INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.
2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.
3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.
4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.
5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

**University
Microfilms
International**

300 N. Zeeb Road
Ann Arbor, MI 48106

8403523

Haley, Allen Woodward, Jr.

DEVELOPMENT AND APPLICATION OF A WHITE BOX APPROACH TO
INTEGRATION TESTING

The Ohio State University

Ph.D. 1983

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

DEVELOPMENT AND APPLICATION OF A
WHITE BOX APPROACH TO INTEGRATION TESTING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Allen Woodward Haley Jr., B.S., M.S.

* * * * *

The Ohio State University

1983

Reading Committee:

Assoc. Prof. Stuart H. Zweben

Ass't. Prof. John S. Gourlay

Assoc. Prof. William F. Odgen

Approved By:

Adviser
Department of Computer
and Information Science

© Copyright by
Allen W. Haley Jr.
1983

ACKNOWLEDGEMENTS

The author would like express his gratitude to his advisor for his support and advice. Special thanks also go to Steve Zeil for his timely suggestions, without which much of this work would have been impossible. Finally, and most importantly, my thanks go to my wife Lynne, whose support and tolerance was vital to the completion of this work.

VITA

February 5, 1955 Born - York, Pa.

1977 B.S. in Physics and
Computer Science, Iowa
State University, Ames,
Iowa

1977-1978. Teaching Associate,
Department of Computer and
Information Science, The
Ohio State University,
Columbus, Ohio

1978 M.S. in Computer and
Information Science, The
Ohio State University,
Columbus, Ohio

1978-1982. Research Associate,
Department of Computer and
Information Science, The
Ohio State University,
Columbus, Ohio

1982-1983. Instructor, Department of
Computer and Information
Sciences, University of
Delaware, Newark, Delaware

PUBLICATIONS

"An Approach to Reliable Integration Testing", Technical
Report TR-81-5, Computer and Information Science Research
Center, The Ohio State University, May 1981; co-author is
S. Zweben.

"Module Integration Testing", Computer Program Testing, Edited by B. Chandrasekaran, S. Radicchi, North-Holland, New York, 1981, 289-299; co-author is S. Zweben.

"Development and Application of a White Box Approach to Integration Testing", Proc. 1982 Workshop on Effectiveness of Testing and Proving Methods, Catalina Island, California, May 1982; co-author is S. Zweben.

FIELDS OF STUDY

Major Field:

Software Engineering. Professor Lee White and
Professor Stuart Zweben

Minor Fields:

Programming Languages. Professor Jay Ramanathan

Computer Architecture. Professor Bruce Weide

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	ii
VITA	iii
LIST OF TABLES	vii
LIST OF FIGURES.	viii
 Chapter	
I. Introduction.	1
I.1 Validation and the Software Life Cycle . .	3
I.2 Validation in the Implementation Phase . .	6
I.3 Program Modularity and the Testing Process	8
II. Program Modularity and White Box Testing . . .	14
II.1 Advantages of Modularity in Testing . . .	21
II.2 Integration Testing Errors.	23
II.3 Detecting Integration Errors.	28
III. Path Selection to Detect Integration Errors .	32
III.1 A Model of Linear Modules.	35
III.2 Detection of Integration Time Domain Errors.	43
III.3 Detection of Integration Time Computation Errors	73
III.4 Combined Integration Testing	97

IV. Integration Testing with Nonlinear Predicates and Computations	100
IV.1 A Model of Modules with Nonlinearity. . .	102
IV.2 Integration Domain Errors with Nonlinear Predicates.	106
IV.3 Integration Computation Errors with Nonlinear Computations.	115
IV.4 Combined Integration Testing for Nonlinear Predicates and Computations . .	119
V. Fundamental Limits of White Box Integration Testing	122
V.1 Integration Path Infeasibility	124
V.2 Predicate Redundancy	132
VI. An Example of the Integration Testing Strategy	141
VI.1 An Example of Integration Domain Error Detection	163
VI.2 Affects of the Fundamental Limits on the Testing of the Example Program.	179
VI.3 An Example of Integration Computation Error Detection	182
VII. Summary and Conclusions	191
APPENDIX A	199
BIBLIOGRAPHY	201

LIST OF TABLES

Table	Page
6.1 List of Input and Output Variables for CALCULATE-FED-TAXES.	157
6.2 Breakdown of Input and Output Variables for CALCULATE-FED-TAXES.	161
6.3 Domain Error Integration Test Set.	174
6.4 Domain Error Paths Examined.	175
6.5 Computation Error Integration Test Set	188
6.6 Computation Error Paths Examined	189

LIST OF FIGURES

Figure	Page
1.1 The Software Life cycle.	3
1.2 Possible Integration Testing Sequences	10
2.1 Example of Statement Coverage Deficiencies	15
2.2 Example of Branch Coverage Deficiencies.	16
2.3 Program with $8 \cdot 10^8$ Paths	20
2.4 Integration of Subprograms with 3 and 4 Paths Respectively	22
2.5 Program Containing a Computation Used Only in a Predicate of a Previously Tested Module.	26
2.6 Module Which Transmits a Program Computation Error.	27
3.1 Sample Module for $m=4$, $n=3$ and $q=2$	37
3.2 Sample Module for $m=3$, $n=3$ and $q=1$	61
3.3 Sample Module for $m=3$, $n=3$ and $q=2$	77
3.4 Sample Module for $m=4$, $n=3$ and $q=2$	81
3.5 Sample Program and Module for Computation Errors	83
3.6 Sample Program and Module with Error Losses.	86
4.1 Nonlinear Module for $m=2$, $n=3$ and $q=2$	107
5.1 Module with Four Paths	125
5.2 Program Calling Module MODULE8	127
5.3 Module with Four Inputs and One Output	129

5.4	Program Calling MODULE9.	130
5.5	Program with Superseding Predicate	134
5.6	Domain Structure of the Incorrect Program Path .	136
5.7	Domain Structure of the Correct Program Path . .	137
6.1	Flow of Control in Highest Level Module.	142
6.2	ADD-OTHER-EARNINGS Module.	144
6.3	CALCULATE-EARNINGS Module.	145
6.4	CALCULATE-FED-TAXES Module	147
6.5	CALCULATE-SIT Module	148
6.6	CALCULATE-STATE-TAXES Module	149
6.7	CALCULATE-CITY-TAXES Module.	150
6.8	CALCULATE-TAXES Module	151
6.9	CALCULATE-DEDUCTIONS Module.	152
6.10	CALCULATE-DEDUCTIONS-A Module.	154
6.11	CALCULATE-EARNINGS Module Integration Paths. . .	177

I. Introduction

The development of computer software is a labor intensive task, and therefore, an inherently error-prone process. While the cost of hardware has been dropping, and the processes for developing that hardware have been improving dramatically, the techniques for developing software systems have not kept pace with the increasing complexity of the tasks the software is designed to perform. Unless better methods are developed for creating and maintaining these software systems they will become a major obstacle to further progress.

In light of the above developments, interest in what is generally termed "Software Engineering" has grown considerably. The discipline has been described as "the practical application of scientific knowledge in the design and construction of computer programs, and the associated documentation required to develop, operate, and maintain them" [Boehm 76]. An important part of this

process concerns the problem of software validation. Software validation is the process of assuring that a particular software system performs according to the specifications of its requirements. This process can take many forms. For example, in an interactive system there might be a requirement that the system response time be no greater than 1 second. While this type of requirement is relatively easy to express it could be very difficult to verify for all possible situations. In many cases the requirements for a software system might not be as easy to express. In order to verify that a software system performs without error, it is necessary to have a set of specifications that precisely indicate the correct system action for any situation. However current practice in writing specifications usually produces an informal document that is incapable of providing the necessary information.

Though software is and will continue to be error-prone, the practiced methods for assuring software quality are largely ad hoc. This leaves considerable room for the development of methods to help systematize the process of software validation.

I.1. Validation and the Software Life Cycle

Computer software follows a "life cycle" during its development and use [Boehm 76, Zweben 81]. The life cycle has a number of identifiable parts (Figure 1.1), starting with the specification of the requirements for the software project. The next phase is the design phase, where the solution to the requirements is proposed and refined. The design phase is frequently broken into two parts, the preliminary design and the detailed design phases. Following design the system implementation phase occurs. This phase involves the actual coding of the design into a machine processable form. Finally, the software enters a maintenance phase, where problems with the delivered product are corrected and enhancements to the product are made.

Figure 1.1
The Software Life Cycle

The cycle part of the software life cycle comes from the validation steps that occur at each phase. Before the software system can proceed from one phase to the next it must first be validated to ensure that the (partially developed) software is "correct". If a problem is found the particular phase might need to be repeated, or if the problem is more serious, it might be necessary to backtrack to a previous phase to correct the problem.

The method of validation used depends on the phase that is being validated. The methods used during the requirements and design phases are largely ad hoc, and are limited by informality in the requirements and design documents. This explains why validation is typically performed through methods such as design and specification reviews. Little can be said about the effectiveness of these methods due to their lack of a formal basis. However it has been estimated that 60% of the errors made in the implementation and maintenance phases require a change in the design of the system [Boehm 76]. This tends to indicate that the validation methods used in the initial phases are not very effective in locating errors. Hopefully with the development of more formal methods for performing the requirement and design phases, more formal

methods for their validation will also be developed.

Since the implementation phase produces a formal output (the program code), the possibilities for a formal validation method are greater than for the other phases. In fact, much of the software validation research effort has been directed at the implementation phase. Even with this effort, the current state of validation as practiced in this phase tends to be nearly as ad hoc as in the other phases. This results from a lack of formal methods that can be completed with a "reasonable" effort (the problem of the complexity of existing methods will be discussed in a later section). Frequently the problem of when validation is completed (i.e. the software is deemed to be correct, or as near correct as possible) is decided by the time and budget constraints of the group performing the validation instead of by a formal measure of completeness. In addition, most implementation validation is a tedious process which is in itself error-prone. To avoid the inherent problems of a manual procedure the process of implementation validation needs to be automated.

The goal of this research is to propose a formal method of validation that can be used in conjunction with other methods during the implementation phase of the

software life cycle. This method will aid in the solution of many of the above problems.

I.2. Validation in the Implementation Phase

The majority of efforts at improving the validation of software in the implementation phase can be classified into two categories. The first category includes methods which attempt to obtain a formal proof of correctness through the use of mathematical proof techniques. The applicability of these techniques have been limited by the mathematical rigor required, and the lack of formal specifications from the design phase against which the proofs can be made. These problems, along with the large amount of work required to form a correctness proof have led to resistance from both programmers and researchers in software engineering [DeMillo 79].

The second category includes methods which demonstrate correctness by running the program with a limited selection of input points. This is the most widely accepted and practiced method of validating computer software. However, the current state of testing practice leaves room for considerable improvement. The test

selection problem has been proven to have no general solution [Howden 76]. That is, the general problem of generating a finite set of test data to determine the correctness of a program is known to be undecidable. However, many researchers have avoided this theoretical limit by restricting the goals of their particular testing strategies [White 80, Zeil 81], say by limiting the types of errors the strategy attempts to detect. If the set of errors that a strategy is capable of detecting is sufficiently large to allow for the detection of the "most commonly occurring" errors then the strategy, while not perfect, may still be helpful for testing programs.

Program testing techniques can be classified in many ways. One classification is that of "black box" vs "white box" testing. In black box testing, test data are selected according to the purpose of the program (as expressed, say by a specification), independent of the manner in which the program is actually coded. Such approaches are described in [Howden 76] and [Gourlay 81]. Unfortunately, the insights needed to develop these ideas into an easily applied testing technique are thus far beyond the state of the art. Again, the lack of a formal specification or design from the early stages of the life cycle prevent the widespread use of these types of methods, and hinder the

automation of the process.

White box testing, on the other hand, makes use of properties of the source code to guide the testing process. While such techniques can (and have been) automated, they tend to suffer from an inability to provide the tester with the information necessary to make a formal statement about the adequacy of the testing [Howden 76, Westley 79]. The lack of formality in white box testing strategies is due in part to the complexity of the methods when completely applied to even the simplest programs. Because of this complexity white box strategies often are restricted to the detection of limited classes of errors. If the complexity of white box testing strategies could be reduced sufficiently, it would allow their application to more reasonably sized programs, and would make a significant step towards a solution to the validation of the implementation phase of software development.

1.3. Program Modularity and the Testing Process

A primary limitation of many white box testing strategies is that they deal with a program as a single

large block of code. In fact, programs are often composed of many small, and relatively simple modules. By ignoring a program's inherent modularity, the testing process can be unduly complicated. This occurs because a large, complex program might be too large for a particular strategy to handle, while the strategy might have easily been applied to the simpler modules that make up the program. In addition, if a testing strategy only deals with a program as a single unit, its application would have to be delayed until the end of the implementation phase instead of occurring in parallel with the implementation.

There are basically two approaches to testing a set of modules which form the overall structure of a system. These are referred to as top down and bottom up testing respectively. In bottom up testing, individual modules are first tested in isolation from one another using their own sets of test data. When groups of related modules have been validated, they are integrated into a higher level unit (subsystem) which is then tested. The subsystem tests in general require new test data, since different inputs and outputs are involved at the higher level. Larger and larger subsystems are combined until eventually the entire system is tested as a unit. Figure 1.2 illustrates a

system configuration and a possible sequence of tests (1 through 7) using a bottom up technique. The primary disadvantages of this form of testing are the variety of test data required at each level, and the increasing complexity of the subsystems as the integration proceeds.

Top down testing, on the other hand, involves starting with the highest level component of the system and proceeding to the next lower level, etc. Using Figure 1.1 as a model once again, the "higher level" test should ideally determine that A functions correctly, knowing only the abstract purposes of B, C, and D, so that when B, C, and D are eventually implemented, it is necessary only to show that they achieve their abstract purpose (that is, theoretically no integration testing is required).

Figure 1.2
Possible Integration Testing Sequences

However, our ability to select appropriate tests of A based on these abstractions and to certify the correctness of A's output on these tests (which, after all, requires output from abstract, as yet unwritten modules), is quite limited. We are more likely when testing A to first write stubs for B, C, and D which do nothing more than indicate that the second level modules have in fact been called, and later embellish B, C, and D so that they can produce correct output for some very small, well known class of possible inputs. While this method helps identify the appropriateness of the invocation of the lower level modules and can speed up the completion of a preliminary version of a complex system, it tends to mask the subtle interrelationships between the components until all are completely developed and an attempt is made to have them work as a unit.

Therefore, one can say that even if a top down testing philosophy is attempted, integration testing will be necessary after the lowest level modules have been completed. That is, some mixture of top down and bottom up testing is probable.

To date, testing research hasn't focused on the problems of integration testing, instead dealing only with

the unit testing problem. While this work is both useful and necessary, by not incorporating a program's modularity into the testing process the methods appear limited to the testing of unreasonably simple programs. This thesis will propose a strategy for making use of the inherent modularity of a program in the testing process. This method of integration testing when used in conjunction with a formal method of white box testing will reduce the complexity of applying the testing strategy to the complete program.

In the next chapter we will examine in greater detail some of the problems of white box testing strategies, and some possible solutions to those the problems. In addition, some of the problems and solutions to using a program's modularity in white box testing will be discussed.

Chapter III will present a formal model of a module, which will be used to examine one possible solution to the integration testing problem. A formal method will be presented to choose paths in a module for integration testing, and an upper bound for the number of paths that need be examined during integration testing will be given.

The model of Chapter III will place certain restrictions on modules in order to simplify the techniques presented. Chapter IV will again deal with this model, and will relax the restrictions imposed in Chapter III. In Chapter V, some of the fundamental limits to the error detection capabilities of the integration testing strategy will be discussed.

In Chapter VI, the integration testing strategy will be applied to an actual production program. This will demonstrate both the method and the potential testing complexity reductions. In addition the fundamental limits introduced in Chapter V will again be examined in relation to their effect on the potential testing reliability of the example.

II. Program Modularity and White Box Testing

Various white box testing strategies have been proposed by different researchers. These strategies involve selecting test data to execute certain elements of the program structure, and can generally be classified into one of the following "coverage" approaches:

1. Statement Coverage.
2. Branch Coverage.
3. Path Coverage.

Statement coverage is the simplest of the coverage approaches. It requires that test data be selected such that every statement in the program be executed at least once. While it is clear that failure to test each program statement will prevent the detection of errors in the statements not tested, testing all statements doesn't ensure detection of certain errors. Consider the program segment in Figure 2.1. It is possible to execute every

```
      .  
      .  
      .  
    IF (condition)  
      THEN X = X + 1  
      .  
      .  
      .
```

Figure 2.1
Example of Statement Coverage Deficiencies

statement in this program without ever having the condition evaluate to "false". If this is the case then we might never know if the null else clause is correct.

The second coverage approach (branch coverage) attempts to to correct the deficiencies of statement coverage by requiring that every possible branch from each conditional statement be executed. This means that the true branch and false branch of every predicate must be exercised at some point in the testing process. While this method solves some of the problems of statement coverage, branch coverage also fails to detect certain potential errors. Consider the program in Figure 2.2. In this program branch coverage testing could select a test which executes the THEN branch of each predicate, followed by another test to exercise the ELSE branch of each

predicate.

Even though every branch from each predicate has been tested, the THEN branch of the first predicate is never tested in combination with the ELSE branch of the second predicate. It is therefore possible to not detect the divide by zero error that can occur on the ELSE branch of the second predicate.

It would seem that to avoid the problems of statement and branch coverage it is necessary to require that every path in a program be tested. While this method also has certain deficiencies (such as a potentially infinite number of paths), it is considerably more powerful in its ability to detect errors. Many of the more powerful

```

      .
      .
      .
IF X=1
  THEN X=X-1
  ELSE X=2**X
IF Y<0
  THEN Z=Y*X
  ELSE Z=Y/X
      .
      .
      .

```

Figure 2.2
Example of Branch Coverage Deficiencies

strategies have been based on a path coverage approach. The process of a path testing strategy (a white box testing strategy based on path coverage) can be broken into a two step procedure:

1. Select the path to be tested.
2. Generate test data to test that path.

In the past, much of the research effort in program testing has been directed at the second step of the above procedure (how to select test data for a single path). This has resulted in such strategies as DOMAIN testing [White 80], and the symbolic execution system developed by Clarke [Clarke 76]. If a reliable¹ method is used to select the test data for each path, and that strategy is applied to all of the paths in the program, a path testing strategy is capable of detecting a wide class of potential

¹ The definition of the term "Reliable" used here comes from Howden [Howden 76]. A set of test data T is defined to be reliable for a program P if the test set T reveals an error if P is incorrect. In Howden's definition he refers to reliability with respect to the program as a whole. However the definition remains valid in the context of the testing a single program path. A reliable testing strategy is one that produces reliable test data for the programs (or paths) to which it is applied. Therefore a reliable strategy applied to a path selects data which will reveal an error in the code along that path if there is such an error.

errors. However, certain types of errors might still be undetectable even when a reliable path strategy is used to test all paths through a program (i.e. a reliable path strategy isn't necessarily a reliable strategy for testing the entire program even when applied to all paths).

Two cases where this type of problem can occur have been identified by White in the DOMAIN testing strategy [White 80]. The first problem has been referred to as the missing path problem. A missing path usually occurs when a particular condition or special case was left out of the code for a program. Since the error is due to missing code, there is in general no way to detect the error by examining the program's structure. Missing paths are an example of where a black box strategy might be used to supplement the error detection capabilities of the white box strategy [Gourlay 81, Weyuker 80].

The second problem is that of path infeasibility. An infeasible path occurs when there is no input to the program which will follow that path. Clearly the paths in a program which are infeasible needn't be tested. However the problem is in determining whether a particular path is feasible. In general the problem of determining the feasibility of a path has been shown to be undecidable

[Weyuker 79].

Despite the above problems, path coverage approaches are capable of detecting a wide class of errors for a given path. The major problem with path coverage approaches is the extremely large number of paths that can occur even in simple programs. Consider the program segment in Figure 2.3.

The program performs a simple linear search of an array of length 20. However the program has over $8 \cdot 10^8$ paths. It is fairly easy to see how a program of reasonable size would have too many paths for even the simplest path testing strategy to test all of them. Until recently path testing strategies have ignored the path complexity issue, instead leaving it to the programmer (or tester) to select the paths to test. A formal method of selecting paths, that adequately test a program, is necessary before path testing strategies can be practically applied to reasonably sized programs.

Zeil has proposed a method to determine if a particular path leading to a predicate or computation is capable of detecting any errors not detectable along other paths [Zeil 81]. This strategy has the effect of placing

```

      .
      .
      .
/* Program segment to perform a linear search*/
/* of an array with 20 elements. The array is*/
/* assumed to be sorted in ascending order. */
/* TEST - The value being searched for      */
/* FOUND - Boolean indicating the success of */
/*        of the search                      */
/* COUNTER - Position in the array of the    */
/*        of the value being searched for */

FOUND := FALSE
COUNTER := 1
DO WHILE (COUNTER < 21 AND FOUND = FALSE)
  IF ARRAY(COUNTER) < TEST
    THEN COUNTER := COUNTER + 1
  ELSE IF ARRAY(COUNTER) > TEST
    THEN COUNTER := 21
  ELSE FOUND := TRUE
END WHILE

      .
      .
      .

```

Figure 2.3
Program With 8×10^8 Paths

an upper bound on the number of paths that need be tested in a program, with this upper bound being dependent on the complexity of the computations and predicates in the program, and independent of the total number of paths in the program. Zeil's work, while very useful, has one major limitation. There is no guarantee that the subset of paths that need be tested can be found without examining all the paths in the program.

II.1. Advantages of Modularity in Testing

In the course of developing the solution to a large, complex problem, software engineering practice dictates breaking the problem into subdivisions which abstract interesting aspects of the problem. These subdivisions might then be refined, implemented, and tested independently. Only then would the subsystems be integrated to form a complete working solution to the problem. The potential advantages of applying testing to these same subdivisions can be substantial. For example consider a program P consisting of subprogram P_1 containing m paths followed by subprogram P_2 containing n paths. The integrated program can have a total of $m \cdot n$ paths (see Figure 2.4 for $m=3$ and $n=4$), since any of the m paths in P_1 can be followed by any of the n paths in P_2 .

If P_1 and P_2 have been tested separately, it would be desirable if the correctness information obtained in unit testing P_1 and P_2 could be used in validating P . If the individual modules² do not contain a large number of paths, it may be possible to test all possible paths in each module. If the additional testing required at

Figure 2.4
Integration of Subprograms with 3 and 4 paths, respectively

² The notion of a module has been characterized in many different ways, and several authors have proposed criteria for what constitutes a "good module" [see e.g. Parnas 72, Yourdon 79, Stevens 74]. It will suit our purposes to allow a module to be a single entry, single exit block of code which can contain an arbitrary amount of internal control structure. For simplicity, we may represent modules as subroutines, with the understanding that the ideas presented herein are not meant to be restricted to this form of module structure.

integration time was negligible compared to the unit testing overhead (for example, if we could ignore the internal control structure of a tested module when integrating it), the result would be a reduction of the magnitude of the testing problem from $O(m*n)$ to $O(m+n)$. While this represents in some sense an ideal situation, it is clear that with such a potential for complexity reduction, even a less than ideal solution might represent a considerable improvement, and provide a substantial degree of practicality.

Our research will focus on the integration testing of independently tested modules. From the previous discussion, we note that this approach has the potential for (1) reducing the total path testing complexity, and (2) making the testing process conform more closely to the way programs are developed.

II.2. Integration Testing Errors

In order to be able to characterize the effectiveness of any testing approach, it is necessary to identify those errors which are of interest to the strategy under consideration. Any finite testing strategy is known to be

faced with certain inherent limitations as to the errors it is capable of detecting. For white box testing, which this research is concerned with, we have already identified the problems of infeasible paths and missing paths. An additional problem that can be encountered is that of "coincidental correctness". This problem arises when the program under examination happens to produce the same results as the (different) desired program on the set of data used for testing. Thus, a statement such as $X=X+2$ cannot be differentiated from $X=X*2$ if the only test data chosen results in $X=2$ on entry to the statement. For this simple case coincidental correctness can be avoided by choosing a test value of $X=3$. However, the correct and incorrect statements could be such that they evaluate the same for any test data chosen by the strategy, and only differ on some test point not chosen. Coincidental correctness is therefore another fundamental limitation of any finite testing strategy.

Admitting that errors due to coincidental correctness and missing paths may go undetected, the next problem is to try to classify those kinds of errors that we might hope to detect. One proposal, due to Howden [Howden 76], distinguishes between domain errors and computation errors. A domain error occurs when a specific input

follows the wrong path due to an error in the control flow of the program. A computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables. While this classification scheme has practical limits in that it may be possible to interpret a particular error in the results of a program as either a domain or a computation error, the scheme has been used successfully by other researchers in program testing [White 80 , Zeil 81].

The notion of domain and computation errors turns out to be useful in characterizing certain types of integration problems. For example, consider a module M which has been thoroughly validated, say by some "Hypothetical Testing Strategy", so that it is free of all domain and computation errors. Module M is to be integrated into program P. Assume that P has some computation whose result (call it C) is used in some predicate of M but is not used anywhere else in the program (see Figure 2.5).

Now suppose that the correct computation in P should have set C to I_{p+1} . In validating M, we may have ensured that M produces the correct output no matter which branch

of the IF statement is taken, but P will still produce the wrong output if the initial value I_p is such that $3 \leq I_p < 4$. However, if we do not happen to choose a value of I_p in this range we will not catch the error in the computation statement. Notice that, from the point of view of the program P, there is only one path to consider (READ I_p ; $C = I_p$; CALL M (...); $O_p = O_m$; PRINT O_p) if we ignore the control structure of the module M at integration time and deal only with P's structure. Yet this example shows that we must do more than just select a couple of values of I_p and examine the resulting values of O_p . In this case, if we were to analyze the integrated program including the module's control structure, we would notice that the program contains a domain error, since values of I_p in the range $3 \leq I_p < 4$ follow the wrong path. We will refer to this type of error as an Integration Domain Error.

Figure 2.5
Program Containing a Computation Used Only
in a Predicate of a Previously Tested Module

Computation errors cause another problem in ignoring the validated module's control structure at integration time. Assume that the program contains an incorrect computation whose result is passed to the validated module. Further assume that the only use of this result is by some computation in the validated module. As an example, suppose P is the same as in Figure 2.5, but M is changed as in Figure 2.6.

Assume once again that the computation in P should set C equal to I_{p+1} instead of I_p . If integration test data were chosen which never exercised the true branch of the condition in M, then the resulting value of O_m would always be 2 and the computation error in P would go undetected by simply examining the output of the program. We will refer to this type of error as an Integration Computation Error.

```

      .
      :
      .
M {  IF (condition)
    THEN  $O_m = C$ 
    ELSE  $O_m = 2$ 

```

Figure 2.6
Module Which Transmits a Program Computation Error

These two examples have elements in common. In both cases there is an error in the code preceding the call to the validated module. The error causes one of the module's inputs to have an incorrect (not invalid) value for the particular input values to the program. It is possible for the error in the module's input to not be reflected as an error in the module's output, since transmission of the error to an output may be dependent upon the particular path chosen through the module. It is therefore clear that, when integrating a previously validated module, one needs to know more than just that the module is correct. If information relevant to the module's internal structure is ignored, it is possible for both domain and computation errors in the integrated program to go undetected. Therefore it is natural to ask at this stage "What, in addition to knowing that the module is correct, will allow effective integration testing to be done?".

II.3. Detecting Integration Errors

Two approaches to answering the question posed at the end of the previous section are suggested by the examples presented in that section. Since our goal is to detect

errors in the module's input, we could simply require that all input values to the module be output (along with the normal output of the calling program). This technique is not new, as programmers often print out values of intermediate/temporary variables. However it is often hard to know whether an intermediate program value is correct, as the specifications for a program may only contain information about the final program outputs. More likely, the programmer is only interested in or capable of examining the final outputs of the (calling) program.

Therefore we consider a second approach. It would appear that the chief problems presented in the previous section are 1) an error in the calling program causes a predicate in the module to have a shifted³ interpretation that isn't detected due to a failure to retest the

³ The term predicate interpretation refers to a predicate in which each program variable is replaced by its symbolic value in terms of input variables. Since the symbolic value of a program variable depends on the particular way in which that variable has been assigned, an individual predicate might have many different predicate interpretations depending on the assignments which exist on the various paths leading to the predicate. The notion of a predicate interpretation shifting has been used by Cohen and White [White 80] to define in a geometric way a predicate interpretation error (domain error). A shifted predicate interpretation has a different symbolic value from the correct predicate interpretation in such a way that some input values to the program follow the wrong branch at the shifted predicate.

particular path in the module that contains the predicate interpretation (an integration domain error), or 2) an error in the calling program causes an error in the module's input that isn't detected due to a failure to retest a path in the module that will transmit the error to the module's outputs (and hence to the program's outputs) (an integration computation error). The solution, therefore, seems to be a matter of "retesting", during integration testing, a set of paths through the module which are sensitive to these problems. We will refer to this set as the Integration Test Set for the module. The integration test set should meet two important criteria. First, it should be capable of detecting all of the integration domain and integration computation errors identified in the previous section. Second, it should contain as few of the module's paths as is necessary to meet the first criterion.

In order to find an integration test set for a particular module it is first necessary to be able to characterize all the possible "different" integration domain and computation errors that can occur in the module. Once this is done, it is necessary to be able to tell, given a path through the module, which of the possible integration errors the particular path will

detect. In the next chapter a method for modeling modules will be introduced. Using this model a method of determining the integration test set will be given.

III. Path Selection to Detect Integration Errors

In the previous chapter we have identified particular error types that need to be considered when performing integration testing. If the goal is to detect these errors by retesting a subset of the paths through the module at integration testing time it would be helpful to first develop a formal method of characterizing the types of errors each path is capable of detecting. In this chapter a model for modules with linear predicates and computations will be given. By using the model it will be possible to examine the relative effectiveness of different paths in detecting integration errors.

The model presented here is a variation of one presented by Zeil [Zeil 81] in his work on testing sufficiency. The model has been adapted to more accurately reflect the nature of a program module and places the following restrictions on the module.

1. Restrict the module such that all input variables are assigned upon entry, and no input variables are reassigned later in the module.

2. Restrict the output variables such that all output variables are assigned at the end of the module, i.e. on a given path after the first output variable is assigned no more assignments may be made to program variables. In addition, no output variable of the module can be used or referenced within the module. (This implies that the order in which the output variables are assigned along a path does not matter.)

3. Restrict all computations and predicates to be linear with respect to the module's inputs.

The first two restrictions serve primarily to simplify the notation which follows. Since any module which doesn't conform to these restrictions can be easily rewritten to satisfy the restrictions, they are not fundamental limitations. The third restriction, though considerable, makes it much easier to model the computation sequences along a path in the module.

The approach taken with the third restriction isn't new, as many researchers have placed limits on the functional complexity of programs for testing purposes in order to overcome undecidability limitations [Chow 78, White 80, Zeil 81]. Furthermore, evidence exists to show that typical programs are quite often very restricted in the type of functional complexity they employ. Early evidence for this is provided by Knuth [Knuth 71] in which he demonstrated that addition and subtraction account for the majority of operators used in practical programs. More recently, a study by Cohen [Cohen 78] of 50 data processing programs found only 1 out of 1225 program control statements involved non-linear expressions. While this doesn't guarantee that the predicates and computations in a module will be linear with respect to the module's inputs, it does show that programs usually involve simple operators, and it isn't unreasonable to expect to find modules which satisfy the third condition. Also implied in the third condition is the lack of non-numeric data types. While this restriction is severe, in that many programs manipulate characters and other non-numeric types, the approach is a common one and can be justified by noting that internally, non-numeric data types are treated as numeric types.

However, in many cases modules might not meet the third restriction of the model. For this reason a relaxation of this third restriction will be discussed in Chapter IV.

III.1. A Model of Linear Modules

Given the three restrictions, a module can be modeled in the following manner. Suppose a program contains m input variables $I_1, \dots, I_m$, n program variables $P_1, \dots, P_n$, and q output variables $O_1, \dots, O_q$. We introduce an environment vector, $\bar{V}$, which contains the current value of all variables at some point of execution.

$$\text{i.e. } \bar{V} = \begin{bmatrix} 1 \\ \text{Value of } I_1 \\ \vdots \\ \text{Value of } I_m \\ \text{Value of } P_1 \\ \vdots \\ \text{Value of } P_n \\ \text{Value of } O_1 \\ \vdots \\ \text{Value of } O_q \end{bmatrix}$$

The 1 in this case is a position for representing all of the constants. When computations or predicates are applied to the environment vector, the constant position will allow a consistent treatment of the constant terms in those computations or predicates.

Consider the module in Figure 3.1 for $m=4$, $n=3$, and $q=2$. If on input to the module the input variables are assigned the values 1, -1, 2, and 3 respectively then the environment vector on entry to the module is given by:

$$\bar{V}_{\text{initial}} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \end{bmatrix}$$

After execution of the first statement the environment vector is:

```

SUBROUTINE MODULE1 (I1,I2,I3,I4,O1,O2)
1.  P1=I1+I2
2.  P2=2*I1-I3+I4
3.  IF P1>0
4.      THEN P3=P2+3
5.      ELSE P3=P1+P2
6.  ENDIF
7.  O1=P3
8.  O2=P1+P2+P3
9.  RETURN
10. END

```

Figure 3.1
Sample Module for m=4, n=3, and q=2

$$\bar{V}_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ 0 \\ \text{undef} \\ \text{undef} \\ \text{undef} \\ \text{undef} \end{bmatrix}$$

At the RETURN statement, the final environment vector is given by:

$$\bar{V}_{\text{final}} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 3 \\ 0 \\ 3 \\ 6 \\ 6 \\ 9 \end{bmatrix}$$

A sequence of computations along a given path can be represented as the product of the C matrices corresponding to the individual assignment statements for that path. The C matrix for the sequence of statements 1,2,4 is given by the matrix.

$$C_{4,2,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that the rows of this matrix, corresponding to the value of individual variables, are expressed in terms of the module's inputs, and not the program variables. In effect, the multiplying of the individual C matrices for a series of statements produces a symbolic execution of those statements.

Predicates in the module can be represented as a $1+m+n+q$ vector $\bar{T}$ which, when applied to the environment vector, yields a scalar which is to be compared to zero. The results of this comparison determines which path is

followed. The elements of $\bar{T}$ contain the coefficients of the constant and variables used in the predicate. From Figure 3.1 the predicate in statement 3 ($P1 \geq 0$) is represented as

$$\bar{T} = \{ \bar{0} \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ \bar{0} \}$$

In order to incorporate the idea of a predicate interpretation, a predicate in the module must be represented along with a path leading up to that predicate. Given a particular predicate in the module represented by its vector $\bar{T}$ and an environment vector $\bar{V}_{PT}$ for a path P leading up to that predicate, a predicate interpretation can be modeled as

$$0 \text{ .relop. } \bar{T} \bar{V}_{PT}$$

By substitution this expression can be written in terms of the initial environment vector $\bar{V}_0$ and the computations C_{PT} along the path P leading to the predicate yielding

$$0 \text{ .relop. } \bar{T} C_{PT} \bar{V}_0.$$

For the predicate in Figure 3.1 the predicate interpretation is composed of the following elements (Note, there is only one path leading to this predicate, and therefore only one choice for C_{PT}):

$$C_{PT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\bar{V}_0 = \begin{bmatrix} 1 \\ \text{Value of I1} \\ \text{Value of I2} \\ \text{Value of I3} \\ \text{Value of I4} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \\ \text{undefined} \end{bmatrix}$$

Thus, the interpretation of the predicate is

$$0 \leq TC_{PT} \bar{V}_0$$

or

$$0 \leq I1 + I2$$

which if evaluated for $\{I1, I2, I3, I4\} = \{1, -1, 2, 3\}$ yields⁴

$0 \leq 0$ or TRUE.

The notion of a predicate interpretation can also be represented with this model. Recall that a predicate interpretation is a predicate expressed in terms of the inputs to the module. Using the notation introduced earlier a predicate interpretation for a particular predicate and path leading to that predicate is given by the quantity $(\bar{I}C_{PT})$. From Figure 3.1 the predicate interpretation is given by

$[0, 1, 1, 0, 0, 0, 0, 0, 0, 0]$.

Using this model it will now be possible to examine the ability of individual paths to detect different integration domain and integration computation errors.

⁴ It is necessary to adopt the convention that it is permissible to multiply undefined values by zero, resulting in a value of zero. Multiplication of undefined values by a nonzero quantity will result in a value which is undefined.

III.2. Detection of Integration Time Domain Errors

We will first examine the methods by which an input error to a module can affect a predicate interpretation for a particular predicate statement in the module. By ensuring that an input error to the module causes a predicate interpretation to "shift" that input error may then be detectable using techniques such as Domain Testing. Using the three restrictions presented earlier the computations along a path leading up to a particular predicate T can be represented by

$$\bar{V}_T = C_t \dots C_1 \bar{V}_0$$

where $C_t \dots C_1$ represent the computations (in reverse order of execution) which assign program variables along the path leading up to the predicate, $\bar{V}_0$ represents the initial environment vector, and $\bar{V}_T$ represents the environment vector immediately before the execution of the predicate. For convenience we will write the above expression as

$$\bar{V}_T = C\bar{V}_0$$

where C is now a single $1+m+n+q$ by $1+m+n+q$ matrix which represents the results of all of the computations leading up to the predicate (the PT subscript for C in the previous section has been dropped to simplify the notation to follow).

If an error exists in the calling program resulting in an incorrect value being passed to the module, this can be viewed as an error in the initial environment vector. If $\bar{V}'_0$ is used to denote the incorrect initial environment then

$$\bar{V}'_0 = \bar{V}_0 + \bar{e} \quad \bar{e} \neq \bar{0}$$

where $\bar{V}_0$ is the correct initial environment vector, and $\bar{e}$ is the initial error term in the environment vector. Since the initial error can only occur in the input variables, $\bar{e}$ is restricted to be 0 in the first position, and 0 in the last $n+q$ positions, and nonzero in at least one position from 2 to $m+1$.

If this new expression is substituted into the expression for the predicate interpretation it yields

$$0 \text{ .relop. } (\bar{T}C)(\bar{V}_0 + \bar{e})$$

or

$$0 \text{ .relop. } (\bar{T}C)\bar{V}_0 + (\bar{T}C)\bar{e}.$$

In this form the predicate interpretation can be viewed as the sum of the correct interpretation and a scalar error term. An integration domain error would be detectable if the following condition holds.

$\forall \bar{e} \neq \bar{0}, \exists \bar{V}_0$ such that

$$\{(0 \text{ .relop. } (\bar{T}C)(\bar{V}_0 + \bar{e})) \neq (0 \text{ .relop. } (\bar{T}C)\bar{V}_0)\} \quad (1)$$

The above expression is simply the definition of a shifted predicate. Implied in the statement of equation (1) is the condition that the values of both $\bar{V}_0$ and $\bar{V}_0 + \bar{e}$ will follow the path to $\bar{T}$ indicated by the computation matrix C. In order for an error in the calling program to produce a nonequivalent predicate interpretation (a shifted predicate) along some path in the module, it is necessary and sufficient for the following condition to hold.

$$\forall k > 0, \forall \bar{e} \neq \bar{0}, \exists \bar{V}_0 \text{ such that} \\ (\bar{TC})(\bar{V}_0 + \bar{e}) \neq k(\bar{TC})\bar{V}_0 \quad (2)$$

Proof If for all possible values of $\bar{V}_0$

$$(\bar{TC})(\bar{V}_0 + \bar{e}) = k(\bar{TC})\bar{V}_0$$

then we can rewrite the condition for a shifted predicate as

$$\{(0 \text{ .relop. } k(\bar{TC})\bar{V}_0) \neq (0 \text{ .relop. } (\bar{TC})\bar{V}_0)\}.$$

Multiplying both sides of the erroneous predicate interpretation by $1/k$ results in the expression

$$\{(0 \text{ .relop. } (\bar{TC})\bar{V}_0) \neq (0 \text{ .relop. } (\bar{TC})\bar{V}_0)\}.$$

Clearly this isn't true, and no predicate shift will occur. In addition given that condition (2) holds for $k = 1$ then this implies that

$$(\bar{TC})\bar{e} \neq 0.$$

Given this we can always choose a value of $\bar{V}_0$ for all $\bar{e} \neq \bar{0}$ such that condition (1) also holds.

case 1: If .relop. is chosen from $\{=, \neq\}$ then

$$\forall \bar{e} \neq \bar{0} \text{ choose a } \bar{V}_0 \text{ such that } (\bar{TC})\bar{V}_0 = 0.$$

Substituting this into condition (1) results in

$$\{0 \text{ .relop. } (\bar{T}C)\bar{e}\} \neq \{0 \text{ .relop. } 0\}$$

Which must hold if $(\bar{T}C)\bar{e} \neq \bar{0}$.

case 2: If .relop. is chosen from $\{<, >, <=, >=\}$ then

$$\forall \bar{e} \neq \bar{0} \text{ choose a } \bar{v}_0 \text{ such that } \bar{v}_0 = -(1/2)\bar{e}.$$

Substituting this into condition (1) results in

$$\{0 \text{ .relop. } (1/2)(\bar{T}C)\bar{e}\} \neq \{0 \text{ .relop. } -(1/2)(\bar{T}C)\bar{e}\}$$

Which must also hold if $(\bar{T}C)\bar{e} \neq \bar{0}$. qed.

Condition (2) simply states that if a non-zero error term occurs in the input to the module, for some value of the correct input, the interpretation of the predicate under the erroneous input should not be a multiple of the interpretation under the correct input. If the predicate interpretation under the erroneous input were a multiple of the correct interpretation then the difference wouldn't be detectable as the path followed by any particular inputs wouldn't be changed.

Once again using the module in Figure 3.1, we consider the predicate $P1 \geq 0$, whose interpretation is $I1 + I2 \geq 0$. If an error in the calling program causes $I1$ and $I2$ to be modified to $I1'$ and $I2'$ in such a way that

$I1'+I2'=k(I1+I2)$ for some $k>0$, then the interpretation of this predicate, $k(I1+I2)\geq 0$, is equivalent to the original. For example, suppose the calling program had inputs X , Y , and Z and the calling program should have set

$$\begin{aligned} I1 &= X+2*Y \\ I2 &= 2*Z \end{aligned}$$

but instead erroneously set

$$\begin{aligned} I1 &= 2*X+Z \\ I2 &= 3*Z+4*Y. \end{aligned}$$

Then, in terms of the inputs to the calling program, the interpretation of the predicate $P1\geq 0$ should have been $X+2*Y+2*Z\geq 0$, but instead is $2*X+4*Y+4*Z = 2*(X+2*Y+2*Z)\geq 0$. In this case both the incorrect and correct predicate evaluate identically for any values of the triplet (X,Y,Z) .

By the restrictions placed on modules by the model, we know that a predicate can only be expressed in terms of the inputs and program variables for a module. This implies that only the first $1+m+n$ positions of $\bar{T}$ are of interest (the last q positions must be zero). In addition, due to the restrictions of the model, only parts of the C matrix are useful. A more detailed examination of the C matrix reveals that the C matrix can be considered to be 9 separate submatrices of the following form.

$$C = \begin{bmatrix} \overline{C(1,1)} & C(1,2) & \overline{C(1,3)} \\ C(2,1) & C(2,2) & C(2,3) \\ C(3,1) & C(3,2) & C(3,3) \end{bmatrix}$$

where:

$C(1,1)$ is an $1+m$ by $1+m$ matrix which describes how the inputs and constant are mapped onto the inputs and constant. By our restrictions this submatrix must be equal to the identity matrix.

$C(1,2)$ is an $1+m$ by n matrix which describes how program variables are mapped onto the inputs and constant. This submatrix must be equal to the zero matrix by our restrictions.

$C(1,3)$ is an $1+m$ by q matrix which describes how the outputs are mapped onto the inputs and constant. This matrix must also equal the zero matrix.

$C(2,1)$ is an n by $1+m$ matrix which describes how inputs and constants are mapped onto program variables. This submatrix is unrestricted in form.

$C(2,2)$ is an n by n matrix describing how program variables are mapped onto program variables. This submatrix is also 0 since C represents a symbolic execution of a particular path from the beginning of the module up to the selected predicate. The only possible exception is the row corresponding to a program variable which is not defined along the path. Such a row would have a 1 in the column corresponding to the program variable, and zeros elsewhere.

$C(2,3)$ is an n by q matrix which describes how outputs are mapped onto program variables (also equal to the zero matrix).

$C(3,1)$ is a q by $1+m$ matrix which describes how inputs and constants are mapped onto output variables. This submatrix is unrestricted in form.

$C(3,2)$ is a q by n matrix which describes how program variables are mapped onto outputs. This submatrix must be equal to the zero matrix for the same reasons as submatrix $C(2,2)$.

$C(3,3)$ is a q by q matrix which describes how outputs are mapped into outputs. This must also be equal to

the zero matrix except in rows corresponding to unassigned output variables.

In condition (2) the only submatrix of the C matrix which is of interest is $C(2,1)$. Therefore the expression $(\bar{T}C)$ is a predicate interpretation vector of which only the first $1+m$ positions can be non-zero. These positions represent the manner in which the input variables are mapped onto the scalar. Hence, condition (2) can now be rewritten as follows.

$$\begin{aligned} \forall k > 0, \forall \bar{e}_{(1, \dots, 1+m)} \neq \bar{0}, \exists \bar{v}_0 \text{ such that} \\ (\bar{T}C(2,1))_{(1, \dots, 1+m)} (\bar{v}_0(1, \dots, 1+m) + \bar{e}_{(1, \dots, 1+m)}) \neq \\ k (\bar{T}C(2,1))_{(1, \dots, 1+m)} \bar{v}_0(1, \dots, 1+m) \end{aligned} \quad (3)$$

where the subscripted quantities indicate the positions of each vector that are being used.

We can further note that the constant position in $\bar{e}_{(1, \dots, 1+m)}$ is always zero. Therefore the first column of $C(2,1)$ can play no part in determining if a predicate satisfies condition (3). If we define $C'(2,1)$ to be $C(2,1)$ without the first column then condition (3) can be rewritten as (numbering the remaining columns from 2 to $1+m$):

$$\begin{aligned}
& \forall k > 0, \forall \bar{e}_{(2, \dots, 1+m)} \neq \bar{0}, \exists \bar{V}_0 \text{ such that} \\
& (\bar{T}C'(2,1))_{(2, \dots, 1+m)} (\bar{V}_0(2, \dots, 1+m) + \bar{e}_{(2, \dots, 1+m)}) \neq \\
& k (\bar{T}C'(2,1))_{(2, \dots, 1+m)} \bar{V}_0(2, \dots, 1+m) \quad (4)
\end{aligned}$$

Given a predicate T and a feasible path leading up to T consisting of computations that can be described by the matrix C , condition (4) describes the circumstances under which some input to the module can cause that predicate to shift from the correct predicate interpretation when an error occurs in the initial environment vector $\bar{V}_0$. We therefore seek a feasible path leading up to a predicate for which it is possible to detect any of the possible input errors in any of the m input variables to the module by examining the way the predicate shifts for each of the input errors.

Even though there are an infinite number of ways in which the incorrect input can differ from the correct input (i.e. there are an infinite number of possible $\bar{e}$ vectors), from linear algebra we know that there are at most m linearly independent errors possible. All the possible $\bar{e}$ vectors can be represented as linear combinations of any of the m linearly independent $\bar{e}$ vectors. We will refer to such a set of m possible error

vectors as error directions, and the space spanned by the set of m linearly independent error directions as the error space of the module.

For a predicate interpretation to satisfy condition (4), the path containing the predicate interpretation must "span the error space" of the module, so that all possible error directions are included in the set of error directions which can be detected by predicate interpretations along that path. Since the dimension of the error space and the input space are the same, we will call such a path input space spanning.

In order to better understand this requirement consider the predicate $P1 \geq 0$ from Figure 3.1. The interpretation of this predicate is $I1 + I2 \geq 0$, which appears to be sensitive to errors in the input $I1$ as long as the error produces a new interpretation that is not equal to $k(I1 + I2) \geq 0$. However, suppose that this predicate appeared along a path in the module previously constrained by the predicate $I1 = 1$. Since $P1 \geq 0$ cannot even be reached if $I1 \neq 1$ its ability to detect errors in $I1$ is severely limited.

The effect of the equality predicate $I1 = 1$ is to reduce the dimensionality of that subset of the input space which can follow a path including the predicate. In reducing the dimension of the input space that can follow the path, the dimension of the error space that is detectable by a predicate interpretation along the path is also reduced and certain errors will no longer be detectable along that path. It isn't necessary for a non-input space spanning path to include an equality predicate. In certain cases a combination of inequality predicates can have the same effect. Consider a path with the predicates $I1 \geq 1$ and $I1 \leq 1$, in this case the combined effect of the two predicates is to reduce the dimension of the input and error space.

We can now conclude.

Lemma 1: Given a module which can be described by the model presented earlier, in order to ensure that an error in the calling program is capable of producing a nonequivalent predicate interpretation along some given path in the module, it is sufficient that there exist a predicate T in the module and an input space spanning path leading to T satisfying

$\forall k > 0, \forall \bar{e}_{(2,\dots,1+m)} \neq \bar{0}, \exists \bar{v}_0$ such that

$$(\bar{T}C'(2,1))_{(2,\dots,1+m)} (\bar{v}_0(2,\dots,1+m) + \bar{e}_{(2,\dots,1+m)}) \neq k (\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{v}_0(2,\dots,1+m)$$

The next question to consider is "Under what conditions can a predicate interpretation satisfy Lemma 1?". The answer to this question can most easily be seen by first considering a subcase of Lemma 1. If we fix the value of k to be 1 then the condition in Lemma 1 can be written as

$\forall \bar{e}_{(2,\dots,1+m)} \neq \bar{0}, \exists \bar{v}_0$ such that

$$\begin{aligned} &(\bar{T}C'(2,1))_{(2,\dots,1+m)} (\bar{v}_0(2,\dots,1+m) + \bar{e}_{(2,\dots,1+m)}) \neq \\ &(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{v}_0(2,\dots,1+m) \end{aligned} \quad (5)$$

Condition (5) states that if an error occurs in the input to the module, then the predicate interpretation must not be equal to the predicate interpretation that would have occurred if the input were correct.

To understand under what conditions a predicate (or predicates) can satisfy condition (5) we will examine the conditions under which a predicate will fail to satisfy condition (5).

Lemma 2: Given a module with at least two input variables any predicate interpretation will fail to satisfy condition (5) for all error directions orthogonal to the predicate interpretation vector.

Proof. Consider the case for a two dimensional error space (a module with two input variables). The form of a predicate interpretation in this module would be

$$0 \text{ .relop. } a * X1 + b * X2 + c$$

corresponding to a predicate interpretation vector $(\bar{T}C'(2,1))$ of (a, b) . By definition, an error vector $(e1, e2)$ orthogonal to this predicate interpretation vector is one in which the dot product of the two vectors is zero.

$$(a, b) \cdot (e1, e2) = 0$$

or

$$a * e1 + b * e2 = 0$$

Rewriting condition (5) in terms of the two dimensional predicate interpretation results in

$\forall (e_1, e_2)^t \neq \bar{0}, \exists (X_1, X_2)^t$ such that

$$(a * (X_1 + e_1) + b * (X_2 + e_2)) \neq (a * X_1 + b * X_2)$$

or

$\forall (e_1, e_2)^t \neq \bar{0}, \exists (X_1, X_2)^t$ such that

$$((a * X_1 + b * X_2) + (a * e_1 + b * e_2)) \neq (a * X_1 + b * X_2).$$

However, since (e_1, e_2) is orthogonal to (a, b) the above expression can be reduced to

$\forall (e_1, e_2)^t \neq \bar{0}, \exists (X_1, X_2)^t$ such that

$$(a * X_1 + b * X_2) \neq (a * X_1 + b * X_2)$$

which can clearly never be satisfied. For higher dimensions of the input space, orthogonality ensures that the expression

$$(\tilde{T}C'(2,1))_{(2,\dots,1+m)} \bar{e}_{(2,\dots,1+m)}$$

will always be equal to zero, and the input error vector will be undetectable. qed.

Consider again the predicate $P1 \geq 0$ from Figure 3.1. The interpretation of this predicate is given by the

expression $I_1 + I_2 \geq 0$. If we choose an error vector orthonormal to this predicate interpretation vector (such as $(2, -2)$) then that error will produce no shift in the predicate interpretation.

The importance of this result is twofold. First, by characterizing the conditions under which a predicate can't satisfy condition (5), we have also characterized a necessary condition under which a predicate interpretation is capable of detecting an error vector (namely a predicate interpretation is only capable of detecting those errors that aren't orthogonal to itself). This is true since any non-orthogonal error vector will result in the expression

$$(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{e}_{(2,\dots,1+m)}$$

being not equal to 0.

Lemma 2 also demonstrates that a single predicate may not be capable of detecting all possible error vectors. Since any predicate interpretation of two or more dimensions will not shift for orthogonal error vectors, then in general, if there is more than one input to a module, no single predicate interpretation will be capable

of satisfying condition (5).

The effect of Lemma 2 may at first seem to be negative in that it prevents us from detecting all possible error vectors. However, if we shift out attention from looking for a single predicate interpretation, to looking for a set of predicate interpretations to satisfy condition (5), Lemma 2 provides us the necessary insight to obtain a solution.

From Lemma 2 we noted that a given predicate interpretation is incapable of detecting those error vectors that are orthogonal to itself, and is capable of detecting those vectors which aren't orthogonal to itself. In addition, the number of possible linearly independent orthogonal error vectors is limited by the number of inputs to the module. Therefore, it should be possible to satisfy condition (5) by choosing a set of predicate interpretations on input space spanning paths such that each interpretation is incapable of detecting a different linearly independent orthogonal error direction. In this way, even though each predicate interpretation will miss certain errors directions, some other predicate in the set will detect that particular error direction.

If for a particular module there are m input variables (i.e. m linearly independent error vectors), then to satisfy condition (5) we need only select a set of m predicate interpretations whose vectors are given by

$$\{[(\bar{T}C'(2,1))_{(2,\dots,1+m)}]_i\}_{i=1}^m$$

from input space spanning paths such that the $m \times m$ matrix M whose i^{th} row consists of $[(\bar{T}C'(2,1))_{(2,\dots,1+m)}]_i$ has a non-zero determinant (i.e. the m predicate interpretations are linearly independent with respect to positions 2 through $1+m$ of their predicate interpretation vectors, and therefore the orthogonal error vectors to each of the predicate interpretations are also linearly independent).

Consider the module MODULE2 in Figure 3.2 with three inputs. The module contains six paths, and three predicates. Along the six paths there are 7 possibly different predicate interpretations. Since each of the predicate interpretations in the module will remain unchanged for certain input errors, it is necessary to use a set of predicate interpretations to detect all three input error directions.

```

      SUBROUTINE MODULE2 (I1,I2,I3,O1)
1.    P1=I1+I2
2.    IF P1≥0
3.        THEN P2=2*I1+3
4.        ELSE P2=P1+I3
5.    ENDIF
6.    IF P2≥0
7.        THEN P3=P1+P2
8.        ELSE P3=-P2+I2+I3
9.    ENDIF
10.   IF P3≥0
11.       THEN O1=2*I1+3
12.       ELSE O1=P1+I3
13.   ENDIF
14.   RETURN
15.   END

```

Figure 3.2
Sample Module for m=3, n=3, and q=1

Along the THEN-THEN-THEN path in the module the predicate interpretation for the first predicate is $I1 + I2 \geq 0$. The THEN-ELSE-THEN path produces the interpretation $-2*I1 - I2 + I3 - 3 \geq 0$ for the third predicate, and along the ELSE-THEN-THEN path the second predicate produces the interpretation $I1 + I2 + I3 \geq 0$. If we use these three predicate interpretations to construct the M matrix we get

$$\begin{bmatrix} 1 & 1 & 0 \\ -2 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The determinant of the above matrix evaluates to 1, and therefore the three predicate interpretations are capable of detecting three different linearly independent error vectors. Since there are only three error directions possible under condition (5), these predicate interpretations are sufficient to satisfy condition (5).

The problem remains of generalizing the results obtained for condition (5) to the more general expression of condition (4). Condition (4) states that for an error to be detectable it must not transform the predicate interpretation into a multiple of itself. An example of how this occurs was given earlier. In general no predicate (or set of predicates) is capable of satisfying this condition for all possible error vectors. This is because for any correct initial environment vector we can choose an error vector such that the resulting predicate interpretation is a multiple of the correct interpretation. If we rewrite condition (4) as follows:

$$\begin{aligned}
 &\forall k > 0, \forall \bar{e}_{(2, \dots, 1+m)} \neq \bar{0}, \exists \bar{v}_0 \text{ such that} \\
 &\{((\bar{T}C'(2,1))_{(2, \dots, 1+m)} \bar{v}_0(2, \dots, 1+m)) + \\
 &((\bar{T}C'(2,1))_{(2, \dots, 1+m)} \bar{e}_{(2, \dots, 1+m)})\} \neq \\
 &k (\bar{T}C'(2,1))_{(2, \dots, 1+m)} \bar{v}_0(2, \dots, 1+m) \quad (4b)
 \end{aligned}$$

we can now examine the conditions under which an incorrect predicate interpretation can evaluate to be a multiple of the correct interpretation. In order to violate condition (4b) it is sufficient to choose a value of $\bar{e}$, for a particular predicate interpretation such that

$$\begin{aligned} & ((\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{e}_{(2,\dots,1+m)}) = \\ & j(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m) \text{ for } j > 0. \end{aligned}$$

Proof. If an error occurs such that

$$\begin{aligned} & ((\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{e}_{(2,\dots,1+m)}) = \\ & j(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m) \text{ for } j > 0 \end{aligned}$$

then by substituting this into condition (4b) we get

$$\begin{aligned} & \forall k > 0, \forall \bar{e}_{(2,\dots,1+m)} \neq \bar{0}, \exists \bar{V}_0 \text{ such that} \\ & \{((\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m)) + \\ & (j(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m))\} \neq \\ & k (\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m) \end{aligned}$$

or

$$\begin{aligned} & \forall k > 0, \forall \bar{e}_{(2,\dots,1+m)} \neq \bar{0}, \exists \bar{V}_0 \text{ such that} \\ & ((j+1)(\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m)) \neq \\ & k (\bar{T}C'(2,1))_{(2,\dots,1+m)} \bar{V}_0(2,\dots,1+m). \end{aligned}$$

Clearly in this form condition (4b) can't be satisfied as we only need to chose a value of $k = j+1$ to violate the condition. qed.

In order to choose predicate interpretations which are capable of detecting errors that cause a predicate interpretation to evaluate to a multiple of itself when an error occurs we need to reexamine the constant position of the predicate interpretation (which we earlier discarded). Recall that the constant position of the initial environment must always be equal to one (by definition). In addition the constant position of the error vector must be equal to zero, since the constant position is independent of the input to the module. By choosing a predicate interpretation that is non-zero in the constant position we can guarantee that the incorrect predicate interpretation can not be a multiple of the correct interpretation.

Rewriting condition (4) to include the constant position results in

$$\forall k > 0, \forall \bar{e}_{(1, \dots, 1+m)} \neq \bar{0}, \exists \bar{v}_0 \text{ such that}$$

$$(\bar{T}C(2,1))_{(1, \dots, 1+m)} (\bar{v}_0(1, \dots, 1+m) + \bar{e}_{(1, \dots, 1+m)}) \neq k (\bar{T}C(2,1))_{(1, \dots, 1+m)} \bar{v}_0(1, \dots, 1+m) \quad (6)$$

By combining the requirements that a predicate interpretation, or set of predicate interpretations, must meet to satisfy condition (5) and the more general condition (6) we can now define the conditions under which a set of predicate interpretations will shift when an input domain error occurs in a module.

If for a particular module there are m input variables (i.e. m linearly independent error vectors), then to satisfy condition (6) we need only select a set of $m+1$ predicate interpretations

$$\{[(\bar{T}C(2,1))_{(1,\dots,1+m)}]_i\}_{i=1}^{m+1}$$

from input space spanning paths such that the $m+1 \times m+1$ matrix M' whose i^{th} row consists of $[(\bar{T}C(2,1))_{(1,\dots,1+m)}]_i$, has a non-zero determinant (i.e. the $m+1$ predicate interpretations are linearly independent with respect to positions 1 through $1+m$ of their predicate interpretation vectors).

We can now conclude

Theorem 1: Given a set of $m+1$ predicate interpretations along input space spanning paths through the correct module such that the set of interpretations satisfy the condition on M' defined above. Then that set of predicate interpretations will be capable of detecting, by shifting for some value of the initial environment vector, any integration time domain error in the module.

As a result of the above theorem it is possible to place an upper bound on the number of paths through the module which will be helpful in detecting integration domain errors. If we assume a path will only be included in the integration test set if it contains a predicate interpretation which shifts for some linearly independent error direction that doesn't cause a predicate to shift on other paths in the integration test set, then since there are only $m+1$ possible linearly independent errors there will be at most $m+1$ paths in the integration test set. This result is important because it shows that the number of paths that need be examined at integration time in order to ensure the shifting of a predicate interpretation when an input error exists is independent of the original path complexity of the module, and is only dependent on the number of inputs to the module.

It may very well be the case that no set of $m+1$ predicate interpretations exists to satisfy the above condition (i.e. certain errors in the input to the module might not be reflected in any predicate interpretation in the module). The simplest example of when this could occur is when a particular input variable is only used in computations, and plays no part in the predicate interpretations for the module. In these cases it isn't necessary to find $m+1$ predicate interpretations, since not all $m+1$ error directions for integration domain errors exist. Instead, if only p error directions are possible for integration domain errors (where $p \leq m+1$) then a set of p linearly independent predicate interpretations covering those p error directions, and involving at most p paths, will suffice. In terms of the M' matrix, if the above situation occurs it will not be possible to build an M' matrix with a non-zero determinant. Instead, it would be sufficient to find p linearly independent rows of the M' matrix.

So far we have only discussed the existence of the integration test set without indicating how to determine the integration test set for a particular module. If we are given a set of paths that are in the integration test

set, and the partial M' matrix which indicates the error directions those paths will detect, then the process of determining if a new path in the module is capable of detecting any new errors is relatively simple.

What we are asking is "Is the predicate (or predicates) on the new path linearly independent of the predicates on those paths in the integration test set?". This question is answered computationally by performing a gaussian elimination on the new path and the partial M' matrix.

Consider a module with four input variables W, X, Y, and Z. Suppose we have already found two predicate interpretations that are members of the integration test set

$$2*W + X - 3*Z < 2$$

and

$$-W + 2*Y > 1.$$

The partial M' matrix for these two predicate interpretations is given by

$$\begin{bmatrix} -2 & 2 & 1 & 0 & -3 \\ -1 & -1 & 0 & 2 & 0 \end{bmatrix}$$

where the individual columns of the matrix represent the constant position and the input variables W, X, Y, and Z. The rows of the matrix represent the coefficients of each of the individual predicate interpretations. If we reduce the above matrix with gaussian elimination we get

$$\begin{bmatrix} 1 & 1 & 0 & -2 & 0 \\ 0 & 4 & 1 & -4 & -3 \end{bmatrix}.$$

If you wish to decide if the predicate $7*W + 2*X - 6*Y - 6*Z > 1$ will add any error detection ability to the integration test set, it is necessary to determine if this new predicate is independent of the existing integration test set. In this case, we add this new predicate interpretation to our partial M' matrix resulting in

$$\begin{bmatrix} 1 & 1 & 0 & -2 & 0 \\ 0 & 4 & 1 & -4 & -3 \\ -1 & 7 & 2 & -6 & -6 \end{bmatrix}.$$

Performing gaussian elimination results in

$$\begin{bmatrix} 1 & 1 & 0 & -2 & 0 \\ 0 & 4 & 1 & -4 & -3 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Since the partial M' matrix is reduced back to the original two linearly independent rows this new predicate interpretation will not shift for any error directions that the previous two predicate interpretations wouldn't have shifted on. This is because the new predicate interpretation is linearly dependent on the previous two interpretations. (Row 3 of the matrix is equal to 2 times row 2 minus row 1).

If the predicate $Y + Z \geq 0$ is being considered for inclusion in the integration test set, then by adding it to the partial M' matrix and performing gaussian elimination we get the partial M' matrix

$$\begin{bmatrix} 1 & 1 & 0 & -2 & 0 \\ 0 & 4 & 1 & -4 & -3 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

In this case the new predicate interpretation will shift for error directions that do not cause shifts in the previous predicate interpretations. Specifically, the new predicate is capable of detecting errors in Y or Z as long as the both aren't in error by equal but opposite amounts.

Determining the integration test set can be done by repeatedly performing this gaussian elimination for each

path in the module until either the M' matrix is found, or until all the paths in the module are examined (in which case there are less than m possible error directions). An example of how this is done for a complete module is shown in Chapter VI. Even though the size of the integration test set is independent of the path complexity of the module, it appears we might need to examine every path in the module to find the set. In many cases this isn't unreasonable as while the program might contain too many paths to examine each one, the path complexity of each module may be small enough to allow this complete examination (this contention will be examined in greater detail in Chapter VI).

If when unit testing the module to determine that it is correct before performing the integration testing a path testing strategy is employed which uses symbolic execution, then the determination of the integration test set can be done in conjunction with unit testing. The only additional overhead would be the gaussian elimination required for each path. While the complexity of gaussian elimination varies depending on the algorithm employed, in general gaussian elimination can be considered to have a computational complexity of $\sim n^3$, where n is the number of elements in the vectors (in this case $m+1$).

Up to this point we have only discussed the detection of integration domain errors in terms of creating a shift in a particular predicate interpretation. However, a shifted predicate interpretation is only detectable when that predicate interpretation forms part of the boundary conditions defining the subdomain of input points that follow the path. Specifically a predicate interpretation will be redundant if it has been superseded by some other predicate interpretation. If that superseding predicate also occurs in the module whose integration test set is being determined, then a reasonable path oriented testing strategy should be capable of indicating that the predicate interpretation has been superseded. In this case that particular predicate interpretation should not be considered for inclusion in the integration test set of the module. A particular predicate interpretation in the module is also incapable of detecting errors if it is superseded by predicate interpretations outside of the module (in the calling program). This situation creates a more difficult problem, and will be discussed in Chapter V.

III.3. Detection of Integration Time Computation Errors

We can now address the second type of integration error introduced in Chapter II, that of integration computation errors. Using the restrictions presented earlier the results of the computations along a particular path can be represented by

$$\bar{V}_f = C_{0q} \dots C_{01} C_k \dots C_1 \bar{V}_0$$

where $C_k \dots C_1$ represent the computations which assign program variables along the path, $C_{0q} \dots C_{01}$ represent the assignments to the output variables, $\bar{V}_0$ is the initial environment vector and $\bar{V}_f$ is the final environment vector. The expression can be simplified to

$$\bar{V}_f = C_O C_P \bar{V}_0$$

and further reduced to

$$\bar{V}_f = C \bar{V}_0$$

where C is now a single $1+m+n+q$ by $1+m+n+q$ matrix which represents the results of all the computations performed along the particular path.

The question to be considered is "In what way can an error in the input to the module affect the results of the module through the computation statements?". As before the error in the initial environment vector can be represented as

$$\bar{V}'_0 = \bar{V}_0 + \bar{e} \quad \bar{e} \neq \bar{0}$$

where $\bar{V}_0$ is the correct initial environment vector, and $\bar{e}$ is the initial error term. As before $\bar{e}$ is restricted to be 0 in the first position, and 0 in the last $n+q$ positions.

Substituting this into the expression for the computations along a path yields

$$\bar{V}_f = C(\bar{V}_0 + \bar{e})$$

or

$$\bar{V}_f = C\bar{V}_0 + C\bar{e}.$$

Since the erroneous input follows the correct path through the module, $C\bar{V}_0$ represents the "correct" final environment vector. Therefore the error is only detectable when $C\bar{e}$ is detectable in the final environment vector of the module. Specifically, we are interested in choosing computations such that

$$C\bar{e} \neq \bar{0}.$$

Since an error in the input is only detectable if it is transformed to the output of the module, we again are only concerned with a submatrix of the C matrix. In particular, the only submatrix of interest is the $C(3,1)$ matrix which describes how the inputs and constant are mapped onto the outputs. By rewriting the expression for the results of the computations along a particular path only in terms of the relevant parts of the vectors the following equation is obtained.

$$\begin{aligned}\bar{V}_{f(1+m+n+1, \dots, 1+m+n+q)} &= C(3,1)\bar{V}_0(1, \dots, m+1) \\ &+ C(3,1)\bar{e}_{(1, \dots, m+1)}\end{aligned}$$

As before, the constant position in $\bar{e}$ is always zero, and can be discarded giving

$$\begin{aligned}\bar{V}_{f(1+m+n+1, \dots, 1+m+n+q)} &= C'(3,1)\bar{V}_0(2, \dots, m+1) \\ &+ C'(3,1)\bar{e}_{(2, \dots, m+1)}\end{aligned}$$

where $C'(3,1)$ is $C(3,1)$ without the first column. We can now say that for an error in the input to be transmitted to an output of the module $C'(3,1)\bar{e}_{(2, \dots, m+1)}$ must be detectable in the output of the module. As before this condition can be simply expressed as

$$C'(3,1)\bar{e}_{(2,\dots,m+1)} \neq \bar{0} \quad (7)$$

From condition (7) it is now possible to describe the conditions under which a path through the module is capable of detecting any of the possible input errors to the module.

Lemma 3 Examination of the outputs of a module on any test exercising an input space spanning path for which

$$\forall \bar{e}_{(2,\dots,m+1)} \neq \bar{0} \Rightarrow C'(3,1)\bar{e}_{(2,\dots,m+1)} \neq \bar{0}$$

is sufficient to enable the detection of any integration time computation error affecting the module.

Consider the module in Figure 3.3 with three inputs, two outputs, and three program variables. This particular module has four possible paths, and therefore four possible $C'(3,1)$ matrices.

The symbolic value of the two output variables along each of the four paths are as follows.

```
THEN-THEN path:
    O1=3*I2+I3+2
    O2=I1+2*I2+I3
```

```

THEN-ELSE path:
    O1=I1-4*I2-I3
    O2=I1+2*I2+I3

ELSE-THEN path:
    O1=2*I1-2*I2+I3+2
    O2=3*I1-3*I2+I3

ELSE-ELSE path:
    O1=-I1+I2-I3
    O2=3*I1-3*I2+I3

```

Therefore the resulting $C'(3,1)$ matrices are

THEN-THEN path: $\begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 1 \end{bmatrix}$

THEN-ELSE path: $\begin{bmatrix} 1 & -4 & -1 \\ 1 & 2 & 1 \end{bmatrix}$

ELSE-THEN path: $\begin{bmatrix} 2 & -2 & 1 \\ 3 & -3 & 1 \end{bmatrix}$

ELSE-ELSE path: $\begin{bmatrix} -1 & 1 & -1 \\ 3 & -3 & 1 \end{bmatrix}$.

```

SUBROUTINE MODULE3 (I1,I2,I3,O1,O2)
1.  P1=I1-I2
2.  IF P1>0
3.      THEN P2=3*I2+I3
4.      ELSE P2=2*P1+I3
5.  IF P2>0
6.      THEN P3=P2+2
7.      ELSE P3=P1-P2
8.  O1=P3
9.  O2=P1+P2
10. RETURN
11. END

```

Figure 3.3
Sample Module for m=3, n=3, and q=2

If we examine each path of this module in relation to Lemma 3 we discover that no path is capable of detecting all of the possible input errors. For example, assume that the correct input to the module should be $I_1=1$, $I_2=2$, and $I_3=3$. However, if the incorrect input was $I_1=2$, $I_2=3$, and $I_3=0$ then on the THEN-THEN path of the module the correct and incorrect values of O_1 and O_2 are as follows.

Correct input:

$$\begin{aligned} O_1 &= 3 \cdot I_2 + I_3 + 2 = 3 \cdot 2 + 3 + 2 = 11 \\ O_2 &= I_1 + 2 \cdot I_2 + I_3 = 1 + 2 \cdot 2 + 3 = 8 \end{aligned}$$

Incorrect input:

$$\begin{aligned} O_1 &= 3 \cdot I_2 + I_3 + 2 = 3 \cdot 3 + 0 + 2 = 11 \\ O_2 &= I_1 + 2 \cdot I_2 + I_3 = 2 + 2 \cdot 3 + 0 = 8 \end{aligned}$$

The THEN-THEN path is incapable of detecting an error vector where

$$\bar{e}_{(2, \dots, m+1)} = (1 \ 1 \ -3).$$

In fact, any error vector that is a multiple of the error vector $(1 \ 1 \ -3)$ is also undetectable along this path

As in the integration domain error case it isn't necessary for a single path which satisfies Lemma 3 to exist. Consider an m by m matrix M constructed from various rows in $C'(3,1)$ matrices taken from different input space spanning paths through the module. If M has a

non-zero determinant then from linear algebra we know that

$$\forall \bar{e}_{(2,\dots,m+1)} \neq \bar{0} \Rightarrow M' \bar{e}_{(2,\dots,m+1)} \neq \bar{0}. \quad (8)$$

Intuitively this means that in order to detect all possible error vectors we need to find m linearly independent functions of the inputs computed somewhere in the module. There is no requirement that these computations exist on the same path. We can now conclude

Lemma 4 Examination of the appropriate output variables of the module from any test exercising those paths corresponding to the construction of M as defined above is sufficient to enable the detection of any integration time computation error affecting the module.

Using Figure 3.3 if we were to examine the assignments to 01 and 02 along the THEN-THEN path in addition to the assignment to 01 along the ELSE-THEN path we would construct the following M matrix

$$\begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 1 \\ 2 & -2 & 1 \end{bmatrix}$$

whose determinant evaluates to -3 . In this case any input error to the module will be reflected in the module's

outputs along either the THEN-THEN or the ELSE-THEN paths. (Instead of finding the determinant of the above matrix guassian elimination could have been used, as in the previous section, to show that the rows of the matrix are linearly independent.)

As in the integration domain error case, it may well be that no set of m linearly independent computations exists to satisfy condition (8). In these cases, if only p error directions are possible for integration computation errors (where $p \leq m$) then a set of p linearly independent computations covering those p error directions will suffice. Consider the sample module in Figure 3.4 with four inputs, and two outputs. For this module the symbolic value of the two output variables along the four possible paths is the same as it was for Figure 3.3. However, for this module there is an additional input variable I_4 which is only used in a predicate, and plays no part in the symbolic value of the output variables. For Figure 3.4 the THEN-THEN and ELSE-THEN paths produce the following M matrix.

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 1 & 2 & 1 & 0 \\ 2 & -2 & 1 & 0 \\ 3 & -3 & 1 & 0 \end{bmatrix}$$

```

      SUBROUTINE MODULE4 (I1,I2,I3,I4,O1,O2)
1.    P1=I1-I2
2.    IF I4>0
3.        THEN P2=3*I2+I3
4.        ELSE P2=2*P1+I3
5.    IF P2>0
6.        THEN P3=P2+2
7.        ELSE P3=P1-P2
8.    O1=P3
9.    O2=P1+P2
10.   RETURN
11.   END

```

Figure 3.4
Sample Module for m=4, n=3, and q=2

Even though the M matrix has a zero determinant, the THEN-THEN and ELSE-THEN paths are still able to transmit to the module's outputs all integration computation errors that are transmitted along any other path through the module.

Lemma 4, while useful, falls short of guaranteeing detection of all integration time computation errors. This is because it only deals with transmitting errors in the module's inputs to the module's outputs. If the goal is to ensure detection of integration computation errors without having to examine intermediate results in the program (such as the module's outputs), it is necessary to make sure that the calling program doesn't "lose" the error that has been passed to the module's outputs. In other words, we need to guarantee that an error in the module's

outputs will be transmitted to the outputs of the calling program. In order to guarantee this, additional selection criteria for the paths in the integration test set will need to be devised. This is true since the results of a computation in the module can be modified by the results of later computations (in the calling program) before an integration error is transmitted to the outputs of the calling program.

A simple example of how this deficiency might occur is the case where paths have been chosen through the module to guarantee that if an error exists in the module's first input variable it will be transmitted to the module's first output variable. This would guarantee the detectability of the error in the module's outputs. However it might be the case that along a particular path in the calling program the first output variable of the module might never be used, or only used in such a way that the error will not be detected in the final outputs of the program. In this case the error will never be transmitted to the outputs of the calling program.

Consider the program and module in Figure 3.5. In this case the module has two input variables (X1 and X2) and three output variables (Y1, Y2, and Y3). Examination

of the module reveals that either of the two paths is capable of transmitting any error in the module's input to the module's outputs. From the module's viewpoint, it doesn't matter which of the two paths is chosen as the integration test set. In this case, assume that the THEN path is chosen as the integration test set. If an error occurs in the second input to the module (variable B) this error will be transmitted to the module's outputs, but due to the computations in the program following the module (specifically the statement $O2 = E - B$) the error will not be reflected in the outputs of the program. If we had chosen the ELSE path in the module as the integration test set, the error would be detectable in the outputs of the program. The problem cannot be corrected by simply choosing

```

      .
      .
      CALL MODULE5 (A,B,C,D,E)
      O1 = C
      O2 = E - B
      WRITE (O1, O2)
      END

      SUBROUTINE MODULE5 (X1,X2,Y1,Y2,Y3)
1.    IF X1 > 0
2.        THEN Y1 = X1
3.            Y2 = X1
4.            Y3 = X1 + X2
5.        ELSE Y1 = X1 + X2
6.            Y2 = X2
7.            Y3 = X1
8.    RETURN
9.    END

```

Figure 3.5
Sample Program and Module for Computation Errors

the ELSE path in the module as, for a different calling program, the THEN path might have been the correct choice.

The problem identified by this example is that without knowledge about the program following the call to the module we have no way of knowing if an error transmitted to a particular output of the module will be transmitted to the program's outputs. Since from the module's viewpoint we can't determine how the module's outputs are used by the calling program, one method of avoiding this type of situation can be obtained by choosing paths in the module to transmit each input error direction to all possible output variables of the module (instead of to a single output variable). If an error in the module's inputs causes an error to be transmitted to each of the module's outputs (along some path through the module) then the problem identified in the last example can be avoided.

We can view the outputs of the module as defining a new vector space which we will call the output error space of the module. Earlier in this section we discussed a method of transmitting each direction in the input error space of the module to a single direction in the module's output error space. In order to do this we choose paths in

order to build a single M matrix whose rows were made up of linearly independent symbolic interpretations of the output variables. In this M matrix the individual rows may represent symbolic interpretations of different output variables of the module. To avoid the problem identified in the last example it is necessary to choose paths through the module to transmit each direction in the input error space to all possible directions in the module's output error space. This would correspond to choosing paths in order to generate a set of M matrices (where there is a separate M matrix for each of the module's outputs). The rows of each of these M matrices would be composed of linearly independent symbolic interpretations of the same output variable. Since each matrix would contain up to m rows (for the m inputs to the module), and there would be n matrices (for the n output variables), there would be a possibility of $m \times n$ error directions to be spanned by the paths in the integration test set.

Even this would not guarantee transmission of an error in the input to the module to the outputs of the program. However, it is only necessary to ensure that if an error in the input to the module can be transmitted along some path to the outputs of the program, then it can be transmitted along one of the paths in the integration

test set.

There is an additional way in which the calling program can interfere with the transmitting of errors in the outputs of the module to the outputs of the program. The best way to describe the problem is through an example. Consider the module and calling program in Figure 3.6. In order to choose paths such that each input error direction is passed to each output direction, we need to

```

      .
      .
      C = X
      CALL MODULE6 (A,B,C,D,E)
      Y = D - X
      Z = E - X
      WRITE (Y, Z)
      END

      SUBROUTINE MODULE6 (I1,I2,I3,O1,O2)
1.    P1 = I1 - I2
2.    IF P1 > 0
3.        THEN P2 = 3*I2 + I3
4.            IF P2 > 0
5.                THEN O1 = P2 + 2
6.                    O2 = P1 + P2
7.            ELSE O1 = P1 - P2 + 2*I3
8.                O2 = P1 + P2
9.        ELSE P2 = 2*P1 + I3
10.           IF P2 > 0
11.               THEN O1 = P2 + 2
12.                   O2 = P1 + P2
13.           ELSE O1 = P1 + P2 + 2*I3
14.               O2 = 3*P2 - P1 + 6*I2
15.    RETURN
16.    END

```

Figure 3.6
Sample Program and Module with Error Losses

choose paths to generate a set of M matrices, a separate M matrix for each different output variable. The module in this example has four paths, where the symbolic interpretations of the output variables along the four paths are given by

```

THEN-THEN path:
    O1=3*I2+I3+2
    O2=I1+2*I2+I3

THEN-ELSE path:
    O1=I1-4*I2+I3
    O2=I1+2*I2+I3

ELSE-THEN path:
    O1=2*I1-2*I2+I3+2
    O2=3*I1-3*I2+I3

ELSE-ELSE path:
    O1=3*I1-3*I2+3*I3
    O2=5*I1+I2+3*I3

```

If we were to choose paths through the module so as to generate the set of M matrices we could choose the THEN-THEN, THEN-ELSE, and ELSE-THEN paths. If M_1 is the M matrix for the first output variable, and M_2 is the M matrix for the second output variable we get the following matrices for the three paths chosen

$$M_1 = \begin{bmatrix} 0 & 3 & 1 \\ 1 & -4 & 1 \\ 2 & -2 & 1 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 1 & 2 & 1 \\ 3 & -3 & 1 \end{bmatrix}.$$

(Notice that the M_2 matrix contains only two rows, as the assignment to 02 along the THEN-THEN and THEN-ELSE paths are the same.) The ELSE-ELSE path in this case would add no linearly independent rows to either of the M matrices generated from the first three paths.

The calling program, in this example, assigns the variable C to the value of X before calling the module, passing C as the third input variable. If an error is made to the assignment of X (resulting in an error in the module in the third input variable), the error will be lost in the calling program on all paths in the module except the ELSE-ELSE path.

Along the first three paths in the module the C variable (from the main program) doesn't occur in the outputs of the program (because X is subtracted from the results of the module). Since C (and therefore X) plays no part in the outputs of the program, an error in the assignment to X can't be detected along the first three paths. The ELSE-ELSE path in the module will reveal an error in X since along this path the error in X won't be

canceled by the statements following the call to the module.

The problem identified by the above example is caused by variables in the main program that aren't passed to the module. In this case the erroneous program variable is not only passed to the module, but is used by the program following the call to the module to cancel the error along the first three paths. In order to solve the problem we need a way of accounting for potential loss of errors due to the influence of variables in the calling program.

In order to see how we might achieve a solution to the above problem we need to expand our vector space which incorporates this "mapping" of each input error direction onto each output error direction. If we consider the module from the viewpoint of the calling program, we note that the purpose of the module is to transform the inputs to the module to the outputs of the module. Stated in terms of our vector space model, the module acts on a particular value of the environment vector of the calling program generating a new value of the calling program's environment vector. However, the dimension of the program's environment vector space depends on the number of variables in the calling program. If we assume the

calling program has r elements in its environment vector (divided between input, program, and output variables), then the module performs a mapping of this r dimension vector into another r dimension vector. The result can be viewed as each path of the module performing one of $r \times r$ possible linearly independent mappings of the program's environment vector onto itself. To ensure the transmission of an error in the input to the module (the program's environment vector before the call to the module) to the outputs of the program we would need to choose paths in the module to span this $r \times r$ vector space.

We will refer to this vector space as the transformation space of the module. Whereas the problem of transmitting an error in the module's inputs to any single direction of the module's output space involves choosing paths so as to "span" all linearly independent directions in the input vector space of the module, the problem of choosing paths through the module to transmit each input error direction to all possible output error directions involves trying to "span" all linearly independent directions in the module's (or program's) transformation vector space. By choosing paths in the module to span this vector space we are ensuring that all linearly independent transformations of the program's environment vector that

can occur in the module are occurring in the integration test set for the module.

The problem with this approach is that the transformation space appears to be dependent on the size of the environment vector of the calling program. However, if we examine the transformation space in greater detail we can simplify it considerably. Of the $r \times r$ positions in the transformation space many may describe the mapping of variables which aren't passed to the module. The part of the transformation space that describes the transformation of the module's inputs to its outputs is a $m \times q$ subvector of the $r \times r$ vector.

If we have a program with 5 elements in its environment vector (1 constant element c , 1 input to the module I , 1 output from the module O , and 2 variables which aren't passed to the module $P1$ and $P2$) then the resulting transformation vector for this module would contain the following 25 elements

```
( c:c, c:P1, c:P2, c:I, c:O,
  P1:c, P1:P1, P1:P2, P1:I, P1:O,
  P2:c, P2:P1, P2:P2, P2:I, P2:O,
  I:c, I:P1, I:P2, I:I, I:O,
  O:c, O:P1, O:P2, O:I, O:O )
```

(where $c:c$ is the element of the vector that describes how the module transforms the constant position onto the constant position). However, many elements of this transformation vector must be 0 by definition. Specifically, the positions

$c:P1, c:P2, c:I, c:0,$
 $P1:c, P1:P2, P1:I, P1:0,$
 $P2:c, P2:P1, P2:I, P2:0,$
 $I:c, I:P1, I:P2,$
 $0:c, 0:P1, 0:P2, 0:I, \text{ and } 0:0$

can only have the value 0 as no transformation is possible between these variables when the module is called. Since these positions of the transformation vector are constrained to be 0, they can be eliminated from the transformation vector resulting in the vector

($c:c, P1:P1, P2:P2, I:I, I:0,$).

Some of the elements in this vector are also restricted in value. Specifically the positions $c:c$, $P1:P1$, $P2:P2$, and $I:I$ must all have a value of 1 in the transformation vector as each of these variables are transformed onto themselves by the call to the module

(i.e. their value isn't changed by the call to the module). Algebraically these positions can be reduced to a single position in the transformation vector with a value of 1. The result is a transformation vector of the form

$$(1, I:0).$$

This transformation vector of the calling program's environment vector onto itself is now dependent only on the number of input and output variables in the module.

If we generalize the above result, the elements of the transformation vector form a vector space consisting of $m*q+1$ members where the elements in the vector space are defined as follows.

$m*q$ represents all of the possible linearly independent transformations from the inputs to the module (m elements) to the module's outputs (q elements). This part of the transformation vector is algebraically equivalent to the set of M matrices introduced earlier.

1 is a position holder which represents all variables that exist in the calling program's environment

vector but are not passed to, or not modified by the module.

Each path through the module will contribute one vector to this transformation vector space, as each path only performs one transformation on the program's environment vector. The elements of the transformation vector for a particular path through a module are derived from the $C'(3,1)$ matrix. As before the $C'(3,1)$ matrix describes how the input variables are mapped onto the output variables. The individual elements of the transformation vector also describe how each input variable is mapped into each output variable. The primary difference between the two vector spaces is that for the $C'(3,1)$ matrix there is an individual vector for each output variable, and for the transformation vector the mapping of the input to the output variables is combined into a single vector, as follows.

Consider a program with three inputs and two outputs, and a path through the module that produces the following $C'(3,1)$ matrix.

	<u>I1</u>	<u>I2</u>	<u>I3</u>
01	2	0	4
02	1	3	0

This corresponds to the following symbolic values for the output variables

$$\begin{aligned} 01 &= 2*I1 + 0*I2 + 4*I3 \\ 02 &= 1*I1 + 3*I2 + 0*I3. \end{aligned}$$

The transformation vector for this example would contain the following 7 elements.

<u>1</u>	<u>01:I1</u>	<u>01:I2</u>	<u>01:I3</u>	<u>02:I1</u>	<u>02:I2</u>	<u>02:I3</u>
1	2	0	4	1	3	0

The transformation vector for a path can be obtained by appending the rows of the $C'(3,1)$ matrix to each other to form a single vector. The initial 1 in the transformation vector represents those variables that aren't passed to, or modified by, the module.

Since there are only $m*q+1$ linearly independent error directions in the transformation space of a module it should be possible to choose paths through the module to construct an M' matrix such that each row of M'

corresponds to the transformation vector of a single path through the module. If M' has a non-zero determinant then we can conclude

Lemma 5 Examination of the output variables of the calling program for any test exercising those paths corresponding to the construction of M' as defined above is sufficient to enable detection of any integration time computation error effecting the module.

As in the case of integration domain errors it is necessary to be able to find the elements to construct M' in a computationally efficient manner. If we again assume that in unit testing of the module a symbolic execution was performed for each path in the module, then with the added overhead of one Gaussian elimination per path we can determine if each new path will contribute to the M' matrix. While this seems to be equivalent to the integration domain error case it should be noted that for integration computation errors the Gaussian elimination is being performed on a matrix of size $m \cdot q + 1$ instead of a matrix of size m .

In addition the upper bound for the number of paths in the integration test set for integration computation

errors is at most $m*q+1$ members (because there are only $m*q+1$ linearly independent error directions in the transformation space).

III.4. Combined Integration Testing

If when testing the desire is to check for both integration domain and computation errors (as would typically be the case), then the integration test set is simply the union of the integration test sets for each case. This places the following upper bound on the number of paths in the integration test set.

$$(m) + (m*q+1)$$

In general the size of the integration test set may be smaller than this upper bound allows, for several reasons.

First, for integration domain errors a path might contribute to the detection of more than 1 input error direction because a particular path might contain more than one predicate. If the predicates on one path detect different errors, then the size of the integration test

set can be reduced by the number of additional linearly independent predicates that exist. This leads to a heuristic which may be helpful when building the integration test set. This heuristic says that long paths should be examined first (where long is defined by the number of predicates on the path) as they can potentially detect more integration domain errors. While there is no guarantee, this could help to reduce the size of the integration test set.

Second, many of the paths in one integration test set might be included in the other set. Hence the union of the two sets might have fewer members than the sum of the sizes of the two sets.

The third reduction in the size of the integration test set was introduced previously. This deals with the case when not all error directions are detectable for either integration domain or computation errors. While at this point we have introduced no arguments for the importance of this factor, in chapter VI we will see where this factor can have a substantial effect on the size of the integration test set.

The fourth reduction in the size of the combined integration test set lies in the fact that the two integration test sets duplicate each other in their error detection ability. All integration errors begin as an error in the input to the module, and if the integration test set for integration domain errors sufficiently covers a certain error direction then there is no reason for the integration test set for integration computation errors to also detect the same error. If selection of the test set for integration domain errors is performed first then this can reduce the size of the error space that integration computation errors have to detect by x^*q where x is the number of error directions detected by the integration test set for integration domain errors. In chapter V we will examine some cases where it might not be advantageous to take advantage of this fourth possible reduction in the size of the integration test set.

IV. Integration Testing with Nonlinear Predicates and Computations

In the previous chapter we examined a method of selecting paths in a module which if retested during integration testing would reveal integration domain and computation errors. The results were obtained by restricting the functional complexity of the module's computations and predicates. While many researchers have argued that programs are usually made up of simple operations, there is still a wide class of programs which don't meet the linearity restrictions of the previous chapter. In order for the integration strategy to be widely applicable the linearity restriction needs to be relaxed.

First, it should be noted that the linearity assumption was not used in its entirety. For example, we can apply the results for detecting integration domain and computation errors as long as we can find a set of paths

which produce enough linear predicates and computations so that the non-singular M' matrices can be produced. While it may not be reasonable to require all computations and predicates to be linear, it is certainly more realistic to assume that some predicates and computations computed along some paths in the module are linear. This restriction is weaker than that of a "linearly domained" program [Zeil 81, White 80], where every predicate is required to have a linear interpretation.

If even the above assumption is too restrictive, a method of dealing with nonlinearity needs to be incorporated into the integration testing strategy. If when selecting the integration test set for a particular module a path with a nonlinear predicate or computation is encountered, the strategy as currently proposed can't determine if that path is capable of detecting any errors not detectable by the other paths in the integration test set.

One method of incorporating nonlinear predicates and computations would be to include each of these paths in the integration test set without trying to determine the types of integration errors that are detectable along those paths. This method would be very simple to add to

the methods of Chapter III, but suffers from an obvious drawback. If the module contains nonlinearity on many of its paths this solution could cause the size of the integration test set to grow considerably. This might negate any reductions achieved by using the integration testing method. However, if the amount of nonlinearity is relatively small this solution may prove to be the simplest to implement and apply. In Chapter VI we will examine a program where the amount of nonlinearity is such as to allow this solution to be applied with very little increase in the size of the integration test sets.

In the next sections we will introduce a method of determining the types of errors that are detectable along nonlinear paths in a module. This will lead to a method of determining an integration test set for a module containing nonlinear predicates and computations.

IV.1. A Model of Modules with Nonlinearity

In order to incorporate the idea of nonlinear predicates and computations the model of Chapter III needs to be slightly altered. The environment vector $\bar{V}$ remains as before, and the computations along a path still

transform one environment vector into another. However, if general nonlinearity is to be handled (i.e arbitrarily complex functions are allowed), the computations can no longer be represented as matrices. The results of the computations along a given path are now represented as

$$C[\bar{V}_0] \rightarrow \bar{V}_f$$

where the notation $X[a]$ represents the application of the function X to a . Predicates still transform the environment into a scalar for comparison to zero. The general form of a predicate would now be

$$0 \text{ .relop. } T[\bar{V}_T].$$

These are the only changes needed in the model to reflect arbitrary computations and predicates. The problem with this extension of the model is that results such as those obtained in Chapter III are no longer achievable. If we consider the case of integration computation errors, the initial environment vector can still be represented as

$$\bar{V}'_0 = \bar{V}_0 + \bar{e}$$

where $\bar{e}$ is the error term. The results of the computations along a path are now

$$c[\bar{V}_0 + \bar{e}].$$

However, it is no longer possible to separate the error term from the rest of the environment as

$$c[\bar{V}_0 + \bar{e}] \neq c[\bar{V}_0] + c[\bar{e}].$$

In addition, if C represents an arbitrarily complex function, the linear algebra methods of Chapter III no longer provide any information as to the types of input errors a particular computation or predicate is capable of detecting. In particular, we can no longer tell whether different C functions from different paths are capable of detecting (or not detecting) the same types of errors. In the previous chapter it was possible to compare the error detection abilities of different functions because of the properties of the linear functions.

We can extend the results of Chapter III if certain restrictions are placed on the complexity of the types of functions allowed in a module. A closer examination of Chapter III reveals the potential nature of those

restrictions. The essential property of the functions in Chapter III which allowed the linear algebra techniques to be applied was that the functions could be represented as vector spaces. With this in mind we can propose a new set of restrictions for the types of modules to which the techniques of Chapter III can be applied. In particular the third restriction of the model in Chapter III, requiring linear predicates and computations, can be relaxed to:

All functions C and T can be represented by a finite dimension vector space over the environment vector V .

The linear functions of Chapter III are a subset of the classes of functions that meet this restriction. In addition to linear functions the class of multinomials and polynomials of finite degree can also be represented as a vector space, and therefore fit the restriction. While many classes of functions (such as rational functions, or functions involving division) don't fit the vector space limitation, these functions can frequently be approximated by the use of polynomial expansion. In the following sections we will examine how the class of polynomials and multinomials can be handled by the techniques of Chapter

III.

IV.2 Integration Domain Errors with Nonlinear Predicates

The general form of a nonlinear predicate interpretation given the restrictions as imposed can now be written as

$$0 \text{ .relap. } (\bar{T}C)(\bar{V}_0)$$

where C is a d by d matrix (where d is the dimension of the vector space describing the functions in C) describing the computations leading up to the predicate, and $\bar{T}$ is a vector of dimension d that describes the predicate. In this case $(\bar{T}C)$ results in a row vector of dimension d which represents the nonlinear predicate interpretation.

Consider the module in Figure 4.1 where there are two inputs, two outputs, and three program variables ($m=2$, $n=3$, and $q=2$). For this module predicate interpretations can be considered to have the following general form

$$c_1 + c_2 I_1^2 I_2^2 + c_3 I_1^2 I_2 + c_4 I_1^2 + c_5 I_1 I_2^2 + c_6 I_1 I_2 + c_7 I_1 + c_8 I_2^2 + c_9 I_2.$$


```

      SUBROUTINE MODULE7 (I1,I2,O1,O2)
1.   P1=2*I1+I2-3
2.   P2=I2*P1
3.   IF P2≥0
4.       THEN P3=P2*I1
5.       ELSE P3=P1*I1
6.   ENDIF
7.   O1=P3
8.   O2=P1+P2+P3
9.   RETURN
10.  END

```

Figure 4.1
Nonlinear Module for m=2, n=3, and q=2

As in the linear case, output variables can play no part in a predicate, and program variables are eliminated and replaced with their interpretation in terms of input variables. For this example the result is a general form of the predicate interpretations of dimension nine (d is equal to 9). If we represent the predicate interpretation with a vector containing the values of the constants c1 through c9, then the predicate interpretation for the predicate $P2 \geq 0$ is represented as

$$(0,0,0,0,0,2,0,1,-3).$$

This corresponds to a predicate interpretation of

$$2*I2*I1+I2^2-3*I2 \geq 0.$$

The primary difference between this notation and that used in Chapter III is that the dimension of the $\bar{V}_0$ vector and the predicate interpretation are potentially different (they will differ if the complexity of the predicate interpretation is of greater complexity than the linear case). This dimensionality difference prevents us from directly multiplying the input variable elements of the environment vector by the predicate interpretation. However, this dimensionality difference, as we shall see in the analysis to follow, can be accounted for in such a way as to not affect the results that can be obtained.

Given an input vector to a module (of dimension m) we can transform that vector to a vector of dimension d . Again using the module in Figure 4.1, any input vector of the two input variables can be converted into a vector of dimension nine by multiplying and adding the two input variables in such a way as to generate the necessary nine elements. If we assume the value of the inputs to the module are $I_1 = 2$ and $I_2 = 3$ then this input vector can be transformed into a new vector of dimension nine in order to match the dimension of the predicate interpretation. In this case the transformed input vector would be

(1 36 12 4 18 6 2 9 3)

corresponding to the values of

$$c + I_1^2 I_2^2 + I_1^2 I_2 + I_1^2 + I_1 I_2^2 + I_1 I_2 + I_1 + I_2^2 + I_2.$$

where c is the constant term in the initial environment vector which by definition must be equal to 1.

If an error exists in the calling program resulting in an incorrect value being passed to the module, this can be viewed as an error in the initial environment vector. If $\bar{V}'_0$ is again used to denote the incorrect initial environment then

$$\bar{V}'_0 = \bar{V}_0 + \bar{e} \quad \bar{e} \neq \bar{0}.$$

As before $\bar{e}$ can only be non-zero in positions 2 through $m+1$.

If an error occurs in the initial environment vector then that error will be preserved in the transformed input vector. This is clearly true since the individual input

variables still appear as independent terms in the transformed input vector. Using Figure 4.1, if we assume that the correct initial environment vector should have been (1 2 4) ($I_1 = 2$ and $I_2 = 4$) but was instead (1 2 3) ($I_1 = 2$ and $I_2 = 3$) then the error term in this case is would be given by the vector (0 0 -1). The correct transformed input vector should have been

$$(1 \ 64 \ 16 \ 4 \ 32 \ 8 \ 2 \ 16 \ 4),$$

and the incorrect transformed input vector is

$$(1 \ 36 \ 12 \ 4 \ 18 \ 6 \ 2 \ 9 \ 3),$$

resulting in the error term being given by

$$(0 \ -28 \ -4 \ 0 \ -14 \ -2 \ 0 \ -7 \ -1).$$

If we represent the transformed initial environment vector as $\bar{V}_0^*$, and the transformed error vector as $\bar{e}^*$, then we can rewrite the general form of a nonlinear predicate interpretation as

$$0 \text{ .relap. } (\bar{T}C)(\bar{V}_0^*).$$

In order for an error in the calling program to produce a nonequivalent predicate interpretation along some path in the module, it is necessary for the following condition to hold (from Chapter III).

$$\begin{aligned} \forall k > 0, \forall \bar{e}^* \neq \bar{0}, \exists \bar{V}_0^* \text{ such that} \\ (\bar{T}C)(\bar{V}_0^* + \bar{e}^*) \neq \\ k(\bar{T}C)(\bar{V}_0^*) \end{aligned} \quad (9)$$

Condition (9) describes the conditions under which a nonlinear predicate interpretation in the module will produce a nonequivalent predicate interpretation if a nonzero error occurs in the input to the module. Furthermore, condition (9) is equivalent to condition (4) from Chapter III. From Chapter III and condition (9) we can now conclude

Lemma 6 Under the restrictions placed on a module by the nonlinear model, in order to ensure that an error in the calling program can produce a nonequivalent predicate interpretation along some path in the module, it is sufficient that there exist a predicate T in the module and a transformed input space spanning path leading to T

satisfying condition (9).

Generally no single predicate interpretation can meet condition (9). As in the linear case, if we can choose a set of d predicate interpretations (where d is the dimension of the vector space describing the predicate interpretation)

$$\{[(\bar{T}C)]_i\}_{i=1}^d$$

from transformed input space spanning paths such that the $d \times d$ matrix M whose i^{th} row consists of $[(\bar{T}C)]_i$ has a non-zero determinant, then

$$\begin{aligned} \forall k > 0, \forall \bar{e}^* \neq \bar{0}, \exists \bar{V}_0^* \text{ such that} \\ [(\bar{T}C)]_i(\bar{V}_0^* + \bar{e}^*) &\neq \\ k[(\bar{T}C)]_i(\bar{V}_0^*) & \\ \text{for all } i = 1, \dots, d. \end{aligned}$$

Finally we can now conclude

Theorem 2: Given a set of d predicate interpretations along transformed input space spanning paths through the correct module such that the set of interpretations satisfy the condition on M above, then any integration

time domain error is capable of being detected in the module by at least one of the predicate interpretations along the path.

The above theorem places an upper bound of d on the number of paths in a module containing nonlinear predicate interpretations which need be included in the integration test set for detecting integration domain errors. Again this result is important because the size of the integration test set is only dependent on the number of input variables to the module, and the complexity of the predicate interpretations in the module. The size of the integration test set is independent of the path complexity of the module.

As in the linear case it may be that no set of d predicate interpretations exist. If this is the case it is sufficient to choose paths containing predicate interpretations covering those directions in the transformed error space that can be detected by some predicate interpretation in the module.

From the above analysis we have shown that nonlinear predicate interpretations can be handled in the integration testing strategy if the predicate

interpretations can be described as a finite vector space over $\bar{V}$. The price that is paid for using the integration testing strategy on nonlinear predicate interpretations is the increased dimensionality of the vector spaces. Recall from Chapter III that in order to find the set of predicate interpretations M it is necessary to perform gaussian elimination on the partial M matrices. Since the complexity of gaussian elimination increases as the cube of the dimension of the vectors involved, the work required for finding the set of M predicate interpretations can become very large. Furthermore, the size of the integration test set is itself increased, in general, due to the higher dimensionality.

One factor that can contribute to the reduction of this increase in complexity comes from the zero terms associated with a particular predicate interpretation. When determining the dimension d of the predicate interpretations in a module it is only necessary to include those terms that are non-zero in at least one predicate interpretation in the module. In the example in Figure 4.1 there were nine terms in the vector describing the predicate interpretation. However, only three of the terms were non-zero. For this example there was only a single predicate interpretation. However, if other

predicates existed in the module it would still be possible that some of the terms in the vectors would be zero for all the predicate interpretations.

While it may be that these zero terms might not significantly reduce the complexity of performing the integration testing strategy on nonlinear predicates, it does indicate that the existence of a single complex term in a predicate interpretation (such as an input variable cubed) won't necessarily increase the complexity of the strategy significantly. As was noted earlier, most predicates (and computations) in a program tend to be simple. Given this it would be reasonable to assume that a module that contains nonlinear terms in its predicate interpretations would in general also be simple.

IV.3 Integration Computation Errors with Nonlinear Computations

The second part of the integration testing strategy is the detection of integration computation errors. As in the integration domain error case, the solutions obtained for the linear case can be expanded to include computations that are nonlinear combinations of the inputs.

to the module. The general form of the results of a set of computations along a particular path in a module can be represented by

$$\bar{V}_f = C[\bar{V}_0].$$

Under the restriction placed on the complexity of computations and predicates earlier in this chapter, C in this case can be represented by a $d \times q$ matrix. Recall that d is the dimension of the vector space needed to describe the computations in the module, and q is the number of output variables in the module.

If an error exists in the calling program resulting in an incorrect value being passed to the module, this can be represented by

$$\bar{V}'_0 = \bar{V}_0 + \bar{e} \quad \bar{e} \neq \bar{0}.$$

As before $\bar{e}$ can only be non-zero in positions 2 through $m+1$.

As in the integration domain error case the dimension of $\bar{V}_0$ will in general not be equivalent to d , the dimension of the C matrix. In order to use the methods of

Chapter III we need to transform the $\bar{V}_0$ vector to a vector of dimension d . Again we will represent the transformed initial environment vector by $\bar{V}_0^*$, and the transformed error vector as $\bar{e}^*$. Using these transformed vectors we can rewrite the results of the computations along a path in the module as

$$\bar{V}_f = c(\bar{V}_0^* + \bar{e}^*)$$

or

$$\bar{V}_f = c\bar{V}_0^* + c\bar{e}^*.$$

An error in the input to the module is only detectable in the outputs of the module when $c\bar{e}^*$ is detectable in the outputs of the module. Specifically, we are interested in choosing paths through the module such that

$$c\bar{e}^* \neq \bar{0}$$

for all possible values of $\bar{e}^*$.

Transmittal of an error in the input to the module to the module's outputs may not be sufficient to guarantee detection of the error in the outputs of the calling program. In order to provide this additional error

detection ability it is necessary to choose paths such that all possible transformed input errors are mapped onto all possible outputs.

As in the linear case, we can choose paths to perform the above mapping by considering the transformation space of the module. In Chapter III the transformation space of a module was defined to be a space of dimension $m \cdot q + 1$ where m is the number of inputs to the module, q is the number of outputs for the module, and the 1 represents the variables in the calling program that aren't passed to the module. In the nonlinear case the results of Chapter III still hold, but the number of inputs to the module (in the transformed input space) is given by d . The C matrix again provides the elements of the $d \cdot q + 1$ transformation vector for a particular path, with a single vector being provided by each path.

Since there are only $d \cdot q + 1$ linearly independent error directions in the transformation space for a module, we seek to choose a set of paths to construct a $d \cdot q + 1 \times d \cdot q + 1$ matrix M' with a non-zero determinant. Given that we can construct this M' matrix we can conclude the following.

Lemma 7 Examination of the output variables of the calling program for any test exercising those paths corresponding to the construction of M' as defined above is sufficient to detect any integration time computation error affecting the module.

If there aren't enough paths to generate a complete M' matrix, it is sufficient to choose paths containing computations so as to span as much of the transformation space of the module as can be spanned by any other set of paths in the module.

IV.4. Combined Integration Testing for Nonlinear Predicates and Computations

By combining the results for nonlinear integration domain, and integration computation errors we can derive a general case limit for the number of paths required in the integration test set for a particular module. That limit results from the addition of the number of paths required in the two cases. The upper bound would be:

$$(d) + (d*q+1).$$

All of the potential reductions in this upper bound that were possible in the linear case are also possible in this case.

From the above theoretical analysis it is clear that nonlinearity presents no major problems for the integration testing strategy if the form of nonlinearity can be expressed as a vector space over the input to the module. The primary difference between the two cases is that in the nonlinear case the potential size of the vectors we are dealing with is much larger. In the linear case the number of elements in the vector spaces describing the computation and predicates was limited by the number of inputs to the module. In the nonlinear case the number of elements in these vectors is limited only by the degree of nonlinearity allowed, which can be arbitrarily complex. Since the complexity of performing the integration testing strategy depends on the size of these vectors, and the number of paths in the integration test set may depend on the size of the vectors, the potential complexity of handling nonlinear predicates and computations is much greater than the linear case. It is important to note that even though the potential complexity is much greater, the actual complexity may not

be any greater.

V. Fundamental Limits of White Box Integration Testing

Any testing strategy is faced with fundamental limitations in its ability to detect certain types of errors. This is true because the general problem of testing a program to show correctness has been proven to be unsolvable. In the first two chapters we examined some of these limitations for particular strategies. Even with these limitations a testing strategy can be useful if the limitations only occur infrequently, or the strategy is still capable of detecting a wide class of errors even when the limitations occur.

In this chapter we will introduce some fundamental problems associated with the integration testing strategy. Our main focus here is to examine problems that arise due to the nature of the integration testing strategy that wouldn't have occurred had a complete path test been performed on the program and module. While many of the limitations of path testing (such as infeasibility

detection) still apply to the integration testing strategy these limits will not be discussed here.

The reduction that the integration testing strategy provides in the path testing complexity of a program is due to the ability of the strategy to select integration test paths for a module without examining the program (or programs) that call the module. If the selection of the integration test set were different for each program (or program path) which calls the module, then the amount of work required for integration testing could increase substantially. If in addition the selection of the integration test set required reexamining all paths in the module for each path in the calling program, this could result in the integration testing strategy requiring as much work as a complete path test.

The property of the integration testing strategy that allows it to select the integration test set without consideration of the calling program results in two basic limitations that wouldn't be a problem if a complete path test of the combined calling program and module were performed. These problems can be characterized as integration path infeasibility and predicate redundancy. The problem of integration path infeasibility affects the

ability to detect both integration domain and integration computation errors, while the predicate redundancy problem only affects the ability to detect integration domain errors.

V.1. Integration Path Infeasibility

The infeasibility problem in general has been shown to be unsolvable. However the problem here isn't the same as the general problem of finding a point to traverse a particular path. The integration path infeasibility problem arises due to the manner in which the integration testing strategy chooses the integration test set for a particular module. Because the integration testing strategy chooses integration test paths by only examining the module, the paths chosen for the integration test set should be feasible with respect to the module's input space. However a path might not be feasible with respect to a particular path in the calling program leading up to the module. This problem could degrade the error detection capabilities of the integration testing strategy if a path not chosen for the integration test set is feasible with respect to the calling program, and that path is capable of detecting the errors that would have

been detected by a path in the integration test set that isn't feasible with respect to the program.

Consider the example module MODULE8 (Figure 5.1). This module contains four paths, each of which are feasible with respect to the module's inputs.

Depending on the order in which the paths in the module are examined the integration test set for integration domain errors will vary. If the paths are

```

SUBROUTINE MODULE8 (A,B)
  IF A + B > 0
    IF A > 0
      .
      .
      .
    ELSE
      .
      .
      .
    ENDIF
  ELSE
    IF -2*A + 3*B < 0
      .
      .
      .
    ELSE
      .
      .
      .
    ENDIF
  ENDIF
END MODULE7

```

Figure 5.1
Module With Four Paths

examined in the order (THEN-THEN, THEN-ELSE, ELSE-THEN, ELSE-ELSE) then the integration test set for integration domain errors would be the single path THEN-THEN. This is because the two predicate interpretations along the THEN-THEN path span as much of the input error space as is spanned by any other paths in the module. It should be noted that any of the paths THEN-ELSE, ELSE-THEN and ELSE-ELSE are also capable of detecting all of the error directions that can be detected by any other path in the module. With respect to the module there is no difference in the error detection abilities of any of the possible integration test sets.

Now suppose the program in Figure 5.2 calls the module. Then the only paths through the module which are feasible with respect to the THEN-THEN path in the program are the paths ELSE-ELSE and ELSE-THEN in the module.

If the assignments of A and B in the program are incorrect, then that error will not be detectable on the path THEN-THEN in the module as that path is infeasible. If we had instead chosen the ELSE-THEN (or ELSE-ELSE) path in the module as the integration test set then an error in the assignment of A and B in the program would be detectable as a change in the interpretation of the

```

INPUT X,Y
IF X < -1
  IF Y < -2
    A = X
    B = Y
    CALL MODULE8 (A,B)
  ELSE
    .
    .
    .
  ENDIF
ELSE
  .
  .
  .
ENDIF
END PROGRAM

```

Figure 5.2
Program Calling Module MODULE8

predicate $-2*A + 3*B < 0$.

It should be noted that the selection of the ELSE-THEN path, while suitable in this case, doesn't solve the problem. If the program in Figure 5.2 were modified such that the first predicate $X < -1$ were changed to $X > -1$ and the second predicate $Y < -2$ were changed to $Y > 2$ then the ELSE-THEN path would be infeasible while the THEN-THEN path would now be feasible.

No matter which path is chosen as the integration test set it is possible to construct a program that calls

the module such that the path chosen is infeasible with respect to the program. This indicates that a solution to the problem of infeasible integration paths will have to rely on information from outside the module. This conflicts with the primary advantage of the integration testing strategy of only having to rely on the module to make the integration test set selection.

While this example dealt only with integration domain errors in the module, a similar infeasibility situation exists for integration computation errors. Consider the module in Figure 5.3. In order to detect integration computation errors it is necessary to select paths that detect all of the potential errors in the transformation space of the module. For this module the transformation space has dimension $1+4*1=5$, and the transformation vector for each of the four paths is as follows.

```

THEN-THEN (1 0 0 1 1)
THEN-ELSE (1 0 0 1 -1)
ELSE-THEN (1 0 0 -1 1)
ELSE-ELSE (1 0 0 -1 -1)

```

From the transformations vectors it is obvious that any three of the four paths in the module are sufficient

```

SUBROUTINE MODULE9 (A,B,C,D,O)
IF A > 0
  IF B > 0
    O = C + D
  ELSE
    O = C - D
  ENDIF
ELSE
  IF B > 0
    O = -C + D
  ELSE
    O = -C - D
  ENDIF
ENDIF
END MODULE9

```

Figure 5.3
Module With Four Inputs and One Output

to find errors in as much of this space as is possible. This is true since any single transformation vector can be expressed as a linear combination of the other three vectors. Since there is more than one choice for the integration test set the paths chosen for the set will vary depending on the order in which the paths are examined. If we examine the paths in the order (THEN-THEN, THEN-ELSE, ELSE-THEN, ELSE-ELSE) then the integration test set will consist of the paths (THEN-THEN, THEN-ELSE, ELSE-THEN).

If the program in Figure 5.4 is used to call MODULE9 then the possibility exists of missing certain integration

```

INPUT W,X,Y,Z
IF W + X < 0
    A = W
    B = X
    C = Y
    D = Z
    CALL MODULE9 (A,B,C,D,O)
ELSE
    .
    .
    .
ENDIF
PRINT O
END PROGRAM

```

Figure 5.4
Program Calling MODULE9

computation errors due to infeasibility. With respect to the calling program the THEN-THEN path of the module is infeasible. However if we had chosen the THEN-ELSE, ELSE-THEN and ELSE-ELSE paths for the integration test set each of those paths are feasible with respect to the calling program. Since the THEN-THEN path is infeasible with respect to the calling program an error which results in an incorrect value of C and D where the error in D is of the same magnitude and opposite sign as the error in C that error would be undetectable.

As in the case of the integration domain errors had we chosen the THEN-ELSE, ELSE-THEN and ELSE-ELSE paths another calling program could have been selected such that

the first path would have been feasible and one of the paths in the integration test set would have been infeasible. If the predicate $W + X < 0$ in Figure 5.4 were changed to $W + X > 0$ then the THEN-THEN path would now be feasible while the ELSE-ELSE path would now be infeasible.

One possible method of avoiding the problem of integration path infeasibility is to delay the selection of the integration test set until the actual integration testing is being done. Instead of performing the actual selection of the integration test set the integration testing strategy, while examining the module, would simply collect the necessary information on each path in the module to allow the selection to be made later. This would delay the selection of the integration paths until the domain restrictions, which result from the path through the program leading up to the module, are known. This effectively allows a solution of the integration infeasibility problem (up to the limits imposed by the unsolvability of the general infeasibility problem). The problem with this type of solution is that it could potentially require a new selection of the integration test set for each path leading up to the module. That selection of the integration test set might require examining each path in the module (or the information

collected for each path). If this were the case then the total complexity of integration testing is the same as the complexity of doing a complete path test of the program and module. Even if the selection of the integration test set doesn't require examining each path, it would at least greatly increase the total number of paths that would be examined in integration testing.

Since any solution to the integration infeasibility problem would require some amount of additional work at integration time, it is helpful to consider how big of a problem infeasible integration paths might be when testing "real world" programs. In the next chapter we will examine a program using the integration testing strategy. In that program infeasible integration paths do occur and their effect on the overall integration testing confidence will be discussed.

V.2. Predicate Redundancy

The second major limit of the integration testing strategy is that of predicate redundancy. This problem is similar in nature to the integration path infeasibility problem. However it only effects the ability to detect

integration domain errors. Predicate redundancy occurs when the path chosen for the integration test set is feasible with respect to the path in the calling program, but one or more of the predicates in the module are superseded by other predicates in the calling program. This causes a problem because the paths in the integration test set were chosen because certain predicates along those paths will shift for certain input errors to the module. Such a shift in a predicate implies that certain inputs will follow an incorrect path allowing the error to be detectable. If a predicate in the module has been superseded by a predicate in the calling program, then even though the predicate in the module will shift, the shift will not be detectable since no inputs will follow an incorrect path.

Consider again the module MODULE8 (Figure 5.1) containing 3 predicates and 4 paths. If we choose the path THEN-THEN through this module as the integration test set, and the following program (Figure 5.5) calls the module, certain errors in the calling program might be undetected. Specifically the predicate $A > 0$ in the module is superseded by the predicate $Y > 1$ in the calling program. This has the effect of preventing the testing of the predicate $A > 0$ by selecting test data that follows

the THEN-THEN path of the module as the predicate $A > 0$ isn't part of the final path domain.

The effects of the redundant predicate can best be seen by examining the domain structure of the program and module. For the true path in the program, and the THEN-THEN path in the module the following domain structure results (Figure 5.6). The predicate $A > 0$ in the module is represented by the line $X + Y > 0$. This predicate plays no part in the borders of the domain defining the path. If an error existed in the program that affected the interpretation of this predicate the error wouldn't be detectable (the paths borders wouldn't change). For example suppose that in the program the assignments for A and B should have been as follows.

```
A = X + 2*Y
B = X - Y
```

```
INPUT X,Y
IF Y > 1
  A = X + Y
  B = X
  CALL MODULE8 (A,B)
ENDIF
END PROGRAM
```

Figure 5.5
Program With Superseding Predicate

In this case the domain structure of the path is the same (Figure 5.7) with the exception of the changed predicate interpretation for $A > 0$ which doesn't contribute to the borders of the domain. The domain structure of both programs, in this case, are indistinguishable from one another, and the error in the program is undetectable. The problem is similar to the integration path infeasibility problem since the above error would have been detectable if the integration test set from the module had been the ELSE-THEN path.

On the surface the problem of predicate redundancy appears to be similar to the problem encountered with the code following the module losing an integration computation error (in Chapter III). In the integration computation error case we were able to avoid the problem by selecting paths that transmitted all possible input error directions to all possible output error directions. While this didn't guarantee transmission of all possible integration computation errors, it did guarantee transmission to the outputs of the program any integration computation error that could be transmitted along any of the paths in the module.

Figure 5.6
Domain Structure of the Incorrect Program Path

Figure 5.7
Domain Structure of the Correct Program Path

The case of predicate redundancy is fundamentally different in that a predicate interpretation in the module, instead of being a mapping from one vector space to another (i.e. from the inputs to the outputs), describes a mapping from the input vector space to a scalar. Since the integration test set for integration domain errors already spans the input vector space, the model of Chapter III provides no additional information as to how to choose predicate interpretations to avoid predicate redundancy. In this sense predicate redundancy is similar to the infeasibility problem presented in the previous section.

Since there is no information contained within a module to indicate that a particular predicate will be superseded, it is necessary to examine the predicates in the calling program to obtain that information. As in the infeasibility problem we again encounter the possibility of performing a complete path test of the program and module in order to avoid the predicate redundancy. This occurs because for any path in the calling program it may be necessary to examine each path in the module to determine which of the possible integration test sets don't involve redundancy.

As was stated earlier in this chapter, any finite testing strategy will be faced with fundamental limitations in its ability to detect certain types of errors. In this chapter we identified two types of errors that might be undetectable when applying the integration testing strategy as opposed to performing a complete path test.

The existence of fundamental limits to a testing strategy doesn't necessarily restrict the testing abilities of a strategy for "typical" types of programs. This approach has been used with other testing strategies to argue the effectiveness of the strategy even though the limitations exist. For the integration testing strategy it will be necessary to examine how often the limitations restrict the testing ability of the strategy to the point of rendering the strategy useless.

Even with these limitations the use of the integration testing strategy has some advantages over performing a complete path test. The primary advantage is the reduction in the number of paths that need to be examined. In the next chapter the size of this reduction will be demonstrated. In certain cases this path reduction

may make the difference between being able to adequately test the program, and having to perform some type of ad hoc testing.

In the next chapter we will examine the effects of predicate redundancy and integration path infeasibility on a sample program. While one sample can't indicate the frequency of occurrence and the resulting effect of the two problems on a wide class of programs, it will serve to illustrate the situations in which integration path infeasibility and predicate redundancy occur.

VI. An Example of the Integration Testing Strategy

The following example of the integration testing strategy is based on a production COBOL program. The program is part of a payroll system that has been in operation since 1979, and the particular program that is examined here calculates the gross pay and deductions for employees who are paid on a monthly basis. The program is written in standard COBOL for a WANG VS system.

The purpose of this example is to first study the path complexity of the program, and to compare that complexity to the complexity that results if the integration testing strategy is applied to the program. In addition the example will demonstrate some of the problems and solutions associated with implementing the integration testing strategy.

At the highest level the program consists of a single loop that is executed once for each employee whose pay is

to be calculated. For the purposes of this example this main loop will be ignored in both the original path complexity of the program and for the path complexity of the integration testing strategy. This is justifiable since the program collects no sums for all employees, and therefore the outer loop performs no computations and only contains a single predicate (namely loop until end of file). Inside this main loop the highest level module sequentially calls six modules (Figure 6.1). Because there

Figure 6.1
Flow of Control in Highest Level Module

is no other path complexity at this level, the highest level module consists of a single path.

The first module called is the GET-RECORD module. This module's function is to read an employee's time record from a file in order that the payroll can be calculated based on the time information. The GET-RECORD module also consists of a single path.

The second module, INITIALIZE-PAY-RECORD, sets up internal tables and initializes variables for the modules to follow. This module also contains a single path.

The third module is the CALCULATE-EARNINGS module. This module calls one lower level module, ADD-OTHER-EARNINGS. The purpose of these modules is to calculate the employee's gross pay from the hours worked and the hourly rate, and to calculate any other earnings that the employee might have had. This module also divides the earnings into taxable and nontaxable groups for use by later modules. The ADD-OTHER-EARNINGS module contains 7 paths (Figure 6.2), and the CALCULATE-EARNINGS module contains 21 paths (Figure 6.3).

Figure 6.2
ADD-OTHER-EARNINGS Module

Because the ADD-OTHER-EARNINGS module is called inside a loop in CALCULATE-EARNINGS the total path complexity of the combined modules is:

$$1 + \sum_{I=1}^5 7^I * 2 * 2 == 1 + 19607 * 2 * 2 == 78429 \text{ paths}$$

Figure 6.3
CALCULATE-EARNINGS Module

The fourth module called from the main module is CALCULATE-TAXES. This module in turn calls CALCULATE-FED-TAXES, CALCULATE-STATE-TAXES, and CALCULATE-CITY-TAXES. The purpose of the module is to calculate the applicable taxes based on the gross earnings figures calculated in the CALCULATE-EARNINGS module.

CALCULATE-FED-TAXES contains 84 separate paths (Figure 6.4). The CALCULATE-STATE-TAXES module calls a lower level module, CALCULATE-SIT. The CALCULATE-SIT module contains 120 paths (Figure 6.5), while the CALCULATE-STATE-TAXES module contains 4 paths (Figure 6.6). The combined path complexity of the two modules is:

$$2 + 2 * 120 == 242 \text{ paths.}$$

The final module called by the CALCULATE-TAXES module, CALCULATE-CITY-TAXES, contains 147 paths (Figure 6.7).

The CALCULATE-TAXES module contains 1008 paths (Figure 6.8).

Figure 6.4
CALCULATE-FED-TAXES Module

Figure 6.5
CALCULATE-SIT Module

Figure 6.6
CALCULATE-STATE-TAXES Module

Figure 6.7
CALCULATE-CITY-TAXES Module

Figure 6.8
CALCULATE-TAXES Module

The combined path complexity of the CALCULATE-TAXES module and it's submodules is:

$$36 + 972 * F * S * C == 36 + 972 * 84 * 242 * 147$$

$$== 2,904,545,988 \text{ paths}$$

The fifth module to be called by the highest level module is the CALCULATE-DEDUCTIONS module. The purpose of this module is to subtract from the gross pay all nontaxable deductions the employee might have. At the outer level CALCULATE-DEDUCTIONS contains a single loop that is executed up to ten times (Figure 6.9).

Figure 6.9
CALCULATE-DEDUCTIONS Module

Inside this loop the CALCULATE-DEDUCTIONS-A module is called. This module contains 293 separate paths (Figure 6.10).

The combined complexity of the CALCULATE-DEDUCTIONS modules is therefore given by:

$$\sum_{I=1}^{10} 293^I == 4,679 * 10^{24} \text{ paths}$$

The final module called by the main module is the WRITE-RECORD module. This module writes the results of the previous modules to a file. The WRITE-RECORD module contains 2 paths. The second path is an error exit in case the write to a file is unsuccessful.

Figure 6.10
CALCULATE-DEDUCTIONS-A Module

A summary of the path complexities of the individual modules is given in the following table.

1. GET-RECORD	1 path
2. INITIALIZE-PAY-RECORD	1 path
3. CALCULATE-EARNINGS	78429 paths
a. CALCULATE-EARNINGS	21 paths
b. ADD-OTHER-EARNINGS	7 paths
4. CALCULATE-TAXES	2,904,545,988 paths
a. CALCULATE-TAXES	1008 paths
b. CALCULATE-FED-TAXES	84 paths
c. CALCULATE-STATE-TAXES	242 paths
1) CALCULATE-STATE-TAXES	4 paths
2) CALCULATE-SIT	120 paths
d. CALCULATE-CITY-TAXES	147 paths
5. CALCULATE-DEDUCTIONS	4.679×10^{24} paths
a. CALCULATE-DEDUCTIONS	10 paths
b. CALCULATE-DEDUCTIONS-A	293 paths
6. WRITE-RECORD	2 paths
TOTAL FOR PROGRAM	2.132×10^{39} paths

Clearly the amount of work required to do a complete path testing of this program is unreasonably large. Even at the subsystem level the number of paths is beyond the

means of most path testing strategies. It is important to note that the number of paths in any one individual module isn't unreasonably large (the largest module contains 1008 paths). This indicates that a complete path testing of any one module could be performed with a reasonable amount of resources.

If the methods of the integration testing strategy are applied to this program it is hoped that the total path testing complexity of the program can be substantially reduced. At the subsystem level it is possible that the total path complexity will be reduced to the point that a complete path test of the subsystem is possible.

To see how the integration test set is determined for a single module we will examine in detail the selection process for the CALCULATE-FED-TAXES module. A listing of this module is included in Appendix A. The first step in finding the integration test set is to divide the module's variables into input and output variables. This breakdown is given in Table 6.1.

Input variables are those variables that are assumed to have a value on entry to the CALCULATE-FED-TAXES

I. Input Variables

1. REGULAR-GROSS
2. OVERTIME-GROSS
3. SICK-PAY
4. OVERTIME-SICK-PAY
5. VACATION-PAY
6. OVERTIME-VACATION-PAY
7. OTHER-EARNINGS
8. BONUS-WAGES
9. NOT-TAXED-WAGES
10. YTD-FICA-WAGES
11. FICA-LIMIT
12. FICA-RATE
13. FED-LOWER-LIMIT (1) through FED-LOWER-LIMIT (14)
14. FED-MINIMUM (1) through FED-MINIMUM (14)
15. FED-PERCENT (1) through FED-PERCENT (14).
16. MARITAL-STATUS

II. Output Variables

1. FEDERAL-WAGES
2. TAX-WITHHELD
3. FICA-WITHHELD
4. TAXABLE-WAGES
5. K

Table 6.1
List of Input and Output Variables
for CALCULATE-FED-TAXES

module, and output variables are those variables which are assigned a value by the CALCULATE-FED-TAXES module. Due to the nature of COBOL (i.e. all global variables), the method of determining the input and output variables isn't as easy as it would be in other languages with a formal parameter passing mechanism. In COBOL it would be necessary to perform a static analysis of the module to determine which variables are used before they are assigned.

The nature of the COBOL language introduces other problems with applying the integration testing strategy. Not all variables in a program (or module) are input or output variables. Specifically a module might contain variables that are only used as temporary storage in the confines of a particular module. In addition, programming style dictates that programmers should avoid the use of constants within the body of a program. To allow for easier modification of the program these constants should be given a variable name and assigned a value early in the program.

Ideally we would like to split the variables which assigned values into two groups, output variables and program variables. This is more than just a method for

reducing the complexity of the integration testing strategy as it is also necessary to avoid transforming integration computation errors to these program variables. This could be accomplished either through programmer interaction, or by making a more complete examination of the entire program to determine those output variables that aren't used in other modules before being reassigned.

We would also like to break the variables which are not assigned values by the module into two groups. These groups would be input variables and constant valued variables. The constant valued variables are intended to include those variables in a module that could have been programmed as constants but instead were coded as variables to facilitate changing the program. The two groups are similar in that both are variables, and both have a value assigned to them outside the scope of the module. However from a global viewpoint a constant valued variable will be assigned a constant value (not based on an input to the program) at some point in the program, and that value will not be changed during execution of the program.

Identification of these variables can be made either through programmer interaction or by enforcing certain

programming standards (i.e. in COBOL all variables that are assigned a value in working storage or in an initialization module are defined to be constant valued variables for the program).

In this example both program variables and constant valued variables occur. In the CALCULATE-FED-TAXES module the variables K and TAXABLE-WAGES are used as program variables, and aren't used outside the module without being assigned a new value. In addition the tax tables that are used in the module are assigned a value in working storage, and are never reassigned in the program. These variables can therefore be considered to be constant valued variables.

Table 6.2 shows how the variables in the CALCULATE-FED-TAXES module are divided into these four groups using the rules described above.

The CALCULATE-FED-TAXES module has 84 paths (all of which are feasible) (see Figure 6.4). Determining the integration test set for this module requires examining all of the possible paths, and the error directions which each path covers. If we assume that all of the modules paths are to be tested during unit testing, then the

I. Input Variables

1. REGULAR-GROSS
2. OVERTIME-GROSS
3. SICK-PAY
4. OVERTIME-SICK-PAY
5. VACATION-PAY
6. OVERTIME-VACATION-PAY
7. OTHER-EARNINGS
8. BONUS-WAGES
9. NOT-TAXED-WAGES
10. YTD-FICA-WAGES
11. MARITAL-STATUS

II. Output Variables

1. FEDERAL-WAGES
2. TAX-WITHHELD
3. FICA-WITHHELD

III. Program Variables

1. TAXABLE-WAGES
2. K

IV. Constant Valued Variables

1. FICA-LIMIT
2. FICA-RATE
3. FED-LOWER-LIMIT (1) through FED-LOWER-LIMIT (14)
4. FED-MINIMUM (1) through FED-MINIMUM (14)
5. FED-PERCENT (1) through FED-PERCENT (14).

Table 6.2
Breakdown of Input and Output Variables
for CALCULATE-FED-TAXES

integration test set selection can be done while the module is being unit tested. To determine the error directions which each path covers, a symbolic execution can be done for each path. If we also assume that this symbolic execution would have been done as part of the unit testing of the module (as in Domain Testing [White 80]) then the added complexity of determining the integration test set is minimal. This will be discussed in more detail after the procedure is more completely examined.

In this example we will ignore the unit testing of the module and instead deal with the determination of the integration test set. However, it is assumed that the testing of the module is being done in parallel with this process. This example will also demonstrate the determination of the integration test sets for domain and computation errors separately. Normally this would be done at the same time, but the process is easier to see by examining each independently.

VI.1 An Example of Integration Domain Error Detection

As was described in Chapter III the order in which paths are examined in a module may affect which paths are selected for inclusion in the integration test set. In this example we will adopt the convention of always examining the True path before the False path for any branch. In addition when a loop is encountered the loop will be executed the maximum number of times first. While selection of the True path first is arbitrary, by first examining the longest paths through loops the number of predicate interpretations that are encountered in the paths being examined first can be increased. This occurs because each time the loop predicate is encountered a new predicate interpretation is generated. In Chapter III we proposed a heuristic which stated that the selection of paths with more predicate interpretation first can potentially reduce the size of the integration test set. This occurs because more predicates occur along the longer paths, and can potentially detect more error directions for each path that is included in the integration test set. As we shall see from the example to follow, the selection of these "longer" paths first will make a small

reduction in the size of the integration test.

The first path to examine in the module corresponds to (T,7,T,T) (true branch, seven times through the loop, true branch, true-branch), this yields the following set of path constraints (predicate interpretations).

"S"	0	0	0	0	0	0	0	0	0	0	0	1
c1	1	1	1	1	1	1	1	-1	-1	0	0	
c2	1	1	1	1	1	1	1	-1	-1	0	0	
c3	1	1	1	1	1	1	1	-1	-1	0	0	
c4	1	1	1	1	1	1	1	-1	-1	0	0	
c5	1	1	1	1	1	1	1	-1	-1	0	0	
c6	1	1	1	1	1	1	1	-1	-1	0	0	
c7	1	1	1	1	1	1	1	-1	-1	0	0	
0	0	0	0	0	0	0	0	1	0	0	0	
c8	1	1	0	0	1	1	1	0	0	1	0	

The first column represents the constant position, and the next 11 columns represent the input variables to the module in the order listed previously. The constants c1 through c8 represent the different constants used in the predicates. Typically, the actual value of these constants would be used instead of a symbolic value, however in this example the use of the symbolic values will better illustrate the role of the constants in the strategy.

The constant "S" corresponds to the predicate

IF MARITAL-STATUS = "S" THEN.

The occurrence of a character type variable and constant will not affect the outcome of the strategy in this case (as can be seen by the following procedures). However, in general character variables are not considered by the integration testing strategy.

The first step in the integration testing strategy is to perform gaussian elimination on the above matrix to determine those linearly independent error directions that the path is capable of detecting. First we can subtract the second row from rows 3 through 8 resulting in

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ c1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ c2-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c3-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c4-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c5-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c6-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c7-c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ c8 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

By dividing through by the constants in row 3 through 8

and row 10 the result is

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \text{c1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1/c8 & 1/c8 & 0 & 0 & 1/c8 & 1/c8 & 1/c8 & 0 & 0 & 1/c8 & 0 \end{bmatrix}$$

which can be reduced to

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \text{c1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1/c8 & 1/c8 & 0 & 0 & 1/c8 & 1/c8 & 1/c8 & 0 & 0 & 1/c8 & 0 \end{bmatrix}.$$

Next we can multiple rows 3 and 5 by c1 and subtract from row 2 yielding

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \text{c1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1-c9 & 1-c9 & 1 & 1 & 1-c9 & 1-c9 & 1-c9 & -1 & -1 & -c9 & 0 \end{bmatrix}$$

where c9 is c1/c8. Finally we divide row 5 by 1-c9 and subtract from row 3. Row 5 is then reduced by dividing through by the constant term resulting in

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \text{c1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 \end{bmatrix}.$$

At this point the matrix can't be reduced further. Of the 11 possible error directions in this module the first path is capable of detecting 5 of these directions (from the 5 linearly independent rows). The above matrix also describes the nature of the 5 error directions that are detectable. For example, row 3 of the matrix is capable of detecting errors in the TAXABLE-WAGES program variable where TAXABLE-WAGES is defined in terms of input variables as

$$\begin{aligned} \text{TAXABLE-WAGES} = & \text{REGULAR-GROSS} + \text{OVERTIME-GROSS} + \\ & \text{SICK-PAY} + \text{OVERTIME-SICK-PAY} + \\ & \text{VACATION-PAY} + \text{OVERTIME-VACATION-PAY} + \\ & \text{OTHER-EARNINGS} - \text{BONUS-WAGES} - \\ & \text{NOT-TAXED-WAGES}. \end{aligned}$$

If an error occurs in one of the input variables that define TAXABLE-WAGES then that error will be detected. If an error occurs in more than one of the input variables defining TAXABLE-WAGES then that error will be detectable only if the combination of the errors are not canceled in the above expression. However, row 3 isn't capable of

detecting errors that result in the TAXABLE-WAGES predicate interpretation evaluating to be a multiple of itself. This type of error is detected by row 2 of the matrix where the value of the constant term is non-zero.

Even though the matrix describes the types of errors that are detectable by the paths in the integration test set, there isn't always an obvious relation between the matrix and the original predicate interpretations in the program code. This occurs because the individual terms of the rows of the matrix are transformed by gaussian elimination.

The next path to examine is the path corresponding to (T,7,T,F,T). The predicate interpretations along this path give the following matrix

[illegible]

The usual procedure would be to append the above 11 rows to the 5 rows generated from the first path, and perform gaussian elimination again. If more than five rows were left after the gaussian elimination then this new path would be added to the integration test set. If gaussian elimination reduced the matrix back to the original 5 rows then the new path would not be included in the integration test set. By inspection we note that the first 10 rows (or 10 predicates) from the second path are identical to the 10 rows from the first path, and therefore can't contribute any new error detection ability. This leaves the eleventh row as the only predicate interpretation which needs to be examined. If we append this row to the 5 rows from before the result is

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \text{c1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 \\ \text{c8} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Performing gaussian elimination on this matrix reduces it to the following matrix

$$\begin{bmatrix} \text{"S"} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ c1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Since there are now 6 linearly independent rows this new path detects an error direction that wasn't detected by the first path. The integration test set therefore now consists of the paths (T,7,T,T) and (T,7,T,F,T). The additional error that is now detectable that wasn't detectable along the (T,7,T,T) path is an error in the input variable YTD-FICA-WAGES. Along the (T,7,T,T) path an error in this variable was only detectable if a canceling error didn't occur in one of the variables SICK-PAY, OVERTIME-SICK-PAY, BONUS-WAGES, or NOT-TAXED-WAGES. The addition of the (T,7,T,F,T) path to the integration test set now allows any error in YTD-FICA-WAGES to be detected.

At this point the effect of the order in which paths are chosen can be illustrated. If we had first looked at the (T,7,T,F,T) path then the (T,7,T,T) path would have added no new rows to the reduced matrix. Given this different path selection order the integration test set would consist of only a single path. If we had examined

the path (T,1,T,T) first (one pass through the loop initially) then the integration test set would have contained one extra path. This occurs because the path with one pass through the loop would have generated the second row in the above matrix. However, row 3 of the above matrix wouldn't have occurred until a second pass through the loop was taken. As was discussed in Chapter III it is generally not feasible to try and determine the smallest possible integration test set. Therefore, in this example, we will adhere to the original path selection ordering.

The next path to consider is (T,7,T,F,F). This path produces the same predicate interpretation matrix as the (T,7,T,F,T) path, and therefore can add nothing new to the reduced matrix.

So far we have examined 3 of the 84 possible paths in this module. From the program listing (in appendix A) we can see that the remaining 81 paths can also add nothing new to the integration test set (this is true since all of the predicate interpretations that can be encountered in the module have already been encountered). The result is an integration test set for integration domain errors in this module consisting of two paths.

In this example examination of the module allowed us to use short cuts to deal with all the paths in the module. Typically in an automated system this would not be possible without some type of programmer (or tester) intervention to guide the integration test set selection. When programmer intervention is allowed the possibility of introducing errors in the test set selection process is increased. However, the possibility for reducing the amount of work required is potentially large (as illustrated in this example).

In an actual implementation of the integration testing strategy the next step would be to determine the integration test set for integration computation errors for this module. However, for the purposes of this example, we will delay the examination of integration computation errors, and will instead generate the integration test set for integration domain errors for the remaining modules in the program. For the lowest level modules the strategy is applied in exactly the same method as for the CALCULATE-FED-TAXES module. For the higher level modules it is necessary to examine all of the paths in these modules combined with all of the paths in the integration test set of the lower level modules to

determine the integration test set. We continue in this "bottom up" manner until the highest level is reached.

The number of paths in the integration test sets for each module in this example program is given in Table 6.3. For the higher level modules the number of paths in the integration test set includes the paths in the integration test sets of the lower level modules.

The numbers in Table 6.3 are somewhat misleading as they don't reflect the number of paths that had to be examined in order to find the integration test set for each module. At the lowest level we needed to examine all of the possible paths, and for the higher level modules we needed to examine all of the paths in that module combined with the paths in the integration test set from the lower level modules. Assuming a system that required examining each of these paths (i.e. assume no programmer intervention to reduce the amount of work required) Table 6.4 gives the number of paths that were examined to find the integration test sets. The number of paths listed here is also the numbers of paths that would be tested in order to perform a complete path test of the program using the integration testing strategy.

<u>Module Name</u>	<u>Integration Test Set</u>
1. GET-RECORD	1 path
2. INITIALIZE-PAY-RECORD	1 path
3. CALCULATE-EARNINGS	3 path
a. CALCULATE-EARNINGS	2 path
b. ADD-OTHER-EARNINGS	2 path
4. CALCULATE-TAXES	13 paths
a. CALCULATE-TAXES	2 paths
b. CALCULATE-FED-TAXES	2 paths
c. CALCULATE-STATE-TAXES	2 paths
1) CALCULATE-STATE-TAXES	2 paths
2) CALCULATE-SIT	1 path
d. CALCULATE-CITY-TAXES	3 paths
5. CALCULATE-DEDUCTIONS	4 paths
a. CALCULATE-DEDUCTIONS	1 path
b. CALCULATE-DEDUCTIONS-A	4 paths
6. WRITE-RECORD	1 path
TOTAL FOR PROGRAM	156 paths

Table 6.3
Domain Error Integration Test Set

<u>Module Name</u>	<u>Paths Examined</u>
1. GET-RECORD	1 path
2. INITIALIZE-PAY-RECORD	1 path
3. CALCULATE-EARNINGS	256 paths
a. CALCULATE-EARNINGS	249 paths
b. ADD-OTHER-EARNINGS	7 paths
4. CALCULATE-TAXES	12,055 paths
a. CALCULATE-TAXES	11,700 paths
b. CALCULATE-FED-TAXES	84 paths
c. CALCULATE-STATE-TAXES	124 paths
1) CALCULATE-STATE-TAXES	4 paths
2) CALCULATE-SIT	120 paths
d. CALCULATE-CITY-TAXES	147 paths
5. CALCULATE-DEDUCTIONS	633 paths
a. CALCULATE-DEDUCTIONS	340 paths
b. CALCULATE-DEDUCTIONS-A	293 paths
6. WRITE-RECORD	1 path
TOTAL FOR PROGRAM	13,103 paths

Table 6.4
Domain Error Paths Examined

If we examine Table 6.4 for the CALCULATE-EARNINGS modules, we note that in order to determine that there are only 2 paths in the integration test set for the ADD-OTHER-EARNINGS module we needed to examine all 7 paths. When examining the CALCULATE-EARNINGS module to determine its integration test set, we would need to examine all of the possible paths in this module (21 paths). Since 20 of these paths involve a call to ADD-OTHER-EARNINGS (see Figure 6.11), these 20 paths must incorporate the 2 paths in the integration test set of the ADD-OTHER-EARNINGS module (as shown in Figure 6.2).

The total number of paths in the flowgraph of Figure 6.11, and hence the number of paths needed to determine the integration test set for CALCULATE-EARNINGS, is 249. By adding the number of paths that we had to examine for each module, the total number of paths that needed to be examined to determine the integration test set for the CALCULATE-EARNINGS subsystem is 256 paths.

These numbers represent a substantial reduction in the original path complexity of the program (from 10^{39} to 10^4). Of course, it may be the case that the resulting numbers may still be too large to allow testing with a

Figure 6.11
CALCULATE-EARNINGS Module Integration Paths

reasonable amount of effort. Two points can be made to argue the effectiveness of the strategy even with the resulting path complexities. First, if we consider individual subsystems in the program, some subsystems that before were too complex to allow path testing should now be amenable to complete testing using the integration testing strategy. Specifically the CALCULATE-EARNINGS and CALCULATE-DEDUCTIONS modules could each be completely tested by examining less than 1000 paths. Second, even when a complete path test is still out of reach due to path complexity, the strategy provides direction as to which subset of paths should be tested. Instead of selecting random test paths from all of the paths in the program, the integration testing strategy provides guidance as to which paths to select in order to avoid duplicating testing effort.

Another solution to the resulting path complexity of the integration testing strategy is suggested by the example. At each step it was possible to reduce the total work of selecting the integration test set by having the programmer (or tester) examine the modules for obvious shortcuts. In the CALCULATE-FED-TAXES (and the other modules in this example) this provided a substantial reduction in the work required. While no firm figures can

be given for the potential reductions which this programmer intervention provides, it may be sufficient to allow the strategy to be applied a wider class of programs.

VI.2 Affects of the Fundamental Limits on the Testing of the Example Program

Chapter V discussed a number of fundamental limitations of the integration testing strategy. At this time we can examine the effect of these limitations on the testing of the example program. In the example, path infeasibility is rather common at the program level (even though it is doesn't exist at any of the lowest level modules). The most common cause of path infeasibility comes from the interaction of the CALCULATE-FED-TAXES and CALCULATE-STATE-TAXES modules. Since both modules bracket the employee's income level, the path chosen through the CALCULATE-FED-TAXES module determines to a large extent the path that must be taken through the CALCULATE-STATE-TAXES module. The potential for path infeasibility in the integration test sets for these two modules occurs because of this inherent infeasibility between the modules. (In general path infeasibility can

only occur in the integration test sets if infeasibility already existed in the original programs paths.) However, because of the way the individual modules were written the integration test sets for both modules chose paths that bracket income at the highest level, thereby avoiding the problem.

The other fundamental limitation of the integration testing strategy was that of predicate redundancy. This problem also occurs in this example in the CALCULATE-TAXES modules. Again the bracketing of the income allows the predicates in the CALCULATE-STATE-TAXES module to be superseded by the predicates in the CALCULATE-FED-TAXES module. This prevents those superseded predicates from detecting integration domain errors. Even though predicate redundancy occurs in this example it doesn't reduce the testing effectiveness of the strategy since the error that would have been detected by the superseded predicates are instead detected by the superseding predicates. For example, the CALCULATE-STATE-TAXES module may not detect an error in the REGULAR-GROSS variable due to the predicate

TAXABLE-WAGES < FED-LOWER-LIMIT (K)

in the CALCULATE-FED-TAXES module superseding it. However, the error in REGULAR-GROSS would have been detected by this predicate in the CALCULATE-FED-TAXES module.

The above arguments aren't intended to say that the fundamental limits introduced in Chapter V don't normally affect the testing ability of the integration testing strategy. However, the example demonstrates that even though fundamental limitations occur in the testing of a program they don't necessarily detract from the error detection abilities of the integration test sets.

Nonlinear predicates also occurred in the example program, but were limited to a single module (the CALCULATE-EARNINGS module). Even though this nonlinear predicate could have been handled using the techniques of Chapter IV, in this example it was simpler to include the nonlinear predicate in the integration test set without trying to determine the types of errors it would detect. This had the effect of increasing the number of paths in the integration test set of the CALCULATE-EARNINGS module by 1.

VI.3 An Example of Integration Computation Error Detection

As was done for the integration domain error case, we will examine the CALCULATE-FED-TAXES module in detail in order to determine the integration test set for integration computation errors. The order in which we will examine the paths in the module will be the same as in the previous section. However, it should be noted that the order in which paths are chosen won't affect the size of the integration test set. This is true since each path can only contribute a single vector to the transformation space of the module.

The first path is (T,7,T,T), and a symbolic execution of this path would generate the following C'(3,1) matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ c1 & c1 & c1 & c1 & c1 & c1 & c1 & .2-c1 & -c1 & 0 & 0 \\ c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each row represents the interpretation of one of the three output variables (in the order presented in Table 6.2). Using the above C'(3,1) to produce a transformation vector for the path results in

[illegible]

Since there is only a single vector in the transformation space matrix (and the vector is non-zero) this first path must be included in the integration test set for the module. The second path through the module corresponding to (T,7,T,F,T) generates the following $C'(3,1)$ matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ c_1 & c_1 & c_1 & c_1 & c_1 & c_1 & c_1 & .2 - c_1 & -c_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Using this matrix to produce a transformation vector, and appending that vector to the transformation vector from the first path produces the transformation matrix

[illegible]

Performing Gaussian elimination on this matrix produces

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & c_1 & c_1 & c_1 & c_1 & c_1 & c_1 & c_1 & .2-c_1 & -c_1 & 0 & 0 & c_2 & c_2 & 0 & 0 & c_2 & c_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & c_2 & c_2 & 0 & 0 & c_2 & c_2 & c_2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since the transformation matrix consists of two rows the path (T,7,T,F,T) should also be included in the integration test set.

The third path to consider is (T,7,T,F,F) generating the $C'(3,1)$ matrix

[illegible]

When this transformation is appended to the previous transformation vector and reduced with gaussian elimination the following matrix results.

[illegible]

The transformation vector from the (T,7,T,F,F) path is also linearly independent of the other vectors in the transformation matrix and therefore at this point there are three paths in the integration test set.

The next path in the module is (T,7,F,T) with the following C'(3,1) matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ c1 & c1 & c1 & c1 & c1 & c1 & c1 & -c1 & -c1 & 0 & 0 \\ -c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Appending this transformation vector to the previous transformation matrix and performing gaussian elimination results in

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & c1 & c1 & c1 & c1 & c1 & c1 & .2-c1 & -c1 & 0 & 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 \\ 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 \\ 0 & -c3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Again the new vector is linearly independent of the previous vectors and is included in the integration test set.

The (T,7,F,F,T) path produces the C'(3,1) matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ c1 & c1 & c1 & c1 & c1 & c1 & c1 & -c1 & -c1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Appending this to the above transformation matrix results in

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & c1 & c1 & c1 & c1 & c1 & c1 & c1 & .2-c1 & -c1 & 0 & 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 & 0 \\ 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -c3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & c1 & c1 & c1 & c1 & c1 & c1 & c1 & -c1 & -c1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

When this is reduced with gaussian elimination the last row is reduced to the 0 vector and can be dropped from the transformation matrix. The result is the following matrix.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & c1 & c1 & c1 & c1 & c1 & c1 & c1 & .2-c1 & -c1 & 0 & 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 & 0 \\ 0 & c2 & c2 & 0 & 0 & c2 & c2 & c2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -c3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 \end{bmatrix}$$

After examination of the first 5 paths through the module the integration test set consists of the 4 paths (T,7,T,T), (T,7,T,F,T), (T,7,T,F,F), and (T,7,F,T). The

remaining 79 paths in the CALCULATE-FED-TAXES module add no new linearly independent transformation vectors to the transformation matrix, and therefore need not be included in the integration test set. Again this is obvious by examination of the module as the remaining 79 paths only generate computations which have already been encountered along the first 5 paths.

The results of applying the integration testing strategy to the remaining modules in the program in a bottom up manner are outlined in Table 6.5. Table 6.6 gives the number of paths that would need to be examined (if no programmer intervention was used) to determine the integration test set. The numbers in this case are considerably higher than those for the integration domain error case. This is probably due in part to the increase in the size of the vectors involved, and the fact that each path can only contribute one vector to the transformation vector instead of multiple vectors as in the integration domain error case.

Even so, these numbers represent a substantial improvement over a complete path test of the program. However, if the integration testing strategy is used to detect integration computation errors the amount of work

<u>Module Name</u>	<u>Integration Test Set</u>
1. GET-RECORD	1 path
2. INITIALIZE-PAY-RECORD	1 path
3. CALCULATE-EARNINGS	6 paths
a. CALCULATE-EARNINGS	2 paths
b. ADD-OTHER-EARNINGS	5 paths
4. CALCULATE-TAXES	49 paths
a. CALCULATE-TAXES	3 paths
b. CALCULATE-FED-TAXES	4 paths
c. CALCULATE-STATE-TAXES	3 paths
1) CALCULATE-STATE-TAXES	2 paths
2) CALCULATE-SIT	2 paths
d. CALCULATE-CITY-TAXES	2 paths
5. CALCULATE-DEDUCTIONS	8 paths
a. CALCULATE-DEDUCTIONS	1 path
b. CALCULATE-DEDUCTIONS-A	8 paths
6. WRITE-RECORD	1 path
TOTAL FOR PROGRAM	2,352 paths

Table 6.5
Computation Error Integration Test Set

<u>Module Name</u>	<u>Paths Examined</u>
1. GET-RECORD	1 path
2. INITIALIZE-PAY-RECORD	1 path
3. CALCULATE-EARNINGS	15,628 paths
a. CALCULATE-EARNINGS	15,621 paths
b. ADD-OTHER-EARNINGS	7 paths
4. CALCULATE-TAXES	23,721 paths
a. CALCULATE-TAXES	23,364 paths
b. CALCULATE-FED-TAXES	84 paths
c. CALCULATE-STATE-TAXES	126 paths
1) CALCULATE-STATE-TAXES	6 paths
2) CALCULATE-SIT	120 paths
d. CALCULATE-CITY-TAXES	147 paths
5. CALCULATE-DEDUCTIONS	1,227,133,805 paths
a. CALCULATE-DEDUCTIONS	1,227,133,512 paths
b. CALCULATE-DEDUCTIONS-A	293 paths
6. WRITE-RECORD	1 path
TOTAL FOR PROGRAM	1,227,175,509 paths

Table 6.6
Computation Error Paths Examined

required without some type of programmer intervention is much too large to be reasonably applied. As in the integration domain error case the effectiveness of the strategy can still be argued based on the improvement over the original path complexity, and the potential for further improvement with some programmer intervention.

The fundamental limits of the integration testing strategy discussed in Chapter V don't affect the integration test set for integration computation errors. Even though infeasibility could have occurred between the CALCULATE-FED-TAXES and CALCULATE-STATE-TAXES modules (due to the inherent infeasibility between the modules), in the integration test set the infeasibility didn't occur because of the way in which the individual modules were written, and the order in which the paths were selected for examination.

VII. Summary and Conclusions

The existence of reliable program testing strategies is a relatively recent development in the field of software engineering. The problem with the strategies is often the complexity of applying them to reasonably sized programs. This research has sought to develop a method of testing which will have the error detection ability of the more reliable strategies, and yet will avoid many of the complexity issues these strategies face.

Many of the more reliable testing strategies rely on the examination of each path in the program being tested, yet the number of paths in a simple program can be prohibitively large. The use of modularity in program development helps break the development of a large complex task into smaller, more easily handled subtasks. The goal of the integration testing strategy is to use a program's inherent modularity to help reduce and manage testing complexity in the same way that it has been used to reduce

and manage software development complexity.

The technique for performing the integration testing strategy is based on choosing a subset of the total possible paths in a module. This subset should be capable of transmitting through the module all possible integration errors that could be transmitted by the module along any of its paths. This "integration test set" in effect characterizes the types of functions the module computes. This subset of paths is then used when testing higher level modules (instead of combining the higher level modules with all of the possible paths in the module).

By placing restrictions on the complexity of the allowable computations and predicates in a module, we have shown that the integration test set can be chosen at unit testing time by performing two gaussian eliminations for each path in the module. Furthermore, the number of paths in the integration test set is dependent only on the number of inputs, outputs, and the complexity of the computations and predicates in the module. This is particularly important since the size of the integration test set for a module is independent of the path complexity of the module. This uncoupling of the

integration testing strategy from the path complexity of the program that is being tested allows for substantial reductions in the complexity of testing the program.

Another way to look at the integration test set for a particular module is that it is a type of white box specification for the module. The integration test set completely describes for a particular module the nature of the computations and predicates in the module. The use of specifications in the testing of programs has been proposed by other authors. However, the advantage of the integration test set over more traditional specifications is that it can be generated from the module's code, and can easily be used in the path testing of higher level functions.

The ability of the integration test set to describe a particular module is based on the use of a vector space model of the module. The power of the model lies in its ability to describe an infinite number of functions and the errors they can detect, yet the description involves only a finite number of terms. It has been shown that vector spaces provide a useful method of describing the existing computations and predicate of a module (even if those computations or predicates are nonlinear).

Furthermore, if a module can be characterized with vector spaces, the integration test set can be derived through the use of common linear algebra techniques.

The use of vector spaces provides a rich method of describing the nature of a module. However, under certain conditions the vector space model isn't easily applied to computations and predicates. Specifically, many relatively simple functions (such as those involving division) can't be directly described using the vector space methods. However, many researchers have shown that for wide classes of programs the computations and predicates that are used mainly involve simple linear functions. Even if a program were to contain a relatively small number of paths containing complex computations or predicates (those that couldn't be described directly as a vector space), the integration testing strategy can take these paths into account without severely affecting the size of the integration test set. In addition, the nature of the vector space model allows those computations and predicates that can't be represented directly to be approximated to an arbitrary degree of accuracy using functions that can be described as a vector space. Although this may represent an unreasonable amount of work for many programs, it does serve to illustrate the

flexibility of the use of vectors for modeling the integration testing strategy.

The integration testing strategy is not without its limitations. The major drawbacks are the fundamental limits introduced in Chapter V. We know, however, that all testing strategies are faced with certain limits in their abilities to detect errors, and the integration testing strategy is no exception. Furthermore, testing in general will never be able to guarantee the correctness of arbitrary programs. The important point to consider is the way in which the limitations affect the ability of the strategy to detect the more "common" errors in a program. While the seriousness of the limitations in detracting from the error detection ability of a testing strategy depends to a large part on the program being tested, we have shown a "real world" example where the existence of the limitations didn't detract from the usefulness of the strategy.

Many open questions remain in the program testing area. Specifically, in order to allow the application of the integration testing strategy it has to be combined with some type of "reliable" path testing strategy to perform the actual testing of the chosen paths. Much

research needs to be performed to expand the scope of existing path testing strategies (such as Domain Testing) to allow them to be used on a wider class of programs. In order to avoid many of the fundamental limitations associated with any testing strategy it may be useful to examine the combining of testing with program proving techniques. This may allow the different approaches to potentially offset the limitations of the individual techniques.

The primary question with the integration testing strategy that remains unanswered is whether it sufficiently reduces the complexity of path testing strategies to allow them to be applied to a wide class of programs. The example in Chapter VI provided some insight into the answer of this question. When the integration testing strategy was applied to the program the resulting path complexity was substantially reduced over the original path complexity of the program. However, the number of paths that had to be examined in order to determine the integration test set for the program (and to test the program using the strategy) was still too large to be performed with a reasonable amount of resources. The hope is that through the use of heuristics and with the intervention of a programmer the complexity can be reduced

further.

The example in Chapter VI provided a basis for believing that further reductions are possible with this programmer intervention. This is not unreasonable, since no testing strategy can operate in a complete vacuum. At some point the people involved in the software development must to involved in the software testing. After all, the purpose of a testing strategy doesn't have to be to test a program automatically, but instead to provide the programmer with the necessary tools such that he might incorporate his knowledge to reliably test the program.

The ultimate usefulness of the integration testing strategy will only become apparent as it is applied to the testing of more programs. And while the examination of programs in a research environment is necessary to develop the strategy further, the real test of the strategies usefulness will come when it is applied in a "real world" program development environment.

The future of testing in software development is insured by its wide acceptance and almost universal application. However, without the continued development and wider application of theoretically sound testing

strategies the problems of software reliability in the software lifecycle may be with us for some time to come.

Appendix A.

CALCULATE-FED-TAXES Module Listing

```
00100  CALCULATE-FED-TAXES.
00200      COMPUTE FEDERAL-WAGES = REGULAR-GROSS +
00300      OVERTIME-GROSS +
00400      SICK-PAY +
00500      OVERTIME-SICK-PAY +
00600      VACATION-PAY +
00700      OVERTIME-VACATION-PAY +
00800      OTHER-EARNINGS.
00900  *
01000      MOVE FEDERAL-WAGES TO TAXABLE-WAGES.
01100      COMPUTE TAXABLE-WAGES = TAXABLE-WAGES -
01200      ( BONUS-WAGES + NOT-TAXED-WAGES ).
01300  *
01400      IF MARITAL-STATUS = "S" THEN
01500          PERFORM CALCULATE-FIT
01600              VARYING K FROM 1 BY 1 UNTIL K > 6 OR
01700              TAXABLE-WAGES < FED-LOWER-LIMIT (K)
01800      ELSE
01900          PERFORM CALCULATE-FIT
02000              VARYING K FROM 8 BY 1 UNTIL K > 13 OR
02100              TAXABLE-WAGES < FED-LOWER-LIMIT (K).
02200  *
02300      COMPUTE TAX-WITHHELD ROUNDED = FED-MINIMUM (K) +
02400      (( TAXABLE-WAGES - FED-LOWER-LIMIT (K) )
02500      * FED-PERCENT (K)).
02600  *
02700      IF BONUS-WAGES > 0 THEN
02800          COMPUTE TAX-WITHHELD = TAX-WITHHELD +
02900          ( BONUS-WAGES * .20 ).
03000  *
03100      MOVE FEDERAL-WAGES TO TAXABLE-WAGES.
03200      COMPUTE TAXABLE-WAGES = TAXABLE-WAGES -
03300      ( SICK-PAY + OVERTIME-SICK-PAY ).
```

```
03400 *
03500 *
03600 IF YTD-FICA-WAGES + TAXABLE-WAGES < FICA-LIMIT
03700 THEN
03800     COMPUTE FICA-WITHHELD ROUNDED = TAXABLE-WAGES
03900         * FICA-RATE
04000 ELSE
04100     IF YTD-FICA-WAGES NOT < FICA-LIMIT THEN
04200     THEN
04300         MOVE 0 TO FICA-WITHHELD
04400     ELSE
04500         COMPUTE FICA-WITHHELD ROUNDED =
04600             ( FICA-LIMIT - YTD-FICA-WAGES )
04700             * FICA-RATE.
04800 *
04900 CFT-EXIT.
05000 EXIT.
05100 *
05200 *
05300 *
05400 CALCULATE-FIT.
05500 *
05600 *
05700 *
```

BIBLIOGRAPHY

- Boehm 76 Boehm, B. W., "Software Engineering", IEEE Transactions on Computers, C-25, 12, Dec. 1976, 1226-1241
- Chow 78 Chow, T. S., "Testing Software Design Modeled by Finite-State Machines", IEEE Transactions on Software Engineering, vol. SE-4, no. 4, May 1978
- Clarke 76 Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, vol. SE-2, no. 3, Sept. 1976, 215-222
- Cohen 78 Cohen, E. I., "A Finite Domain-Testing Strategy for Computer Program Testing", Ph.D. Dissertation, 1978, The Ohio State University
- DeMillo 79 DeMillo, R. A., Sayward, F. G., and Lipton, R. J., "Program Mutation: A New Approach to Program Testing", Infotech State of the Art Report on Program Testing, 1979, Infotech International
- Gourlay 81 Gourlay, J. S., "Theory of Testing Computer Programs", Ph.D. Dissertation, 1981, The University of Michigan
- Howden 76 Howden, W., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, vol. SE-2, no. 3, Sept. 1976, 208-214

- Knuth 71 Knuth, D. E., "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, vol. 1, no. 2, April-June 1971, 105-133
- Parnas 72 Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, 15, 12, December 1972, 1053-1058
- Stevens 74 Stevens, W., Myers, G. and Constantine, L., "Structured Design", IBM Systems Journal, No. 2, 1974, 115-139
- Westley 79 Westley, A., ed., Infotech State of the Art Report on Software Testing, Vol. 1, Infotech International, 1979
- Weyuker 79 Weyuker, E. J., The Applicability of Program Schema Results to Programs, International Journal of Computer and Information Sciences, Vol. 8, No. 5, 1979, 387-403
- Weyuker 79 Weyuker, E. J., Translatability and Decidability Questions for Restricted Classes of Program Schemas, SIAM Journal on Computing, Vol. 8, No. 4, 1979, 587-598
- Weyuker 80 Weyuker, E. J., and Ostrand, T. J., "Theories of Program Testing and Application of Revealing Subdomains", IEEE Transactions on Software Engineering, vol. SE-6, no. 3, May 1980
- White 80 White, L. J. and Cohen, E. I., "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, vol. SE-6, no. 3, May 1980, 247-257
- Yourdon 79 Yourdon, E. and Constantine, L., Structured Design, Prentice Hall, Englewood Cliffs, N.J., 1979
- Zeil 81 Zeil, S. and White, L., "Sufficient Test Sets for Path Analysis Testing Strategies", Proceedings. 5th International Conference on Software Engineering, San Diego, CA. March 1981, 184-191

- Zweben 81 Zweben, S. H., "Computer Program Testing - An Introduction", Computer Program Testing, Chandrasekaran, B. and Radicchi, S. eds., North-Holland, New York, N.Y., 1981