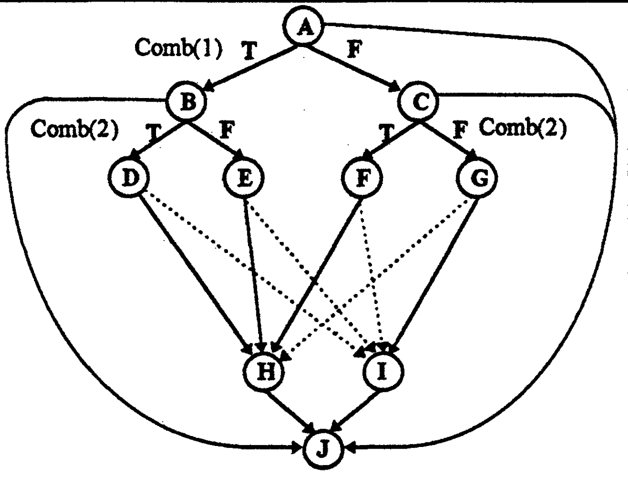
Flowgraph of Operation Update_GL in its original version above (Figure 5-12a) and altered to include only two loop iterations to the right (Figure 5-12b).

Figure 5-12 Update_GL Flowgraph

Step Two List the Input, Output, and Transitional Variables as follows:

Input Variables:	PreState (GL1, State2)
	NewSignals (Set0, Set1, E, Tc1, Vc1, Set2, Setn)
Output Variable:	State1
Transitional Variables:	Chktrip, Chkveto, Comb .

Step Three Construct the new flowgraph which reflects only unique states of the input variables.



Operation Update_GL
Step Three Flowgraph
Values of the control variable Comb which cause certain paths to be executed are noted above. Unfeasable paths which have been remove are indicated by dashed lines.

Figure 5-13 Update_GL Step 3 Flowgraph

Step Four Calculate the value of the McCabe metric and choose the linearly independent paths. The McCabe metric of the flowgraph above has a value of eleven (v=19-10+2=11). **Table 5-4** contains the paths which are chosen as the set of linearly independent paths.

Test Paths for Operation Update_GL

Path Number	Sequence
0	A+B+E+H+J
1	A+J
2	A+C+F+H+J
3	A+C+G+I+J
4	A+C+J
5	A+B+J
6	A+B+D+H+J
7	A+B+D+I+J
8	A+B+E+I+J
9	A+C+F+I+J
10	A+C+G+H+J

Table 5-4 Test Paths: Update_GL

Step Five Determine the values of the input variables which cause the executable paths above to be executed.

The fifth step results in the removal of several unfeasible paths. The paths are indicated by dashed lines in **Figure 5-12.** If at any time in the operation the value of Comb is set to true, then the guardline signal is tripped. Node H is only reached if the value of Comb is set to true at some point. The reverse is also true. If Comb remains false throughout the operation, then only node I can be reached. Thus, the paths indicated with dashed lines are not feasible and have been eliminated.

Test Cases for Operation Update_GL

Path	TripCond(1)	VetoCond(1)	TripCond(2)	VetoCond(2)
0	TRIP'	OK'	** OK'	** 'VETO
1	NULL	NULL	NULL	NULL
2	* TRIP'	* 'VETO	TRIP'	OK'
3	* OK'	* OK'	** TRIP'	** 'VETO
4	* TRIP'	* 'VETO	NULL	NULL
5	TRIP'	OK'	NULL	NULL
6	TRIP'	OK'	TRIP'	OK'

* indicates that the set of TripCond(1) and VetoCond(1) can be anything but TRIP' and OK'.
** indicates that the set of TripCond(2) and VetoCond(2) can be anything but TRIP' and OK'.

Table 5-5 Test Cases: Update_GL

The guardline is always OK' when the program initializes. The trip condition is always OK' as well.

Test Results from Operation Update_GL

Path	Guard Line
0	TRIP'
1	OK'
2	TRIP'
3	OK'
4	OK'
5	TRIP'
6	TRIP'

Table 5-6 Test Results: Update_GL

This concludes the testing of operation Update_GL. The output listed above is identical to that expected based upon the RPS specification. The guardline should TRIP' if the veto condition is OK' and the trip condition is TRIP'. This is the case in paths 0, 2, 5 and 6. In the other paths, the guardline is left in the OK' setting.

5.3.1.4 Operation Update_Indi

Note that operation Update_Indi (see **Figure 5-4**) is essentially the same as operation Update_Upon_Trip. Thus, the flowgraph and testing steps are all the same in both cases.

Step One Construct the flowgraph and alter it to include only two loop iterations.

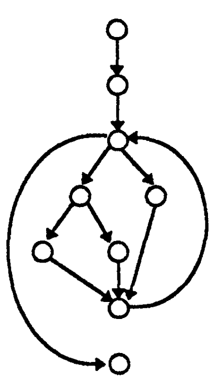


Figure 5-14a

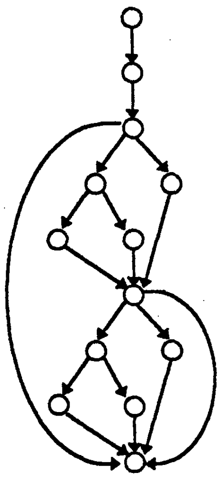


Figure 5-14b

The flowgraph of Operation Update_Indi in its original form above (Figure 5-14a) and altered to included only two loop iterations to the right (Figure 5-14b).

Figure 5-14 Update_Indi Flowgraph

Step Two List the input, output, and transitional variables as follows:

Input Variables: State1 (Ind1, Ind01, E1, Ind001)

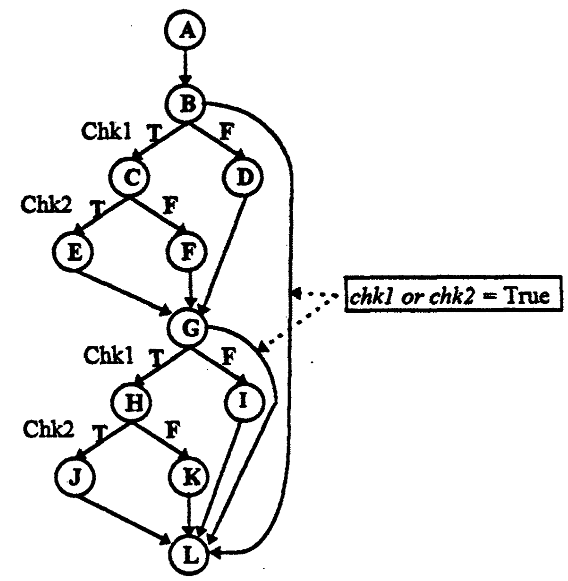
New Signals (New Signals01, E2, InputTC, New Signals02, New Signals03)

Output Variable: NewState

Transitional Variables: Size1, Size2, Chksize, Chk1, Chk2, Ind2, Chk1(s), Chk2(s), E11

(ThisIndState, Ind02, Ind03) .

Step Three Construct the new flowgraph which reflects only unique states of the input variables.



Operation Update_Indi
Step Three Flowgraph

The flowgraph above indicates the value of the decision variables Chk1 and Chk2 and *Chk1* and *Chk2* (user defined structure variables) for certain branches. Doing this aids in the devlopment of testing cases.

Figure 5-15 Update_Indi Step 3 Flowgraph

Step Four Calculate the value of the McCabe metric and choose the linearly independent paths.

Test Paths for Operation Update_Upon_Trip

Path Number	Sequence
0	A+B+C+F+G+H+K+L
1	A+B+L
2	A+B+D+G+H+K+L
3	A+B+C+E+G+H+K+L
4	A+B+C+F+G+L
5	A+B+C+F+G+I+L
6	A+B+C+F+G+H+J+L

Table 5-7 Test Paths: Update_Upon_Trip

Step Five Determine the value of the input variables which cause the paths above to be executed.

Full Set of Test Paths for Operation Update_Indi

Path	Indication1	Trip Condition 1	Indication2	Trip Condition 2
0	OFF’	TRIP’	OFF’	TRIP’
1	* ON’	* NULL	* OFF’	* NULL
2	ON’	? TRIP’	ON’	? OK’
3	OFF’	OK’	OFF’	TRIP’
4	OFF’	TRIP’	* OFF’	* NULL
5	OFF’	TRIP’	ON’	? TRIP
6	OFF’	TRIP’	OFF’	OK’

? - indicates that the value of the object is irrelevant for test case indicated.

* - indicates that one or both of the pair of indication and trip condition is null.

Table 5-8 Test Cases: Update_Upon_Trip

The value of the Guardline is always TRIP’, and the value of the veto condition is always OK’ except when the signal is null.

Test Results From Update_Indi

Path	Indication 1	Indication 2
0	ON’	ON’
1	ON’	OFF’
2	ON’	ON’
3	OFF’	ON’
4	ON’	OFF’
5	ON’	ON’
6	ON’	ON’

Table 5-9 Test Results: Update_Upon_Trip

These results are identical to that expected based upon the RPS specification. The only remaining code is the high level code Update_State. which is tested next.

5.3.2 Testing the High Level Code, Update_State

All of the operations have been completely tested. The high level code Update_State can be tested by considering the operation calls as single lines of code.

The testing set is therefore very simple. Upon inspection one can derive the test cases.

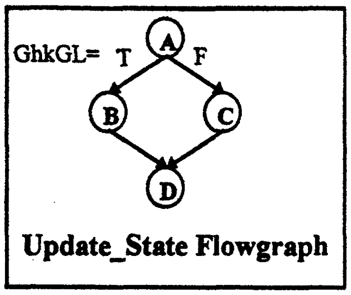


Figure 5-16 Update_State Flowgraph

The code consists of a single decision which depends upon the variable ChkGL, which is the output variable of operation CheckGLOK. There are two test cases for the Ground Level Code. A base path A+B+D and path one A+C+D. The state of the input variables in the paths are as follows:

Test Paths for Update_State

Path	Pre State (GuardLine)
0	OK'
1	TRIP'

Table 5-10 Test Paths: Update_State

The value of the remaining data fields must also be recorded. There is only one Indication and it is set to OFF'. There is also one Signal which is set to trip condition TRIP' and veto condition OK'. These test cases include the test case for the CheckGLOK operation. Thus the ground level code and CheckGLOK can be tested simultaneously. This removes one unnecessary test.

The results of testing the high level code are shown below.

Test Results for Update_State

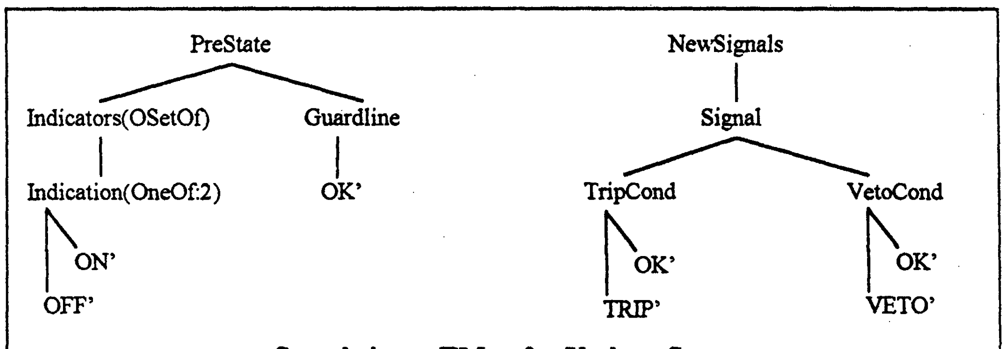
Path	Pre State (GuardLine)	Indication
0	OK'	ON'
1	TRIP'	ON'

Table 5-11 Test Cases: Update_State

These test results are reflective of the specification document. The program has been Feasible Structured Path Tested (FSPT) with a k value of two.

5.3.3 Conclusion of the FSPT of the RPS

The entire RPS code can be testing using the FSPT in 23 tests when only two loop iterations are allowed. As the value of k increases, the number of test cases increases rapidly. The RPS has been tested and the entire input / output set been exercised according to the FSPT criteria. The tester must now examine the results to determine if they match the requirements of the specification document. In this case, the tester develops an oracle directly from the specification document. Each operation is tested over a wide range of input possibilities. The integration of each operation with the high level code has also been tested. It is possible for the tester to determine the output of any possible test set for the high level code. This is done in the following manner. Suppose the input variables PreState and NewSignals are as indicated below in **Figure 5-17**. The value of the output variable NewState is determined from the results of the FSPT.



Sample input TMap for Update_State.

Figure 5-17 Sample Input TMap to Operation Update_State

There are two signals and two indications. The first operation called in ChkGL which check the guardline field and sets ChkGL to true if the guardline is OK' and false if it is TRIP'. In this case ChkGL is set to true. Thus the next operation called is Update_GL. Examining the test paths and the results achieved one can determine the value of the output variable State1. The operation follows test path three and results in a guardline signal of OK' for State1 (See **Tables 5-5** and **5-6**). The final operation called is Updäte_Indi. The input variables State1 and NewSignals cause a combination of test paths two and three to be executed. The first signal of path two and the second signal of path three are of interest State1 (See **Tables 5-8** and **5-9**). The resulting indications are both in the ON' state. Thus, the resulting data structure from the test case in **Figure 5-17** can be derived without executing a single test. It is shown in **Figure 5-18**.

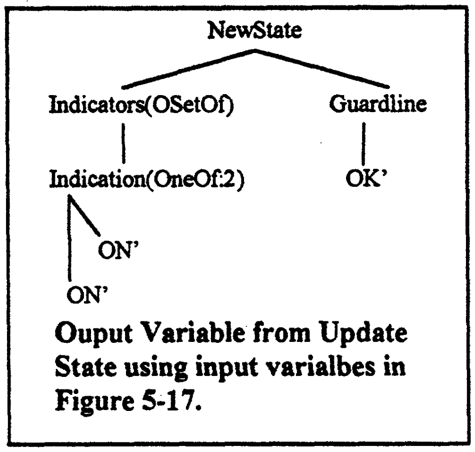

NewState
Indicators(OSetOf)
Guardline
Indication(OneOf:2)
OK'
ON'
ON'
Ouput Variable from Update State using input varialbes in Figure 5-17.

Figure 5-18 Sample Output TMap from Update_State

The FSPT of the RPS found no errors in the code. This is because there are none present. FSPT gives the tester a list of input / output sets. It is the tester's job to analyze these sets to assure that the results meet the criteria given in the specification document.

5.4 FSPT Error Location Experiment

The Reactor Protection System is tested error free in **Section 5.3**. An experiment was performed subsequently to determine the usefulness of the FSPT for the location of errors. Random errors were introduced into the RPS. These errors were capable of passing the OO1

Analyzer, meaning that they can only be discovered during program execution. The Analyzer is the compiler of the OO1 system. In all, seven errors were introduced into the FMaps. The FSPT was used in an attempt to locate and remedy these errors.

The procedure for locating the errors is no different from that used above. The code is examined and tested in a modular fashion. The results of the tests are then compared with those of the specification document. An example of testing one of the operations is given below. Operation Update_GL is listed in **Figure 5-19** and contains one error.

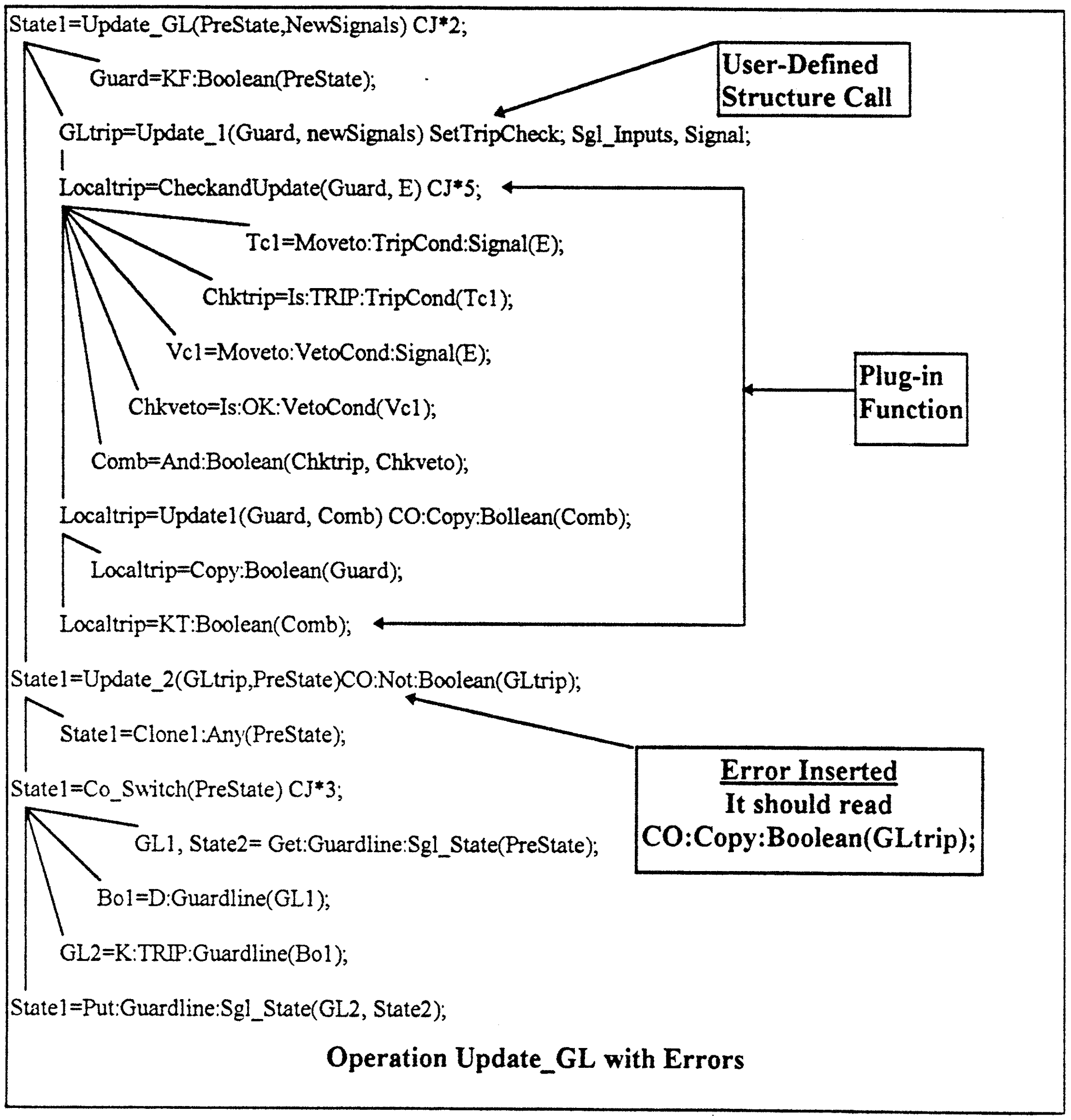
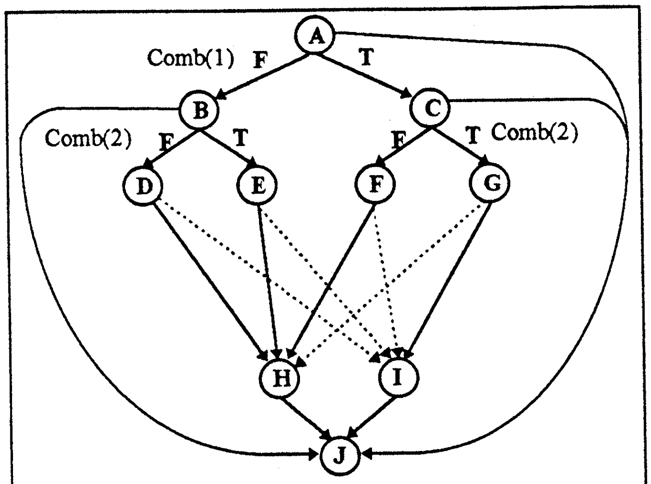

```
State1=Update_GL(PreState,NewSignals) CJ*2;
    Guard=KF:Boolean(PreState);
  GLtrip=Update_1(Guard, newSignals) SetTripCheck; Sgl_Inputs, Signal;
  Localtrip=CheckandUpdate(Guard, E) CJ*5;
                 Tc1=Moveto:TripCond:Signal(E);
              Chktrip=Is:TRIP:TripCond(Tc1);
           Vc1=Moveto:VetoCond:Signal(E);
        Chkveto=Is:OK:VetoCond(Vc1);
    Comb=And:Boolean(Chktrip, Chkveto);
  Localtrip=Update1(Guard, Comb) CO:Copy:Bollean(Comb);
    Localtrip=Copy:Boolean(Guard);
  Localtrip=KT:Boolean(Comb);
State1=Update_2(GLtrip,PreState)CO:Not:Boolean(GLtrip);
  State1=Clone1:Any(PreState);
State1=Co_Switch(PreState) CJ*3;
      GL1, State2= Get:Guardline:Sgl_State(PreState);
    Bo1=D:Guardline(GL1);
  GL2=K:TRIP:Guardline(Bo1);
State1=Put:Guardline:Sgl_State(GL2, State2);
```

Figure 5-19 Update_GL with Errors

The normal five step process is not shown here, as it is nearly identical to the one demonstated in **Section 5.3.2**. The presence of the error shown in **Figure 5-19** greatly changes the program flowgraph. For convenience, the normal convention of the true branch being to the left and the false branch to the right is reversed in the step three flowgraph shown in **Figure 5-20**.



Operation Update_GL with Errors
Step Three Flowgraph
Values of the control variable Comb which cause certain paths to be executed are noted above. Unfeasable paths which have been remove are indicated by dashed lines.

Figure 5-20 Update_GL with an Error Step 3 Flowgraph

The test cases and their results are given in the tables below.

Test Cases for Operation Update_GL

Path	TripCond(1)	VetoCond(1)	TripCond(2)	VetoCond(2)
0	TRIP'	OK'	** OK'	** VETO'
1	NULL	NULL	NULL	NULL
2	* TRIP'	* VETO'	TRIP'	OK'
3	* OK'	* OK'	** TRIP'	** VETO'
4	* TRIP'	* VETO'	NULL	NULL
5	TRIP'	OK'	NULL	NULL
6	TRIP'	OK'	TRIP'	OK'

* indicates that the set of TripCond(1) and VetoCond(1) can be anything but TRIP' and OK'.
** indicates that the set of TripCond(2) and VetoCond(2) can be anything but TRIP' and OK'.

Table 5-12 Test Cases: Update_GL with an Error

Test Results from Operation Update_GL

Path	Guard Line
0	OK'
1	TRIP'
2	OK'
3	TRIP'
4	TRIP'
5	OK'
6	OK'

Figure 5-21 Test Results: Update_GL with an Error

If one assumes that the function of operation Update_GL is to update the guardline signal, the results indicate a problem. The specification document requires that the guardline be tripped only if a signal is in a trip condition of TRIP' and has a veto condition of OK'. Otherwise, the guardline condition is to remain unchanged. In this case, the input value of the guardline is always OK'. In every case where the guardline should TRIP', it remains OK' and in every other case has a value of TRIP'. These results indicate that an error exists in the code. It is the responsibility of the tester to locate this error and remedy the situation. Once this takes place, one then tests the code again to assure that no errors remain.

The remaining six errors are found by the FSPT in each of the operations and in the high level code. The details of this process are not given in this report as the process is identical to the example given in **Section 5.3**.

5.5 Conclusion

The Feasible Structured Path Testing method is used to test the Reactor Protection System. The RPS is defined in the OO1 case tool language, and is a modular system consisting of four operations, two user-defined structures and a high level code. The FSPT with a k value of two is completed for the RPS in twenty three tests. The FSPT is a versatile method which efficiently tests code and only sacrifices completeness when looping structures are present.

6. Conclusion

6.1 Recommendations to Designers

Safety-critical software, such as that used in monitoring nuclear power reactors, must be highly reliable. In order to achieve high reliability, the software must contain no programming errors. Testing the software is a crucial in locating programming errors. By locating the most complete and practical testing method, this work reported here focused upon the use of testing techniques to improve the reliability of the software. Through the course of the work, many lessons were learned, which are applicable to software designers.

The design of software code begins with the drafting of a specification document. It is from this document that programmers create software. The manner in which the specification document is written can have a great effect upon the efficiency of the software coding process. A well prepared specification document greatly improves the reliability of the resulting code. When writing a specification, a modular and detailed approach should be taken. The programmer determines the sections of the code and their purposes directly from the specification document. The document should include lists of every data structure used in the code and precisely describe how they are used to achieve the program's goal. Including specifications for each modular code section allows the tester to test each module incrementally. Testing each module separately improves the chance of locating errors by allowing a more in-depth testing strategy derived from eliminating inter-modular error propagation..

Throughout the entire software development process, a formal development method should be used. In this study, the development before the fact (DBF) method, and its supporting tool the

OO1 Case Tool language was used. The formal method removes many of the inconsistencies introduced into programs by human errors. By using completely reliable building blocks, and feedback between design phases, the formal method is far superior to the ad hoc methods, such as have been commonly used in the past.

There are many design procedures, specific to the software code itself, which aid the Feasible Structured Path Testing process. The FSPT sacrifices complete testing in the presence of looping structures. Since this is true, the designer is advised to only include loops in programs when they are crucial. A program without loops is, in general, completely testable. Removing loops from a program greatly improves that programs testability.

The FSPT can also be used as a feedback tool to the designer. When an unfeasible path is found on the flowgraph, the designer should consider removing this path. Eliminating unfeasible paths, decreases testing time and makes for a more accurate test result.

With the FSPT, the detection of errors depends totally upon the presence of an oracle. Recall that an oracle is a database of a complete set of input and corresponding output combinations. The presence of an oracle is often assumed. In this report, the oracle was developed directly from the specification document. The specification document must be correct in order for the program to be correct. Should the specification contain an error, this error will be passed into the program. The use of a formal design method aids greatly in reducing the likelihood of such an occurrence.

6.2 Future Work

Future work with the FSPT should include an attempt to automate the testing process. This could be done for any programming language. A further study into the feedback which the FSPT gives the designer should be completed. The use of new programming styles which aid in the testing process should be examined. Since, the possibility of a coincidentally correct test case is always present, a false positive could cause an error to go unnoticed through the entire testing process. The possibility of coincidentally correct test results is high when the output variable only exists in a few states. An example is a complex program with a Boolean variable output. Such a program would require the tester to have a method of tracing the path of execution for each test case. If the execution path differs from the intended one, an error exists and should be repaired. In future work, a method for tracing the path of execution is necessary to prevent coincidental correctness. Finally, a study should be completed which examines a method of measuring the reliability of software code when a complete test cannot be achieved, which is often the case. This measure could be similar to those used in determining the reliability of safety critical hardware components.

6.3 Review

The goal of the study presented here is the identification of a complete yet practical software testing method. This method is intended to improve the reliability of safety-critical software systems. The work began with a review of previous work and a literature search which revealed two complete testing methods. Only one of the methods, complete path testing, was feasible. However, in the presence of looping structures, the number of test cases needed to achieve complete path testing was unfeasibly large. The great many test cases which are sometimes needed cause the technique to be unfeasible for many program structures. A new testing method, known as the Feasible Structured Path Testing method (FSPT) was developed in this study. The FSPT is five step formal testing method based upon complete path testing. The FSPT limits the number of test cases in the presence of loops to a reasonable number. The FSPT was used on a piece of code written in the OO1 specification language to search for errors. The results of the testing are excellent. Every programming error was located by the FSPT. The FSPT improves the correctness of software code examined in this work.

References

Arno94	Arno, Verification and Validation of Safety Related Software, Report No. MIT-ANP-TR-022, Massachusetts Institute of Technology, June 1994.
Brown92	B. Brown, R. F. Roggio, J. H. Cross II, and C. L. McCreary, "An Automated Oracle for Software Testing," IEEE Transactions on Reliability, Vol. 41, No. 2, pp. 272-280, June 1992.
Beiz90	Beizer, Software Testing Techniques, Van Nostrand Reinhold, New York, N.Y., 1990.
Bloo86	R.E. Bloomfield, P.K.D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software," IEEE Tras. on Software Eng., Vol. se-12, No. 9, pp. 988-993, September 1986.
Ntaf88	C. Ntafos, "A Comparison of Some Structural Testing Strategies," IEEE Transactions of Software Engineering, Vol. 14, No. 6 pp. 868-873, June 1988.
Howd75a	E. Howden, "Completeness Criteria for Testing Elementary Program Functions," IEEE Trans. Comput., vol. C-24, No. 5, pp. 544-559, May 1975.
Howd75b	E. Howden, "Methodology for the Generation of Program Test Data," IEEE Transactions of Software Engineering, Vol. c-24, No. 5 pp. 554-559, May 1975.
Howd77	E. Howden, "Symbolic testing-design techniques, costs and effectiveness", NTIS PB-268517, May 1977
Poor94	E. Poorman, On the Complete Testing of Simple, Safety-Related Software, Report No. MIT-ANP-TR-019, Massachusetts Institute of Technology, February, 1994.
Frank88	G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," IEEE Transactions on Software Engineering, Vol. 14, No. 10, pp. 1483-1498, October 1988.
Frank93	G. Frankl and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," IEEE Transactions on Software Engineering, Vol. 19, No. 8, pp 774-787, August 1993.
Hami92	M.H. Hamilton, Increasing Quality and Productivity with a "Development Before The Fact" Paradigm, Hamilton Technology Inc. Document, 1992.
Hals77	H. Halstead, Elements of Software Science, Elsevier, North-Holand, 1977.

McCa76 J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. se-2, No.4, pp. 308-320, December 1976.

McCa91 J. McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," National Bureau of Standards special publication 500-99, December 1991.

Myers79 J. Myers, The Art of Software Testing, John Wiley & Sons, New York, N. Y., 1979.

Rich94 J. Richardson, "TAOS: Testing with Analysis and Oracle Support," ACM 0-89791-683-2, 1994.

Weyu93 J. Weyuker, "More Experience with Data Flow Testing," IEEE Transactions on Software Engineering, Vol. 19, No. 9, pp. 912-919 ,September 1993.

Biem89 M. Bieman and J. L. Schultz, "Estimating the Number of Test Cases Required to Satisfy the All-du-Paths Testing Criterion," ACM 089791-342-6, 1989.

Hoff91 M. Hoffman and Paul Strooper, "Automated Module Testing in Proglog," IEEE Transactions on Software Engineering, Vol. 17, No. 9, pp. 934-943, September 1991.

Weiss88 N. Weiss, and E. J. Weyuker, "An Extended Domain-Based Model of Software Reliability," IEEE Transactions on Software Engineering, Vol. 14, No. 10, pp. 1512-1524, October 1988.

Ouya95 Ouyang, "An Integrated Formal Approach for Developing Reliable Software of Safety-Critical System", Report No. MIT-ANP-TR-035, Massachusetts Institute of Technology, August, 1995.

Pete94 Peters, "Generating a Test Oracle from Program Documentation," ACM 0-89791-683-2, 1994.

Roper93 R. Roper and A. R. B. A. Rahim, "Software Testing Using Analysis and Design Based Techniques," Software Testing, Verification on Reliability, Vol. 3, pp. 165-179, 1993.

Rapps85 Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," IEEE Transactions of Software Engineering, Vol. SE-11, No. 4 , pp. 367-375, April 1985.

6213-66