

---

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

---

1991

## Error flow in computer programs

Branson Wayne Murrill

*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Murrill, Branson Wayne, "Error flow in computer programs" (1991). *Dissertations, Theses, and Masters Projects*. Paper 1539623805.

<https://dx.doi.org/doi:10.21220/s2-d0pc-8q38>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# U·M·I

University Microfilms International  
A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
313/761-4700 800/521-0600



**Order Number 9207759**

**Error flow in computer programs**

**Murrill, Branson W., Ph.D.**

**The College of William and Mary, 1991**

**U·M·I**  
300 N. Zeeb Rd.  
Ann Arbor, MI 48106



**ERROR FLOW IN COMPUTER PROGRAMS**

---

**A Dissertation**

**Presented to**

**The Faculty of the Department of Computer Science**

**The College of William and Mary in Virginia**

**In Partial Fulfillment**

**Of the Requirements for the Degree of**

**Doctor of Philosophy**

---

**by**


**Branson W. Murrill**

**1991**

Approval Sheet

This dissertation is submitted in partial fulfillment of  
the requirements for the degree of


Doctor of Philosophy

  
\_\_\_\_\_  
Author

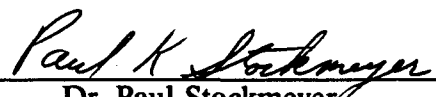
Approved, July 1991

  
\_\_\_\_\_  
Dr. Larry Morell (Advisor)

  
\_\_\_\_\_  
Dr. Susan Brilliant  
Virginia Commonwealth University

  
\_\_\_\_\_  
Dr. John Drew  
Department of Mathematics

  
\_\_\_\_\_  
Dr. Keith Miller

  
\_\_\_\_\_  
Dr. Paul Stockmeyer

## DEDICATION

I dedicate this work to my family for their constant love and support throughout this long process :

my wife, Linda,

who held the family together, took over many of my duties, and made many sacrifices. This achievement would not have been possible without her love, help, and perseverance.

my children, Jenny, Kathy, and Sara,

who reluctantly gave me up to my work on many occasions (but sometimes insisted on giving me a welcome respite), watched me work with interest and excitement, and played quietly while I worked (most of the time). If I have not seen as far as I might have, it is because midgits have sat upon my shoulders.

my father, Malcolm Murrill,

who introduced me to computers many years ago, and who by his example of teaching, scholarship, and service to others encouraged me to pursue a career in higher education.

my mother, Betty Murrill,

who has always had faith in my ability to achieve my goals.

my grandmother, Eva Hasher,

who has given me her unconditional love and affection all my life.

my brother, Bill Murrill,

who has followed my progress with interest and encouragement.



## TABLE OF CONTENTS

	Page
Dedication .....	iii
Acknowledgements .....	vi
Abstract .....	vii
Chapter 1. Introduction .....	2
1.1 Previous research in white box analysis and testing .....	7
1.1.1 Data flow analysis .....	8
1.1.2 Data flow testing .....	10
1.1.3 Error-based and fault-based testing .....	14
1.2 Deficiencies in existing white box strategies .....	18
1.3 A new approach - error flow analysis .....	22
1.4 Contributions of this thesis .....	26
1.5 Outline of this thesis .....	28
Chapter 2. Background theory and terminology .....	29
2.1 Mathematical functions .....	29
2.2 Flow graphs .....	31
2.3 Faults, errors, and coincidental correctness .....	33
Chapter 3. A functional view of programs and errors .....	38
3.1 The von Neumann model of computation .....	38
3.2 Function implementations, syntactic and semantic distance ..	41
3.3 Error sets and error traces .....	45

3.4	Coincidental correctness and functions computed by paths ..	47
Chapter 4.	Experiments in dynamic error flow analysis .....	57
4.1	The DEFA system .....	57
4.2	Potential uses for the DEFA system .....	66
4.3	Programs analyzed using the DEFA system .....	67
4.3.1	The Triangle program .....	68
4.3.2	The Digitseq program .....	77
4.3.3	The Cancel program .....	85
4.4	Comments on experimental results .....	93
4.5	Comments on dynamic error flow analysis .....	97
Chapter 5.	Estimating error flow behavior through static analysis .....	99
5.1	Static analysis techniques .....	99
5.2	The threshold model of error flow testing .....	106
5.3	An example of threshold testing .....	112
5.4	Discussion of the threshold model .....	118
Chapter 6.	Summary and future directions .....	121
Appendix	.....	125
Bibliography	.....	129
Vita	.....	132

## ACKNOWLEDGEMENTS

The author wishes to acknowledge the contributions of the following towards the completion of this work :

Larry Morell, for his guidance and friendship over the past few years, and for going above and beyond the call of duty to help me achieve this goal.

Committee members Sue Brilliant, John Drew, Keith Miller, and Paul Stockmeyer, for their service and helpful suggestions.

Chairman Dick Prosl, for accommodating my part-time status, and for financial support through the Department during the Spring, 1990 semester.

Dr. Bill Haver, Dr. Reuben Farley, and others at Virginia Commonwealth University for their friendship and their support over the past 5 years. The financial and release-time support from VCU was crucial to my success.

NASA grant NAG-1-884 for partial support in the summer of 1990 and a course reduction in the Fall, 1990 semester.

## ABSTRACT

White box program analysis has been applied to program testing for some time, but this analysis is primarily grounded in program syntax, while errors arise from incorrect program semantics. We introduce a semantically-based technique called *error flow analysis*, which is used to investigate the behavior of a program at the level of data state transitions. Error flow analysis is based on a model of program execution as a composition of functions that each map a prior data state into a subsequent data state. According to the fault/failure model, failure occurs when a fault causes an infection in the data state which then propagates to output. A faulty program may also produce coincidentally correct output for a given input if the fault resists infection, or an infection is cancelled by subsequent computation. We investigate this phenomenon using dynamic error flow analysis to track the infection and propagation of errors in the data states of programs with seeded faults. This information is gathered for a particular fault over many inputs on a path-by-path basis to estimate execution, infection, and failure rates as well as characteristics of error flow behavior for the fault. Those paths that exhibit high failure rates would be more desirable to test for this fault than those with lower failure rates, and we look for error flow characteristics that correlate with high failure rate. We present the results of dynamic error flow experiments on several programs, and suggest ways in which error flow information can be used in program analysis and testing.

## **ERROR FLOW IN COMPUTER PROGRAMS**

## 1. Introduction

Program verification, the activity of certifying that a program correctly implements its intended function, has always been a difficult issue. In fact, if we take this definition of verification literally, then we are doomed to disappointment. A computation produced by a computer is the end result of a multilevel process of abstraction. We say that the transformation of one abstraction into another abstraction is *semantic-preserving* if the meanings of the abstractions under their corresponding interpretations are equivalent. It is possible to completely verify that the transformations between some of the levels of abstraction are semantic-preserving. For example, we may verify that a circuit design correctly implements a boolean function, or that a language compiler produces correct machine code for statements in a higher-level language. Program verification attempts to show that the program implements some specification, and we may attempt formal verification of this specification against a higher-level specification. Ultimately, however, we must verify some concrete specification against the intent conceived in a human mind, and this is not susceptible to formal methods, but rather human beliefs or consensus.

With this caveat, we focus on program verification as the act of certifying that a program correctly implements a specification (which we will assume is correct).

An *oracle* is a decision procedure that determines whether or not a program's output is correct for a given input. Unfortunately, oracles rarely exist in practice, but they are useful in discussing verification methods.

There are four primary approaches to program verification :

- proof of correctness
- exhaustive testing
- random black box testing
- white box testing

Potential problems exist with each of these strategies. Testing strategies require an oracle to decide if the output is correct, and this is a common weakness of all testing strategies. Most previous non-exhaustive testing strategies also do not help us quantify our increased confidence after successful testing<sup>1</sup>. Let us briefly consider each strategy.

Formal proof of correctness [Hoa69] has been used as a verification technique for a number of years, mostly on small programs. It is tedious and difficult to apply to large, complex programs. Although portions of the process can benefit from automation, it relies substantially on the skills of the person(s) doing the proof. As in all proofs, correct reasoning applied to incorrect or insufficient assertions can lead

---

<sup>1</sup>A statistical model for estimating the probability of failure after successful testing is presented in [Mil90]

to erroneous conclusions, so assertions must be carefully chosen. Mistakes in the proof can also lead to an erroneous claim of program correctness. Unlike mathematical proofs, which are scrutinized by many mathematicians to verify both the assertions and the validity of the proof before general acceptance (still no guarantee, as history has shown), correctness proofs of programs are rarely independently verified by others due to the sheer effort involved, so the proof must be verified by other means. One reasonable approach is to test the program after doing a proof of correctness, each reinforcing the other. However, a subtle error in the proof might still remain unobserved after testing with a particular set of test data.

Exhaustive testing is usually intractable due to the size of the input domain, even though all actual input domains are finite due to the discrete representation of data in a computer. A simple example illustrates this point :

Consider a relatively simple computer program that takes three 32-bit integer inputs. There are  $8 \times 10^{28}$  input combinations in this program's input domain. Suppose there was a supercomputer that could execute this program in 1 microsecond (beyond current technology). Then if that computer began to exhaustively test this program at the instant the universe was created (about 15 billion years ago), as of today that computer would have tested less than .001% of the program's input domain.



There are some cases in which the program's input domain is sufficiently restricted so that exhaustive testing is feasible. It is also possible to partition the input domain of some programs into equivalence classes such that the successful test of one member from the class demonstrates the correct computation for all members of the class. This can only be done by proving certain properties of the program code, incurring the same problems associated with proof of correctness.

Random black box testing [Dur84] concentrates on the input/output behavior of a program while ignoring the internal code and computation, and attempts to imply reliability of a program after random testing of a small part of the program's input domain. The fact that the inputs are randomly chosen prevents a biased choice of test data, which can occur when the same mental mistake made in the programming also leads to a choice of test data that does not reveal the error. For example, a person writing a routine that uses trigonometric functions may have designed the program with only first quadrant points in mind and might thus generate test data based on first quadrant points, while the specification calls for the program to handle points from any quadrant. Another advantage of random testing is that random tests are easily chosen for programs with real or integer domains by the standard technique of pseudo-random number generation. Disadvantages are that after testing a small portion of the input domain and seeing no errors, we may not infer much confidence in absolute correctness, and we know nothing about the program's behavior with untested inputs. While we cannot achieve much confidence

in absolute correctness using random black box testing unless a large proportion of the input domain is successfully tested, we can obtain an estimate of the probability of failure after  $n$  successful random tests [Mil90]. Some faults cause program failure for many inputs and are thus easily caught by random testing, but some faults cause program failure on only a few inputs, and these are difficult to reveal through random testing.

White box testing strategies [Hua75, How86] use properties of a program's code to produce test data. This requires white box analysis techniques that discern these properties from syntactic and semantic analysis of the code. There is strong intuitive appeal for the idea that test data selected on the basis of knowledge of a program's code would be more effective at discovering errors than test data randomly selected. Indeed, this is exactly the way most programmers informally test their programs, by choosing data that will exercise different parts or aspects of their code. Most white box strategies are coverage-oriented, i.e., they attempt to cover some structural aspect of the program's code. The primary problem with most white box strategies is that they concentrate on syntactic aspects of the code rather than semantic aspects. Because coverage of syntactic aspects of a program does not incorporate possible semantic differences within the same syntactic aspect, the program may fail on untested inputs even though all syntactic aspects have been covered. White box analysis and testing strategies are a promising method of program verification and are discussed in more detail in the next section.

## 1.1 Previous research in white box analysis and testing

A number of program testing methods [Mil74] are based on the principle that unless certain aspects of the program's structure or code have been tested, testing is incomplete. These aspects are specified through *structural coverage criteria*. A coverage-based testing criterion is *covered* when the program is executed with test data that exercises (a minimum of once) each aspect of the program as described by the criterion. Structural testing essentially involves executing a subset of paths from the potentially infinite set of all possible execution paths. For example, the (control flow) criterion of *statement coverage* says that test data must be selected that will cause execution of every program statement, since we would have very little confidence in the correctness of a statement that had never been executed. A *fault* is a portion of program code (or lack thereof) that causes erroneous output for some input. A particular execution path is *error-revealing* if its execution will always detect an error if one is present. A testing criterion is *reliable* [Goo75] if it always selects test data that results in the execution of at least one error-revealing path. Some paths that contain faults do not always exhibit an error when executed, an example of a phenomenon called *coincidental correctness*<sup>2</sup>. Laski and Korel [Las83] call such paths *error-sensitive* and define a *viable* testing criterion as one that always selects test data that results in the execution of at least one error-sensitive path. Since a program may

---

<sup>2</sup>We use the commonly accepted definition of this term, not the original definition found in [Whi80], and discuss this concept in detail in Sections 2.3 and 3.4.

contain no error-revealing paths, no universally reliable structural testing criterion exists. As an example, consider the program

```
read (x)
y := x * 2      { * should have been + }
write (y)
```

There is only one path in the program and it is not error-revealing since testing with an input of 2 produces the correct (according to specification, oracle, etc.) result. This disappointing result early in the development of testing theory is discussed by Weyuker and Ostrand in [Wey80]. Although structural testing methods can never guarantee the absence of errors, they do provide a systematic and sensible framework for program testing. Such testing does in practice discover many programming errors.

### 1.1.1 Data flow analysis

Data flow analysis [Hec77, Fos76] involves finding the relationships between the binding of a value to a variable (called a variable *definition*) and other locations in the program that *use* or *reference* that particular definition. We say that a definition *reaches* a use along a given path if the path contains no intervening redefinition of that variable. Data flow algorithms traverse the control flow graph of

a program to determine which definitions reach which uses. This technique was originally used by compilers to perform global optimization during code generation. Two common optimizations are solutions to the live variable problem and the available expression problem. A variable's definition is *live* at a particular point in the program if there is at least one path continuing from that point that reaches a use of that definition. Optimization involves deallocating storage for any variable that is not live for the rest of the program, or avoiding an unnecessary assignment when the assigned value would never be used. The available expression problem involves determining when the previously computed and stored value of an expression can be used to save recomputation of the same value when the same expression is encountered at a subsequent point in execution. If there is no redefinition of any of the expression's variables after the first occurrence of the expression along all paths between the first and a subsequent occurrence, then the expression is *available*.

Fosdick and Osterweil [Fos76] discuss these problems and the data flow algorithms to solve them, and then suggest that data flow techniques using these algorithms can be applied to detect certain data flow anomalies that may imply programming anomalies. These data flow anomalies are described in terms of definitions (*d*), references (*r*), and undefinitions (*u*) : the ineffective assignment anomaly (*dd* or *du*) and the uninitialized variable anomaly (*ur*). Fosdick and Osterweil state that this static analysis can be added to a compiler at relatively little cost and describe DAVE, a system for performing this analysis on FORTRAN

programs. Korel [Kor87] also uses static data flow analysis in combination with control flow analysis to identify redundant code (code which has no effect on output) and to analyze the relationship between input and output variables.

### 1.1.2 Data flow testing

Although the above data flow anomalies account for a small portion of programming errors, other researchers in the testing community saw the potential for data flow analysis in developing coverage criteria for program testing. Laski and Korel [Las83] propose two coverage strategies based on a program's data flow. The guiding principle in these strategies is that every use of a variable must be tested with each of its live definitions, since an untested definition might be used incorrectly. Test data must be selected to execute a set of paths that will ensure coverage of all of these *definition-use (du) pairs*. This brings up the infeasible path problem : some *du* pairs are not possible because the corresponding execution path(s) are not executable. Determining in general through static analysis whether or not a path is executable is an undecidable problem, since we cannot in general determine if an arbitrary predicate guarding a path is ever true or ever false (consider the predicate  $a^n + b^n = c^n$  for  $n > 2$ ). In many cases, however, we can determine that a particular path is infeasible. We thus eliminate from consideration any *du* pair for which we find only infeasible paths.

Before describing their strategies, Laski and Korel give several definitions. The *input variables* of a statement are those variables whose values are used in the computation of the statement. The *output variables* are those variables defined by the statement. The *data environment* of a statement is the set of all live definitions of the statement's input variables. An *elementary data context* for a statement is a subset of its data environment that contains one live definition for each input variable, as defined by a particular execution path from the start of the program. The statement's *data context* is the set of all its elementary contexts. The two proposed testing strategies are then:

1. Cover the data environment of each statement in the program
2. Cover the data context of each statement in the program

Ntafos [Nta84] extends these strategies by observing that data flows along chains of definitions and uses, i.e. a definition is used in another definition, and that definition is used in another definition, etc. He calls these *k-dr interactions*<sup>3</sup>, where *k* is the number of definitions along the chain, and suggests a family of strategies called *required k-tuples*. When *k=2*, this is equivalent to Laski and Korel's data environment coverage. Higher values of *k* test for more complex data flow interactions and require increasingly more test cases. For comparison, consider the following program :

---

<sup>3</sup>Some authors use *r* for reference and others *u* for use; they are equivalent

```

read (b)
if P1 then
  a := 1
else
  a := -1
b := a * b
if P2 then
  c := b - 1
else
  c := b + 1

```

Test data that drives the paths  $\{P1=true, P2=true\}$ ,  $\{P1=true, P2=false\}$ , and  $\{P1=false, P2=true\}$  satisfies data context coverage since each use has been tested with each of its live definitions. However, a potential chain of data flow,  $\{P1=false, P2=false\}$ , has been missed. The required 3-tuples strategy would include this one.

Rapps and Weyuker [Rap85] define a family of data flow testing criteria and establish a hierarchy that compares their data flow criteria with each other and with some common control flow criteria. They distinguish between a variable used in a computation (i.e. assignment or output) and a variable used in a predicate, called a *c-use* and *p-use*, respectively. The hierarchy of criteria is partially ordered by an *inclusion* relation where  $C_1$  strictly includes  $C_2$  if a proper subset of the paths chosen under  $C_1$  also covers  $C_2$ . Weyuker further examines this hierarchy in [Wey84] and derives upper bounds on the number of test cases required for each criterion. This hierarchy demonstrates that the data flow criteria nicely fill the gap between the control flow criteria of all-edges, which is generally considered inadequate, and all-paths, which is generally unattainable. Zeil [Zei88] examines data flow and control



flow as general classes of criteria and compares their *selectivity*. He defines a criterion  $C_1$  to be more selective than a criterion  $C_2$  for some testing goal  $G$  if 1)  $G$  is true for all test sets chosen by  $C_1$  but false for some test sets chosen by  $C_2$ , or 2)  $G$  is always true for test sets chosen by both criteria, but  $C_1$  never produces more and sometimes produces fewer tests than  $C_2$ . Selectivity can thus be used as a basis for cost comparisons. Zeil shows that the class of data flow criteria are more selective than control flow criteria for certain testing goals. The hierarchy described by Rapps and Weyuker is further extended in [Cla89] to include the data flow criteria of Laski and Korel [Las83], and Ntafos [Nta84].

Data flow testing has several important advantages over earlier coverage-oriented strategies. First, the coverage criteria are based on intimate knowledge of a program's control structure and the relationship of variable definitions and references. Second, the problem of infinitely many possible execution paths due to looping is avoided because a loop needs to be iterated only a finite number of times (twice for a body of sequential code) to exercise all *dr* pairs in the loop body. Finally, the static analysis portion of data flow testing generally results in smaller test sets than those produced by earlier testing methods, making the costly dynamic portion of the testing procedure more efficient. There are some disadvantages, however. Selecting input to drive particular paths can be non-trivial, and data flow testing suffers from the same problem as other coverage-oriented strategies: just

because a path has been covered and produces the correct result for a particular input doesn't mean that it is free of faults.

### **1.1.3 Error-based and fault-based testing**

Error-based testing methods [DeM78, Mor81, Mor84, Mor87, Mor88, Mor90, Voa90, Wey80] approach testing from a different viewpoint than that of structural coverage methods. Rather than attempting to achieve some coverage goal and hoping that no more errors exist, error-based criteria attempt to demonstrate that certain classes of errors are in fact absent. After demonstrating the problems of finding revealing criteria, Weyuker and Ostrand [Wey80] take this approach in trying to partition the program input space into subdomains which are revealing for certain types of errors. If restricted in this manner, only one input value is required for testing, since "revealing" implies that values in the subdomain either all succeed or all fail. Finding revealing subdomains can be difficult. Weyuker and Ostrand use a combination of the partitions defined by paths and other properties from the program specification, algorithm, and data structures in an effort to discover revealing subdomains.

All testing methods count on being able to detect the presence of a fault by observing erroneous output. Recall that the absence of reliable structural criteria

means that two different input values can follow the same execution path, yet one produces the correct result and the other produces an incorrect result. This coincidental correctness for some input values is a fundamental problem in program testing. The problem is acknowledged by most researchers and has been investigated in [Mor81] and [Mor88]. Morell and Hamlet [Mor81] investigate a fault/failure model of the creation and propagation of errors, and identify three necessary and sufficient conditions for a fault to cause an error:

1. Execution - the fault must be executed
2. Infection - execution of the fault must result in the introduction of an incorrect value into the subsequent data state
3. Propagation - the error in the data state must propagate along the execution path and be observed as erroneous output

Figure 1 illustrates these conditions with a simple example.

Morell [Mor88, Mor90] uses an approach to error-based testing that he classifies as *fault-based testing*. The difference is that fault-based testing uses a syntactic description of certain faults in the code, while error-based testing may include other descriptions of faults, as noted above. Using the fault/failure model, Morell has developed a testing method called *symbolic testing*. In this method

Figure 1. A simple illustration of a fault and its execution, infection, and propagation behavior under different inputs.

```

read (n)
i := 1
sum := 0
while i <= n do
  begin
    sum := sum + i
    i := i * 2           ( should be i := i + 1 )
  end
write (sum)

```

n	fault		correct		error?	effect of fault
	i	sum	i	sum		
0	1	0	1	0	no	$\sim E$
1	1	0	1	0	no	E and $\sim I$
	2	1	2	1		
2	1	0	1	0	no	E and I and $\sim P$
	2	1	2	1		
	4	3	3	3		
3	1	0	1	0	yes	E and I and P
	2	1	2	1		
	4	3	3	3		
			4	6		

*symbolic faults* are inserted into the code, which is then *symbolically executed* with real and/or *symbolic input*. Infection occurs when the symbolic fault is executed and causes a *symbolic error* to enter the data state. The ensuing symbolic execution traces the effect of the symbolic error as it propagates. The symbolic output and the actual output are equated to form a *propagation equation*, which allows the determination of values that, if substituted into the data state at the analysis location, would produce coincidental correctness. These values are then used to identify a class of faults that would produce the values and hence go undetected by this symbolic execution.

Mutation testing [DeM78] is similar in some ways to fault-based testing. Mutant programs with single faults are created by perturbing the program syntax according to a set of mutation operators, such as an expression value off by one or a variable name substitution. The mutants are then executed with test data to see if they produce different results from the original program (called "killing the mutant"). If any mutants remain after a sufficient amount of testing, an attempt is made to show that they are equivalent to the original program; otherwise more executions are required to kill them. The goal of mutation testing is to produce a set of test data that kills all simple mutant faults from a mutation set. A fundamental assumption of mutation testing, called the *coupling effect*, is that the test data developed in killing the simple mutant faults will also detect the presence of more complex faults.

Voas [Voa90] has investigated a statistical method for analyzing software that he calls *fault sensitivity analysis*. Using a fixed input distribution, estimates for execution, infection, and propagation are produced for each location in a program. An execution estimate predicts the probability that the location is reached and is obtained by instrumenting the program. Mutation analysis is used to produce an infection estimate of the probability that the succeeding data state becomes infected. A propagation estimate of the probability that an infected data state will propagate to output is produced by randomly perturbing the location's succeeding data state and observing how many such perturbations produce different output. Locations that have low execution, infection, and/or propagation estimates are *insensitive* and indicate where hard-to-find faults might be hiding.

## **1.2 Deficiencies in existing white box strategies**

Only exhaustive testing is guaranteed to discover all faults in a program. Random black box testing relies on probabilistic "luck" to reveal faults at the expense of a large test set. White box strategies attempt to improve their "luck" with a smaller test set by selecting test data based on criteria derived from the program's code, but there is still an element of chance involved with respect to the fault-revealing ability of the actual test data selected. Test data selected by even the most simplistic white box strategy might reveal an obscure fault, while test data selected to satisfy the most

sophisticated white box strategy might not reveal the fault. The techniques of white box testing described in the previous section have many positive qualities, but some deficiencies as well. These deficiencies and the techniques that exhibit them are listed below.

*Rely primarily on program syntax*

Almost all of the strategies primarily use syntactic aspects of the program to drive testing, yet the results of program execution are determined semantically. Control flow strategies like statement coverage and path coverage pay no attention to the semantics of the computation that takes place along an execution path. Neither does the family of data flow testing strategies, although the exercising of *dr* pairs is motivated by an attempt to partially cover some semantic properties of the code. Error-based strategies like mutation testing are somewhat more semantically oriented since they rely heavily on dynamic analysis, but even fault-based testing and fault sensitivity analysis perturb program syntax to drive their testing.

*Ignore coincidental correctness*

Coincidental correctness is a fundamental problem in program testing, yet it has been largely ignored by researchers and plays little part in testing methods. Structural coverage techniques, including data flow testing, at best simply cover each structural aspect (possibly multiple times) and hope that errors will be revealed if they exist. Mutation testers simply continue testing a mutant with different input data

until the mutant is killed or they finally decide to attempt to show that the mutant is an equivalent form of the original program. Symbolic testing captures the full semantics of the computation on a path-by-path basis, but focuses on whether coincidental correctness occurs and not how or where. Fault sensitivity analysis partially handles coincidental correctness based on input/output relationships and not the semantics of the computation sequence.

### Computationally intensive

The structural coverage technique of all-paths is in general impossible to attain due to the potentially infinite number of program paths. In many programs the combinatorial explosion of paths also makes this technique untenable. One advantage that data flow testing has is the fact that a finite (and often reasonable) number of paths will cover all definition-use combinations. Mutation testing can result in the execution of a large number of mutants, possibly many times each. Symbolic testing captures the full complexity of the fault/failure model but involves computationally expensive symbolic executions. The computational demands of locating insensitive code make fault sensitivity analysis costly.

### Defensive

We say that a testing technique is *defensive* if it selects test data not to actively locate faults, but rather to ensure that the opportunity for a fault to be revealed was not missed. All structural coverage techniques are defensive in nature. The only claim



that can be made after successful structural testing with a particular test set is the trivial claim that testing was performed with that test set and no faults were found. Coincidental correctness also prevents positive statements about the absence of faults after applying structural coverage techniques.

*Results of testing are not quantifiable*

None of the white box testing techniques described allow quantification of test results in terms of probability of failure for the tested program.

*Can't handle missing code faults*

Techniques based on structural coverage make no explicit effort to compensate for missing code, so missing code faults may only be revealed fortuitously with these techniques.

*Can't handle infeasible or infrequent paths*

Infeasible paths cannot in general be identified by static analysis. Testing techniques that require coverage of certain paths may choose infeasible paths, which could result in wasted testing in an attempt to cover such paths or additional effort (often human-intensive) to show their infeasibility. The selection of paths that are feasible but infrequently executed can also cause problems for the tester because randomly generated test data is unlikely to cover the path and the process of selecting input data to cover a specific path can be very difficult. Dynamic testing and

analysis methods are based on paths that are actually executed, which obviates the infeasible paths problem but leaves these methods susceptible to incomplete analysis due to the omission of one or more infrequent paths.

*Don't necessarily make best use of limited testing resources*

Resources that can be devoted to software testing are often limited, so we would like to get the maximum value from each test in terms of error-revealing power. None of the structural coverage strategies make any distinction among the error-revealing power of different paths or test data. Fault-based strategies and fault sensitivity analysis make no distinction among the error-revealing power of different paths.

**1.3 A new approach - error flow analysis**

White box analysis techniques provide useful information about the properties of programs. This information can be used in many ways including testing, debugging, efficiency measures, and complexity measures. We believe that white box analysis should be grounded in semantic behavior in addition to syntactic structure. To this end, we need a theory of the semantic behavior of errors in programs that will allow us to determine the possibilities and limitations of semantic analysis. This position is supported by Podgurski [Pod89, Pod90] who has formally characterized the strong

and weak syntactic and semantic dependencies in the flowgraph of a program. He demonstrates that syntactic dependence is a necessary but not sufficient condition for semantic dependence, and suggests that a combination of data flow analysis with semantic analysis would be appropriate.

Mathematical functions serve as our semantic model of computation, and we analyze the decomposition of a computation into its components along with the interaction between these components. We use this model to discuss errors and faults, and describe the phenomenon of coincidental correctness as a fundamental issue in testing. Coincidental correctness has two components in terms of the fault/failure model. The first component, *resistance*, occurs when a fault is executed but the data state is not infected. *Cancellation* occurs when an infection is nullified by the action of a subsequent computation. We examine these phenomena and discuss analytic methods that can identify or predict coincidental correctness as well as the limitations of such analyses.

We introduce the term *error flow analysis* to describe the analysis of the semantic behavior of a potentially faulty program in terms of data state transitions. This includes both *dynamic error flow analysis*, which studies the actual infection of data states and the propagation of errors in a program under given input, and *static error flow analysis*, whose goal is to statically approximate a program's error flow behavior through syntactic and semantic analysis of the program's code.

This research effort differs from previous white box research in several important ways, and addresses some of the problems mentioned previously. Perhaps most importantly, it is grounded in the semantics of program execution and based on a sound mathematical model. The focus is on how potential data state errors are handled by the computation rather than the syntactic forms of faults. This allows us to address the coincidental correctness problem at the fundamental level of data state transitions. The data state orientation of error flow analysis also makes it possible to analyze the effect on the data state of some missing code faults. Another advantage of our method is that it takes an offensive approach in locating potential faults, in that it attempts to use properties of the computation sequences involving the potential fault to demonstrate either its presence or absence by testing. We hope this will ultimately contribute to quantifiable test results in terms of estimating the probability of failure for a program.

Dynamic error flow analysis, like other dynamic methods, is computationally intensive and its application to testing requires the use of mutant faults as a basis for estimating the behavior of the program in the presence of real faults. Unlike other techniques, dynamic error flow analysis captures failure and coincidental correctness information on a path-by-path basis, and uses this information to identify paths that would be desirable to test. This differentiation among the error-revealing power of paths makes dynamic error flow analysis more selective than techniques that do not

make the distinction, and potentially saves testing resources that would be devoted to output verification.

Static error flow analysis uses analytic techniques based on the program flow graph which are similar to those used in data flow analysis and other static methods, and suffers from their common weaknesses involving infeasible paths and other issues related to undecidability. The goal of static error flow analysis is to approximate the error flow behavior that would be observed under dynamic error flow analysis and identify paths to be tested, which requires attention to coincidental correctness and other semantic issues. Static error flow analysis of a program is less computationally intensive than the corresponding dynamic error flow analysis, and is preferable if it can achieve a good estimate of the program's dynamic behavior.

Our approach to the development of testing methods also differs from previous research. Other testing schemes have either chosen some program property that seems reasonable to cover, or developed a model of some aspect of program structure or behavior and then derived the testing strategy from that model. We have developed a model of the semantic behavior of programs and used that model to investigate properties of program behavior. Our testing strategies are derived from these observed properties rather than the model.

Error flow analysis has other applications besides testing. Dynamic error flow analysis captures the semantic behavior of different program paths under different inputs. We can use dynamic error flow analysis of a particular fault as a basis for evaluating the effectiveness of testing methods in choosing paths that will reveal the fault. This evaluation can also be used to compare the performance of testing methods that are not otherwise comparable in the data flow-based subsumption hierarchy in [Cla89].

#### **1.4 Contributions of this thesis**

This thesis makes the following contributions :

- Introduces the technique of error flow analysis, which uses semantic and syntactic properties of a program to discover information about its behavior, both past and potential.
  
- Describes a formal model of data state errors in computer programs in terms of mathematical functions. This model serves as the basis for error flow analysis.

- **Describes the DEFA system, a new tool for analyzing the dynamic error flow behavior of different execution paths for a given fault in a program.**
  
- **Demonstrates empirically that different paths through a fault have different failure rates and different error flow properties.**
  
- **Identifies an error flow statistic (average maximum error set size) which for particular errors in the programs analyzed corresponds to paths with high failure rates.**
  
- **Presents experimental evidence which suggests that the identification of error flow characteristics that correlate with high failure rate paths can result in testing techniques that are more selective than existing white box testing strategies.**
  
- **Describes the use of the DEFA system to compare the performance of other testing techniques.**

## **1.5 Outline of this thesis**

Chapter 2 contains some terminology, notation, and background theory necessary to the remainder of this thesis, including a discussion of faults, errors, and coincidental correctness. Chapter 3 presents a semantic model of errors in computer programs based on mathematical functions, and discusses the coincidental correctness problem in this context. Fundamental concepts in error flow analysis are also introduced. Chapter 4 describes empirical studies of the dynamic error flow behavior of several programs using a tool constructed for this purpose, and analyzes the results. Chapter 5 contains a discussion of the estimation of a program's dynamic error flow behavior through static analysis, and presents a possible technique suggested by the empirical study. Chapter 6 presents a summary of this thesis and describes ongoing and future research resulting from this work.



## 2. Background theory and terminology

This chapter contains some of the background theory, notation, and terminology we use in subsequent chapters. Some of the terms were introduced informally in the preceding description of past research. Most of the non-mathematical terms presented here are defined in the context of statements and variables in a high-level programming language. In chapter 3 we will redefine some of these terms in the context of mathematical functions, but it should be clear to the reader that the usage is consistent. There is some disagreement within the research community on the meaning of certain terms like "fault", "error", and "cancellation". We explicitly state the meaning we ascribe to such terms.

### 2.1 Mathematical functions

We begin with some familiar definitions. The *Cartesian product* of two sets  $A$  and  $B$ , denoted  $A \times B$ , is  $\{(a,b) \mid a \in A \text{ and } b \in B\}$ . A *relation*  $R$  from a *domain* set  $A$  to a *range* set  $B$  is a subset of  $A \times B$ . We write  $a r b$  to denote  $(a,b) \in R$ . The *inverse* relation  $R^{-1}$  is defined as  $\{(b,a) \mid a r b\}$ . A *function*  $F$  from set  $A$  to set  $B$  is a relation that maps an element  $a \in A$  to a unique  $b \in B$ , and we denote the

mapping of the function as  $f: A \rightarrow B$ . We will use  $F$  to denote the set of ordered pairs that defines the function, and  $f$  to denote the function operator. An operation on an element  $a$  of the domain  $A$  of function  $F$  that produces element  $b$  in the range  $B$  of  $F$  is written as  $a f b$  or  $f(a)=b$ . We use  $f$  to denote a string that implements function  $f$  under a particular interpretation. If  $a f b$  is defined for all  $a \in A$  and some  $b \in B$ , the function is a *total function*, otherwise it is a *partial function*. In a partial function,  $f(x)$  is *undefined* for each  $x \in A$  such that  $(x,b) \notin F$  for any  $b \in B$ , and we write  $f(x) = \perp$ . For  $f: A \rightarrow B$ , we define the *image*  $I_f$  to be  $\{b \mid b \in B \text{ and } a f b\}$ . If  $I_f = B$ , we say  $f$  is *onto*  $B$ , otherwise  $f$  is *into*  $B$ . Function  $f$  is *one-to-one* iff for each  $x,y \in A$ ,  $x \neq y \rightarrow f(x) \neq f(y)$ . A function that is not one-to-one is *many-to-one*. A total, one-to-one, onto function is called a *one-to-one correspondence*. For two functions  $f: A \rightarrow B$  and  $g: B \rightarrow C$ , we define the *composition of functions*  $g \circ f: A \rightarrow C$  as  $g(f(a))$ , that is  $\{(a,c) \mid a \in A \text{ and } (\exists b \in B \mid b = f(a) \text{ and } g(b) = c)\}$ . For two functions  $f: A \rightarrow C$  and  $g: B \rightarrow D$ , where  $A \cap B = \emptyset$ , we define the *union of functions*  $f \cup g: A \cup B \rightarrow C \cup D$  as  $\{(x,y) \mid y=f(x) \text{ or } y=g(x)\}$ .

Since computers perform transformations between finite domains of input and output values, we may map these values into  $\mathbb{N}$ , the set of natural numbers, and view the transformations as functions of the form  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  for finite  $k$ . A major result of computability theory tells us that a class of functions of the above form, called *partial recursive functions*, are exactly those functions that can be computed.

## 2.2 Flow graphs

Static analysis of a program requires a representation of its structure and the relationship between various parts of the code. Flow graphs provide a good formalism for this purpose. The following standard definitions are adapted from [Pod90].

A *directed graph*  $G$  consists of a finite set of *vertices*,  $V_G$  and a set of *arcs*,  $A_G$ , which is a subset of  $V_G \times V_G - \{(v,v) \mid v \in V_G\}$ . An arc  $(u,v) \in A_G$  is directed from  $u$  to  $v$ . We say that  $u$  is *adjacent to*  $v$  and  $v$  is *adjacent from*  $u$ . A *predecessor* of vertex  $v$  is a vertex adjacent to  $v$  and a *successor* of  $v$  is a vertex adjacent from  $v$ . The *indegree* of a vertex is its number of predecessors, and the *outdegree* of a vertex is its number of successors. A *path*  $P$  in  $G$  is a sequence of vertices  $v_1, v_2, \dots, v_n$  such that  $n \geq 2$  and  $(v_i, v_{i+1}) \in A_G$  for  $i = 1, 2, \dots, n - 1$ .

A *control flow graph*  $G$  is a directed graph that satisfies :

1. The maximum outdegree of any node is at most two<sup>1</sup>.
2.  $G$  contains two distinguished vertices : the *initial vertex*  $v_I$ , which has indegree 0, and the *final vertex*  $v_F$ , which has outdegree 0.
3. Every vertex in  $G$  occurs on some path from  $v_I$  to  $v_F$ .

---

<sup>1</sup>This restriction is made for simplicity only

The vertices of a control flow graph for a program may represent either individual statements or *basic blocks* of statements. A basic block is a group of statements that are executed in a fixed sequence without other possible flow of control. A *predicate* is a Boolean-valued expression that guards a possible branch. Each predicate occupies its own vertex and has an outdegree of 2. Paths in the control flow graph represent a possible sequence in which the program statements might be executed.

For a control flow graph  $G$ , vertex  $u \in G$  *forward dominates* a vertex  $v \in G$  iff every path from  $v$  to  $v_F$  contains  $u$ . We say that  $u$  is a *proper forward dominator* of  $v$  iff  $u$  forward dominates  $v$  and  $u \neq v$ . The *immediate forward dominator* of a vertex  $v \in (V_G - \{v_F\})$  is the vertex that is the first proper forward dominator of  $v$  to occur on every path from  $v$  to  $v_F$ . Informally, the immediate forward dominator of a vertex containing a predicate is the earliest point where the paths that split from the predicate are rejoined. We use the immediate forward dominator, which we simply call the *forward dominator*, in our execution trace matching algorithm that compares the execution of the faulty program with the corresponding correct program. The matching algorithm uses forward dominators to re-synchronize the matching process when structural differences exist between the programs.

### 2.3 Faults, errors, and coincidental correctness

Consider a high-level procedural programming language in which one uses a set of statements to manipulate program variables, each of which has a specified data type, to produce a desired output from a given input. A *data state* is the set of variable/value pairs at a given point during program execution. Since it will be necessary to know which statement the program is executing, the program counter (PC) is considered a variable in the data state. In addition to initial and final data states, there is a data state immediately following the execution of each statement that represents the effect the statement had on the data state immediately preceding its execution; thus each statement computes a function that maps its *prior data state* onto its *subsequent data state*.

We can view a *fault* in a program as an incorrect statement which will cause the program to produce the wrong output (according to the program's specification) for at least one element of the program's input domain. A *faulty program* contains one or more faults. Note that a program that meets its specification cannot contain faults. We consider only specifications that describe the output values that should be produced by a program, and do not consider time/space efficiency, time deadlines, or other issues. We also do not consider faults that result in non-termination.

It is not always possible to label a particular statement as a fault, since there

may be many ways to change one or more statements to produce a correct program from a faulty program. An illustration is the familiar too many/too few loop iteration problem, which is usually solved by changing the position of the loop counter increment statement, changing the exit condition, or changing the initial or final counter value. We are not concerned with the identification of the fault, but rather the effect or potential effect the fault might have on program execution. In our analysis we consider a potential fault in isolation by comparing the faulty program to a correct program that results from fixing the fault. Since we compare the data states produced by executing the programs in tandem, we require that the "fix" be done in a fashion that minimizes the change to the structure of the faulty program, which can often be accomplished by changing only one statement. This is discussed in more detail in Section 3.3. We can model the effects of a *missing statement fault* by defining the fault as a null statement at the location where the missing statement belongs.

When a fault is executed, it may result in the assignment of an incorrect value to a variable in the data state as compared with the corresponding data state of the correct program. We say that such a variable is *in error* or *infected*. A data state is infected if it contains one or more infected variables. Observe that a fault that causes the wrong branch to be taken causes the PC to become infected. We use the term *error* to refer to an infected variable, or program output produced from an infected variable. An infection may *propagate* to other variables in subsequent data states in

one of two ways :

1. An infected variable is used in an expression that assigns a value to a non-infected variable, and computation of the expression's value produces an incorrect result.
2. A fault in a predicate causes a wrong branch, which then results in one or more incorrect assignments to non-infected variables. This can include variables on the wrong path being assigned incorrect values as well as variables on the correct path not being assigned their correct values.

Propagation also includes the transfer of an infected variable from a data state to its subsequent data state without being changed by the intervening statement. A *program failure* occurs when an infection propagates to variables that are then used to produce incorrect output. The data flow characteristics of a program determine the potential for propagation of infected variables, since the definition/reference pair is the mechanism which links the value of one variable to the definition of the value of another variable.

We define *coincidental correctness* as the phenomenon that occurs when a faulty program produces the correct output for a given input. There are three ways that coincidental correctness can occur ( the complement of failure ) :

1. A faulty statement was never executed.
2. The execution of a faulty statement did not result in an infection. We call this *resistance*.
3. All infections of live variables were *cancelled* by subsequent computation before they could propagate to output.

Resistance and cancellation are dynamic events and only occur during program execution. We say that a fault at a particular location has the *potential for resistance* if not all executions of the fault will result in infection, and we say that a statement has the *potential for cancellation* if it uses infected variables but will produce the correct result for at least one execution of the statement. Some faults have no potential for resistance; for example, if the correct assignment statement is  $a := 1$  and the corresponding fault assigns some constant other than 1 to  $a$ , an infection will always result. Other faults have varying potentials for resistance. Consider the fault  $a := b * c$  which was supposed to be  $a := b * d$ . Assuming the variables  $b$ ,  $c$ , and  $d$  are uninfected,  $a$  will receive its correct value if 1)  $b$  has a value of 0, or 2)  $c$  and  $d$  are equal. A "faulty" statement  $a := b + b$  which should have been  $a := 2 * b$  is really not a fault at all, since both expressions compute the same function. Perhaps predicates have the greatest potential for resistance, e.g. a predicate like  $a <= b$  that should have been  $a < b$  rarely causes an incorrect branch (assuming  $a = b$  is a rare event).



Cancellation can occur in either of two ways : *expression cancellation* or *assignment cancellation*. In expression cancellation, a correct statement contains an expression that involves one or more infected variables, but the evaluation of the expression produces the correct result. Using the same example as above, suppose  $a:=b*c$  is a correct statement, but  $c$  is infected. The variable  $a$  will still be assigned the correct value whenever  $b=0$ . Some correct statements have no potential for expression cancellation. The statement  $a:=b+1$  will always infect the variable  $a$  if  $b$  is infected. The second type of cancellation, assignment cancellation, occurs when a correct value is assigned to an infected variable. A statement of the form  $a:=f(\text{uninfected variables})$ , where  $f$  is a correct function, will always cancel an infection of the variable  $a$ . Other assignments to an infected variable that involve expressions that reference infected variables may or may not cancel the infection.

### **3. A functional view of programs and errors**

Viewing computer programs as mathematical functions is not new. Computable functions and abstract machines were in fact the precursors of computer programs and computers. We use this solid mathematical foundation to formalize our concepts of the infection and propagation of data state errors in the fault/failure model, and redefine some of our previous terminology in this context. The properties of mathematical functions allow us to prove some theorems about testing and coincidental correctness, and provide a formalism for error flow analysis.

We consider here only total functions that map a finite domain onto a finite range, since our interest is in faulty programs that terminate with defined, erroneous output.

#### **3.1 The von Neumann model of computation**

Most computers are based on the *von Neumann model of computation*, where a *processor* capable of executing a set of primitive instructions is connected to a *memory* which can hold data values and programs. A *program* is a string of symbols

that, when interpreted by the processor, results in the execution of a sequence of zero or more *primitive instructions*, each of which may read or write zero or more data values in memory. The set of values stored in memory together with information concerning the current point in the program string where interpretation is taking place (the *program counter*) constitutes the *data state* at a particular point during execution of the program. Each primitive instruction has a *prior data state* that exists immediately before its execution and a *subsequent data state* that exists immediately after its execution. The transformation of a primitive instruction's prior data state into its subsequent data state caused by executing the instruction is called a *primitive computation*. A *computation* is a sequence of one or more primitive computations, and represents the net effect on the computation's prior data state of the execution of the corresponding sequence of primitive instructions. The set of all possible primitive computations for a particular primitive instruction defines the *function computed by the instruction*, and its domain consists of those prior data states of the instruction that could possibly be produced from the program's input domain. We equate a primitive computation with an application of the function computed by the corresponding primitive instruction to its prior data state, and equate a sequence of computations with the application in composition of the corresponding functions. The complete sequence of computations for a given input defines the output computed by the program for that input, and the set of sequences over all inputs defines the function computed by the program. Programs whose instructions manipulate an internal data state to compute the program function are called *imperative programs*.

Imperative programs have several properties that make it difficult to infer their semantic behavior from a syntactic examination of their code. The entire data state serves as both domain and range for each computation in the program. This is necessary since a subsequent computation may potentially operate on any variable in the data state, even though each computation typically operates on a small subset of the data state. Those computations that operate on the same portion of the data state may be separated by many other computations during execution, so that the sequence of computations may bear little resemblance to the actual function being computed. The program statements that specify computations that operate on the same part of the data state may also be widely separated in the code, and the lexical ordering of statements in the code may not reflect the order of the corresponding computations. All of this serves to obfuscate the true function of the program.

In spite of these problems, imperative programming is still the most common paradigm, so many techniques have been and will continue to be oriented in its direction. Flow graph-theoretic techniques such as data flow analysis have been of great assistance in dealing with the aforementioned problems. Podgurski [Pod90] has used flow graph theory to formalize the possible syntactic and semantic dependences between statements in a program with respect to possible control flow and data flow. This is essential to determine how errors might propagate in a program, but a more complete characterization of the behavior of a fault as related to program failure requires consideration of the semantics of the computations along the path from fault

to output.

### 3.2 Function implementations, syntactic and semantic distance

A *specification* of a function is a complete description of the domain to range mapping that the function performs. A *correct implementation* of a function results from a semantic-preserving transformation of the specification into a string of symbols that has a meaningful interpretation under a predefined calculus. The application of this calculus to the implementation with each element of the function's domain must produce the specified element in the function's range if the transformation was semantic-preserving. We will call the application of the calculus to the function's implementation an *execution* of the implementation. The element of the function's input domain used in the execution is called the *input* for that execution and the corresponding range element produced is called the *output*. There are infinitely many correct implementations of a particular specification, although only a small number are realistic alternatives. The *syntactic distance* between two implementations is the number of symbol substitutions performed in transforming one string into another string. Symbol substitutions take one of the forms  $s_1 \rightarrow s_2$ ,  $s_1 \rightarrow \Lambda$ , or  $\Lambda \rightarrow s_1$ , where  $s_1, s_2$  are single symbols and  $\Lambda$  is the empty symbol. We require that the set of substitutions result in a syntactically correct string, and do not count substitutions that have no semantic significance. When dealing with an actual

implementation we must consider the *granularity* of the symbols in its string. A program written in a typical structured high-level language may be viewed, for example, as a string of characters, a string of lexical tokens, or a hierarchy of statements. We might correspondingly define a "symbol" in this implementation to be a single character, a token, or a statement.

An *incorrect implementation* of a function results from a transformation that is not semantic-preserving. Thus there exist one or more elements of the function's domain such that execution of the incorrect implementation with each of these inputs will produce an output that is inconsistent with the function's specification. Such an output is called an *error*. The *fault* in an incorrect implementation is defined as a set of symbol substitutions required to transform the incorrect implementation into a correct implementation. Since the implementation is usually composed as a syntactic hierarchy of (possibly multisymbol) objects, it makes sense to decompose a fault into a set of *syntactic faults*. Note that this set is not unique -- different people might identify different sets of syntactic faults in fixing an incorrect implementation. It is this property that makes the identification or counting of faults in computer programs a difficult issue.

We define the *least syntactic distance* between two function implementations  $f_1$  and  $f_2$  as the fewest number of symbol substitutions required to transform  $f_1$  into  $f_2$ , or vice versa. The set of symbol substitutions made to achieve the least syntactic

distance is not necessarily unique. An implementation  $f_1$  is *syntactically close* to an implementation  $f_2$  if the least syntactic distance between  $f_1$  and  $f_2$  is not greater than some arbitrary limit  $k$ . The *semantic distance* between two functions  $f$  and  $g$  with the same domain  $D$  is defined as the number of domain element to range element mappings on which the functions differ, i.e., the size of the set  $\{ x \mid x \in D \text{ and } f(x) \neq g(x) \}$ . Function  $f$  is *semantically close* to function  $g$  if the semantic distance between  $f$  and  $g$  is not greater than some arbitrary limit  $k$ . Finally, consider an incorrect implementation  $f'$  of function  $f$ . In the set of correct implementations for  $f$  that might realistically be constructed, there is at least one whose least syntactic distance from  $f'$  is minimal. We presume that such an implementation was the intended implementation of  $f$  as a basis for comparison of the incorrect and correct implementations of  $f$ .

How are least syntactic distance and semantic distance between two functions related? Let's consider a couple of extreme cases. We will assume that the granularity of our implementation views symbols as individual characters. Suppose  $f$  is a program that implements some function  $f$ , which computes a unique value from each input, and the first statement in  $f$  is a predicate that sends half of the input values down one path and half down the other path by using a comparison " $<$ ". Now transform  $f$  to  $g$  by replacing " $<$ " with " $> =$ ". The least syntactic distance between  $f$  and  $g$  is small (2), but the semantic distance between  $f$  and  $g$ , the function computed by  $g$ , is maximal if no input-output pairs match between  $f$  and  $g$ .

Now consider a function  $f$  whose input is a binary tree and whose output is the node sequence from a preorder traversal of the tree. Implementation  $f_1$  is a correct implementation of  $f$  using recursion, and implementation  $f_2$  is a correct implementation of  $f$  using iteration and a stack. Although the syntactic distance between  $f_1$  and  $f_2$  is large, the semantic distance is 0.

Can we predict any general relationship between least syntactic distance and semantic distance, ignoring pathological cases like the above? One might guess that as you increase the least syntactic distance, the semantic distance also increases, possibly quickly. While this has an intuitive appeal, there are other factors that must be considered. One such factor is the set of syntactic and semantic dependences of the changed statement both before and after the change; i.e., a change to the definition of a variable that has few data flow chains (*a la* [Nta84]) would possibly have less impact on the semantic distance than a change to the definition of a variable that has many data flow chains. Another factor is the proportion of inputs that drive the path containing a changed statement. Even major changes to a path driven by few inputs would have limited impact on the semantic distance. Finally, the impact of the change on the coincidental correctness behavior must be considered as well.



### 3.3 Error sets and error traces

There is reason to believe that programs which contain faults compute a function with a small semantic distance from the specified function, since a program with a larger semantic distance would be revealed as faulty by routine testing in the development phase. This view is espoused in [DeM78] as the "competent programmer hypothesis". We conjecture that in general these programs also have a small syntactic distance. We also believe that programs with a small least syntactic distance usually have a similar structure. Our approach in dynamic error flow analysis involves using a (presumed) correct program to produce a syntactically close faulty program. Because dynamic error flow analysis compares data states produced by the execution of program statements, we choose the level of granularity to be symbols that may be predicates, assignment statements, or procedures, and define "syntactic closeness" as  $k=2$  for our experiments. Since the programs have a similar structure, they can be executed in tandem with the same input, and their data states compared to observe the change in semantic behavior. This allows us to estimate the semantic distance between the correct and faulty programs.

We define an *error set* as a subset of a data state in a faulty program that consists of those variables whose values are incorrect as compared with the variables in the corresponding data state of the correct program, given that both programs are executed with the same input. An *execution trace* is the sequence of data states

produced by executing a program with a particular input. We note that an execution trace corresponds to the sequence of program statements executed and results from the composition of the respective functions computed by the statements. An *error trace* is the sequence of error sets generated by comparing corresponding data states in the execution traces of both programs under the same input. When the sequences of statements executed by both programs are in 1-1 correspondence, both in length and respective positions of each statement in the sequences, the term "corresponding data state" is clear; when they are not we need to determine which data states will be compared. An error set is associated with each data state in the faulty program, and any synchronization is done with respect to the sequence of statements executed in the faulty program. We synchronize the comparison of data states to the greatest degree possible by matching the control flow graphs of both correct and faulty programs, since a comparison of data states from different structural parts of the program is likely to lead to abnormally large error sets. This is the reason that dynamic error flow analysis is based on syntactically close programs. Data state comparisons are synchronized when necessary by finding the forward dominator common to both flow graphs and "catching up" one or both data state sequences to that point. When the faulty program's data state sequence must catch up with the correct program's data state, an error set is produced for each data state in the faulty program by comparing each with the correct program's data state. When the correct program's data state sequence must catch up with the faulty program's data state, one error set is produced for the faulty program's data state by comparing it with the last

data state in the correct program's data state sequence. In either case, the data states in the corresponding programs are resynchronized at the common forward dominator of both control flow graphs, and normal comparison can resume. When both sequences follow different paths, some means of comparison must be done until both reach the common forward dominator. We choose to compare corresponding states in each sequence until one branch runs out, and then perform "catch up" comparison.

An error trace for a given input captures not only the execution, infection, and failure semantics for the faulty program with that input, but also the semantics of each computation's effect on its subsequent data state, including any resistance or cancellation. The set of error traces over all inputs characterizes the difference in the semantic behavior of the correct and faulty programs.

### **3.4 Coincidental correctness and functions computed by paths**

We now come to a fundamental problem of verification through testing : If we successfully test less than the entire input domain of a function, we may not claim that an implementation is correct with respect to the function's specification.

### Theorem (Inadequacy of Testing)

Let  $f$  be a total function that maps a domain  $D$  onto a range  $R$ , and let  $F = \{(x,y) \mid x \in D \text{ and } y \in R \text{ and } x f y\}$  be the mapping for  $f$ . Let  $f$  be the implementation of  $f$ , which computes the mapping  $F' = \{(x,z) \mid x \in D \text{ and } x f z\}$ . Suppose that for some  $S \subset D$  we define the restricted mappings  $F_S \subset F$  and  $F'_S \subset F'$  by limiting their respective domains to  $S$ . Then

$$(F_S = F'_S) \text{ does not imply } (F - F_S = F' - F'_S)$$

### proof

Assume  $(F_S = F'_S)$  does imply  $(F - F_S = F' - F'_S)$ . Since  $D - S$  is non-empty, we may choose  $x \in (D - S)$ . For this  $x$  and its mapping  $x f z$ , choose  $w$  such that  $x f w \neq x f z$ . Now change  $F'$  to  $F' - (x,z) + (x,w)$ . If  $(F_S = F'_S)$  was originally true, it remains true with the new  $F'$ ; however, now  $F - F_S \neq F' - F'_S$ , a contradiction.  $\square$

This theorem simply states what testers have always known - in the absence of other information we have no assurance of getting correct output from untested inputs.

Using the notation from the above theorem, we say that the implementation  $f$  has a *potential for coincidental correctness* w.r.t.  $F$  iff  $F'$  can be partitioned into two non-empty subsets  $F'_C$  and  $F'_E$  where  $F'_C = \{(x,y) \mid x \in D \text{ and } (x,y) \in F\}$  and  $F'_E = \{(x,y) \mid x \in D \text{ and } (x,y) \notin F\}$ . This effectively partitions  $D$  and defines the function computed by  $f$  as the union of two functions, one which performs the correct mapping and one which performs an incorrect mapping. There is no potential for coincidental correctness if  $F'_C = F'$  (always correct) or  $F'_E = F'$  (always wrong). We say that coincidental correctness has occurred when the implementation  $f$  of function  $f$  has the potential for coincidental correctness and is executed with  $x \in D$  producing  $y$  such that  $x f y$ .

Can we examine (short of exhaustive testing) a function's specification and implementation to determine the correctness of the implementation? If the specification and implementation are described in terms of properties from which the mappings may be derived, we can use inductive and deductive logic to show equivalence. This is the method of formal proof of correctness. For a function that cannot be further decomposed, exhaustive testing and formal proof of correctness are our only alternatives. However, for functions that may be decomposed into a composition of subfunctions, we may be able to verify the correctness of the implementation by first determining some properties of the subfunctions and their interaction under composition, and then testing to the extent necessary to verify those properties. In particular, we must determine those properties related to coincidental



$$f([v_1 \dots v_k' \dots v_n]_i) = [v_1 \dots v_k \dots v_n]_{i+1}$$

assignment cancellation

4.  $f'([v_1 \dots v_k' \dots v_n]_i) = [v_1 \dots v_k \dots v_n]_{i+1}$   
 $f'([v_1 \dots v_k' \dots v_n]_i) = [v_1 \dots v_k' \dots v_n]_{i+1}$   
 $f'([v_1 \dots v_k' \dots v_n]_i) = [v_1 \dots v_j' \dots v_k' \dots v_n]_{i+1}$

The *potential* for infection, resistance, propagation, or cancellation for a given function implementation is defined as the existence of at least one element of the function's domain for which the corresponding above form occurs during execution of the implementation with that input. We clearly expect form 4, where an incorrect implementation is applied to an infected data state, to make things worse instead of better. Note that in the first case of form 3, it is impossible to tell whether infected variables propagated by default (not referenced by  $f$ ), self propagated (an already infected variable assigned a wrong value), or whether expression cancellation occurred unless we know the particular variables defined and/or referenced by  $f$ .

Since all of the functions represented by computations have finite domains and ranges, we may think of each function as a function of one variable : the data state, i.e.  $g: DS \rightarrow DS$ . Although  $DS$  is the Cartesian product of the set of values for each of the program variables, the domains and ranges of each function are usually a (sometimes small) proper subset of  $DS$ . We note that the position of the function  $g$

in a composition of functions  $h \circ g \circ f$  representing the execution sequence  $\mathbf{f};\mathbf{g};\mathbf{h}$  will restrict its domain to  $I_f \subseteq DS$ , the image of  $f$ , and its range  $I_g$  will become the domain of  $h$ . These restricted domains are crucial in discussing potential resistance and cancellation. It is important to observe that the same function may have different domains at different points in execution, as would a computation inside a loop. We will sometimes describe functions of one or more variables, i.e.,  $f(x,y)$ , in our discussion but it is understood that functions which represent computations are functions of the data state.

Consider a function  $g$  incorrectly implemented as  $\mathbf{g}'$  in the computation  $\mathbf{f};\mathbf{g}';\mathbf{h}$ , where  $\mathbf{f}$  and  $\mathbf{h}$  are correct implementations. What can we say about the potential for resistance of  $\mathbf{g}'$ ? First, we must have knowledge of function  $g$  and its domain  $I_g$  but unfortunately such knowledge is not usually available in practice. If this knowledge were available we could find the proportion of data states in  $I_f$  for which  $\mathbf{g}'$  resists infection :

$$R_{\mathbf{g}'} = | \{x \mid x \in I_f \text{ and } \mathbf{g}'(x) = g(x)\} | / |I_f|$$

If the data states produced by  $\mathbf{f}$  are uniformly distributed over  $I_f$ , then the probability of infection (i.e. *infection rate*<sup>1</sup>) is  $1 - R_{\mathbf{g}'}$ . Even if the original input is uniformly

---

<sup>1</sup>Definitions for infection rate and propagation rate (p.54) that are based on non-uniform input distributions are given in [Voa90].



distributed over its domain, this is not necessarily true for the states produced by  $f$ . Clearly, estimation of the infection rate by other than empirical means is a difficult problem.

While any discussion of the potential for cancellation of a function must include the distribution of data states in its domain, we can make some observations by viewing the function in isolation.

Theorem (Cancellation)

If  $g$  is a correct implementation of function  $g$  with a domain of data states  $D_g$  and  $g$  computes a one-to-one function, then  $g$  has no potential for cancellation.

proof

For  $g$  to have the potential for cancellation, by definition there exists  $x, x' \in D_g$  such that  $x'$  is the incorrect state that should have been  $x$ , and  $g(x) = g(x')$ . This violates the assumption that  $g$  computes a one-to-one function.  $\square$

### Corollary

If  $g$  is many-to-one, then  $g$  has the potential for cancellation iff there exist one or more data state pairs  $(x, x')$  where  $x, x' \in D_g$  such that  $x'$  should have been  $x$  and  $g(x) = g(x')$ .

### proof

Follows from the proof of the Cancellation Theorem.

It is important to note that a many-to-one function that occurs in a composition of functions may not result in the degree of cancellation expected by viewing it in isolation. For example, the function  $f(x) = x \text{ div } 2$  that has the potential to cancel the effect of an erroneous odd number that is off by  $+1$  does not have potential for cancellation in the composition  $f(g(x))$  if  $I_g$  contains only positive even integers. Notice that  $f$  is one-to-one in this restricted domain.

Informally, the proportion of data states for which the application of  $g$  results in cancellation,  $C_g$ , is the ratio of the number of data states which contain infected variables but for which no additional variables become infected to the total number of data states in the function's domain. The *propagation rate* is  $1 - C_g$ . As with the determination of infection rate, determination of propagation rate by analytic means requires that we know both  $I_f$  and the frequency distribution of its members in a typical execution. We must additionally identify the data state pairs referred to in the

above Corollary.

Static analysis of the functions in isolation will not yield the information necessary to determine their cancellation behavior in many instances, but can in some instances identify correct one-to-one functions that have no potential for cancellation. Determining the correctness of individual functions can be difficult [How85]. Although the determination of cancellation properties by static analysis alone is in general undecidable, the degree to which static analysis can be used to accurately estimate the cancellation behavior of functions in composition by using information about the original input domain, input distribution, and the properties of the functions in the composition is an open question.

Each execution of a program with an input from its domain executes a sequence of statements called a *path*. The function computed by a path is the composition of the functions computed by the corresponding program statements. Execution of the program over all inputs defines the set of possible program paths, each with an associated disjoint subset of the program's domain, so we may describe the program function as a union of the functions computed by the individual paths. The control flow graph of the program can thus be viewed as an abstraction that collapses the set of paths by merging common subpaths and finding repetitive patterns. For a particular path, it should be clear from the above discussion that any function in the composition of functions computed by the path can exhibit different

properties of infection and cancellation based on the subset of data states that can possibly occur as its domain, and this is ultimately a result of the path's input domain. Infection and cancellation rates for a particular function in a path can be empirically estimated using a random sample of the path's input domain and observing the action of the function, which is the technique we use in dynamic error flow analysis. Since the same function may appear in a number of paths in a program, we want to observe its behavior in all such paths. Clearly, the same function can exhibit different behavior in different paths in terms of infection and cancellation properties. For the purpose of testing the program for correct behavior of this function, we want to test the path where the function is most likely to infect and propagate. The semantic information necessary to make this choice is missing from other techniques. The purpose of error flow analysis is to obtain such information.

## **4.0 Experiments in dynamic error flow analysis**

Often the best way to understand a phenomenon is to observe it under controlled experimentation. The model presented in the previous chapter provides the theoretical framework for describing the infection and propagation of errors in a faulty program's data state as compared with the correct program. The faulty program must be syntactically close to the correct program for the comparison to be meaningful. We have developed a prototype system, which we call the Dynamic Error Flow Analysis system (DEFA), that allows us to investigate the dynamic error flow behavior of such programs. This chapter describes the DEFA system and its potential uses, and presents the results of using the system to investigate the error flow behavior of several programs.

### **4.1 The DEFA system**

Our prototype Dynamic Error Flow Analysis system is implemented in Turbo Pascal in an IBM-compatible/Microsoft DOS environment. We chose these vehicles based on familiarity, availability, and potential portability to other testing researchers. The DEFA system is currently used to analyze programs written in a subset of

standard Pascal that restricts data types to simple integer types and restricts control flow constructs to **while**, **if-then-else**, and procedures. Furthermore, we require unique variable names across all scopes and prohibit the use of global variables in procedures. All output must take place at the end of the program, and output lists may only contain variables.

Most of these restrictions, such as those dealing with control structures and data types, were made to expedite the implementation of the prototype system, and would be lifted in an enhanced version of DEFA. We currently use a simple hand-generated symbol table; a more elaborate symbol table that would address the scope restrictions could be automatically constructed during parsing. We sometimes desire to turn off data state tracing in a procedure to cut down on path length and/or number of paths. The procedure then is treated as a single multi-assignment statement that assigns values to its **var** parameters, so that only the net effect on those variables is recorded. Global variables in the procedure body are prohibited because their side effects on the data state would not be recorded if tracing was turned off inside the procedure. We note, however, that our restricted language can implement any computable function.

Figure 1 shows the basic operation of the DEFA system, with a running example in Figure 2. First, we use a correct program to produce a syntactically close faulty program (Figure 2a). Note that we may define any arbitrary program as

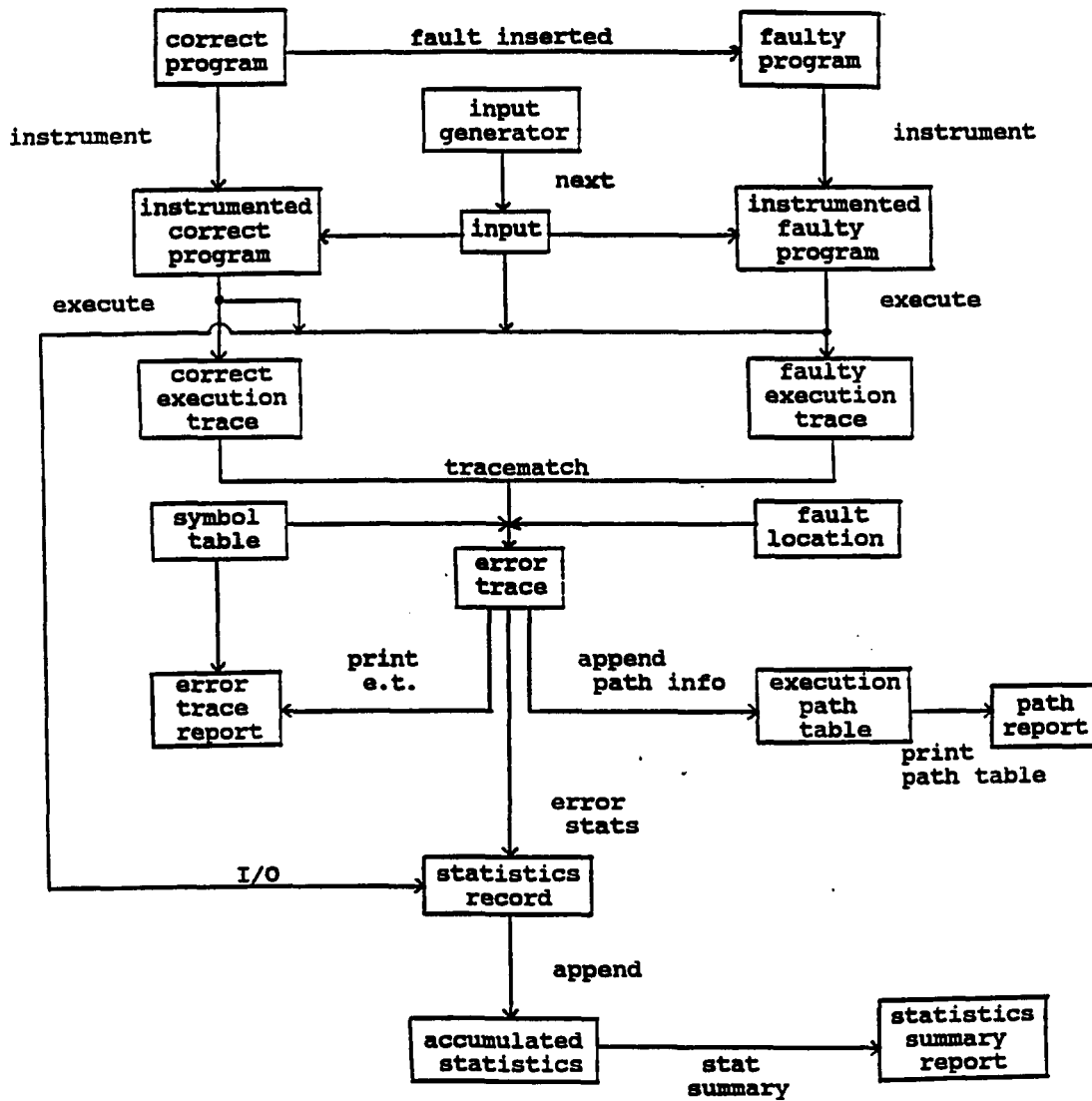
"correct", since our primary interest is in observing the difference in the error flow behavior of syntactically close programs. The correct and faulty programs are then instrumented so that they output a file that traces the changes in their data state as they execute. This instrumentation is accomplished by parsing a program to construct its parse tree, then reproducing the source code with trace file output statements inserted after every statement that causes a data state change. An output trace record consists of the statement number, the level in the program hierarchy for that statement, the variable changed by the statement, and the new value of the variable. Level numbers are used to capture the program's structure, so that we may determine when execution has gone into or come out of loop or decision bodies. This allows us to determine, for example, when a wrong branch has been taken or when a forward dominator has been reached.

The sequence of output trace records constitutes the program's execution trace for a particular input. We use the term "input" to refer to the set of values that must be supplied for one execution of the program. Both the correct and faulty programs are executed with the same input, producing the corresponding outputs and execution traces (Figure 2b). The input, correct program's output, and faulty program's output are saved as artifacts of this execution. The execution traces are then processed by a trace matching program to produce a sequence of error sets with respect to the execution of the faulty program, called the *error trace* (Figure 2c)<sup>1</sup>. Next, the error

---

<sup>1</sup>A similar idea from the standpoint of mutation testing is suggested in [Woo88].

Figure 1. The Dynamic Error Flow Analysis system





**Figure 2.** An example of the operation of the DEFA system

**a. Correct and faulty programs**

```

program correct;
var a,b,c,d,e,f : integer;
begin
1  c := 0; d := 0;
3  readln (a, b);
4  if a < b then
5    c := b - a
  else begin
6    d := 1;
7    a := a * d;
8    while d < 3 do begin
9      c := c + b - a;
10     b := 2 * b;
11     d := d + 1
    end
  end;
12 e := c div 20;
13 f := a + b - c - d;
14 writeln (e, f)
end.

program fault;
var a,b,c,d,e,f : integer;
begin
  c := 0; d := 0;
  readln (a, b);
  if a < b then
    c := b - a
  else begin
    ; (fault - missing stmt)
    a := a * d;
    while d < 3 do begin
      c := c + b - a;
      b := 2 * b;
      d := d + 1
    end
  end;
  e := c div 20;
  f := a + b - c - d;
  writeln (e, f)
end.

```

**b. Execution traces for an input of (9,2)**

**c. Error trace**

correct	fault	stmt no.	Error set
<1,1,c=0>	<1,1,c=0>	1.0	
<2,1,d=0>	<2,1,d=0>	2.0	
<3,1,a=9>	<3,1,a=9>	3.0	
<3,1,b=2>	<3,1,b=2>	3.0	
<4,1,=>	<4,1,=>	4.0	
<6,2,d=1>	<6,2,=>	6.0	d
<7,2,a=9>	<7,2,a=0>	7.0	a d
<8,2,=>	<8,2,=>	8.0	a d
<9,3,c=-7>	<9,3,c=2>	9.0	a c d
<10,3,b=4>	<10,3,b=4>	10.0	a c d
<11,3,d=2>	<11,3,d=1>	11.0	a c d
<8,2,=>	<8,2,=>	8.0	a c d
<9,3,c=-12>	<9,3,c=6>	9.0	a c d
<10,3,b=8>	<10,3,b=8>	10.0	a c d
<11,3,d=3>	<11,3,d=2>	11.0	a c d
<8,2,=>	<8,2,=>	8.0	a c d
<12,1,e=0>	<9,3,c=14>	9.0	a c d
<13,1,f=26>	<10,3,b=16>	10.0	a b c d
<14,1,e=0>	<11,3,d=3>	11.0	a b c
<14,1,f=26>	<8,2,=>	8.0	a b c
	<12,1,e=0>	12.0	a b c
	<13,1,f=-1>	13.0	a b c f
	<14,1,e=0>	14.0	a b c f
	<14,1,f=-1>	14.0	a b c f

**d. Statistics record**

9 2 0 26 0 -1 1 24 1 1 1 18 4 3.0 4 1

trace is processed to produce a statistics record (Figure 2d), and these records are accumulated over all tested inputs. Each different execution path encountered is recorded in the execution path table, and a count is kept for each path of the number of inputs that drove the path during an experiment. Since we are only interested in paths that execute the fault, we need to know the statement number of the fault. This allows the determination of the execution rate during the experiment, and saves space in the path table by eliminating uninteresting paths. The DEFA system can produce several reports, including the error trace for a single input, a description of paths and execution counts, and a summary by paths of the accumulated statistics.

The statistics record produced by processing an error trace contains the following information :

- The input for this execution
- The correct program's output for this input
- The faulty program's output for this input
- The path number, which is the index of this path in the path table
- The length of the path, i.e., the number of statements executed. Note that each variable input or output counts as one statement executed, as does the predicate in **if** or **while**.
- Whether or not the fault was executed
- Whether or not execution of the fault resulted in an infection

- Whether or not the program failed
- The distance from the infection point to either output or total cancellation of the infection. Distance is measured as the difference between the ordinal positions in the execution sequence of the two events.
- The maximum size over all error sets
- The average size of the error sets, produced by summing error set sizes starting at the infection point across the above distance, then dividing by distance + 1
- The size of the final error set
- The number of assignment cancellations that occur, i.e. those that remove an infected variable from the preceding error set. Expression cancellation can be measured by using temporary variables to decompose the expression.

The algorithm to match correct and faulty program execution traces is given in Figure 3. This algorithm attempts to synchronize the program traces as much as possible to avoid the artificial fluctuations in error sets that could be produced by an unsynchronized comparison of data states. As an extreme example, suppose two identical execution traces were matched by comparing all data states in one trace to the initial data state of the other trace, and then comparing the final state of the first trace to subsequent states of the other trace. Although the final error set would be empty, the error trace would have a large average error set size when that average size should be zero.

**Figure 3.** Execution trace matching algorithm

In the following algorithm, C and F refer to the correct and faulty execution traces, respectively. The operation **match** compares the data states for C and F, adding variables to the error set when their C and F values are not equal, and removing (if necessary) variables from the error set when their C and F values are equal. Implicit in **match** is the fetching of trace records as needed and the output of the error set for each statement in F.

```
while more trace records in both C and F do
  begin
    if Flevel = Clevel and Fstmtnum = Cstmtnum then
      match

    else if extra stmt(s) in F then
      match each extra stmt in F with current stmt in C

    else if Flevel > Clevel then
      catch up with C by matching each stmt in F with current
      stmt in C until Fstmtnum = Cstmtnum

    else if Flevel < Clevel then
      catch up with F by accumulating C's state changes
      until Fstmtnum = Cstmtnum

    else if Flevel = Clevel and Fstmtnum <> Cstmtnum then
      catch both traces up to their common forward dominator,
      matching stmts in C and F pairwise until one branch
      runs out
      if C branch runs out first then
        match rest of F stmts with last C stmt
      if F branch runs out first then
        accumulate C's state changes
  end

if more trace records in F then
  match rest of F with C's final state

if more trace records in C then
  accumulate rest of C's state changes
  match with F's final state
```

Our algorithm uses the faulty program's execution trace as the basis for the comparison and requires a correspondence between statement numbers in the correct and faulty programs. A missing statement in the faulty program is represented as a null statement. An extra statement in the faulty program has a fractional part added to the line number of the statement preceding the extra statement(s), i.e., statement 5.1 would be an extra statement inserted between statements 5 and 6. One error trace record consisting of the statement number and the error set is produced for each statement in the faulty trace. When the faulty trace must "catch up" with the correct trace, e.g., when the faulty trace has too many iterations of a loop, error trace records are produced by matching each state in the faulty trace with the current state in the correct trace until the traces are resynchronized. When the correct trace must catch up with the faulty trace, changes in the data state for the correct trace are accumulated until resynchronization but no error trace records are produced. The biggest problem in synchronization arises when the correct and faulty traces follow different branches from a decision statement. We have chosen to match the states along each branch pairwise between the branches until one branch runs out, and then process any remaining states along a branch as above. An alternative would be to accumulate state changes along both branches and then match the resulting states, but that is inconsistent with our notion of matching each state in the faulty trace with a state in the correct trace.

## **4.2 Potential uses for the DEFA system**

The DEFA system is a tool for investigating the semantic behavior of errors in programs at the level of data state transitions. Some of the information produced, such as overall execution rate of a fault and program failure rate, is also produced by other tools, but to our knowledge no other tool produces a breakdown of infection and failure rates by path, or any kind of error set information. We believe this information will be valuable in discovering properties of individual programs and properties found to be common to many programs. Potential uses include :

- Investigating the properties of paths that result in different infection rates for a fault.
- Investigating the properties of paths with different failure rates.
- Investigating the properties of paths that always fail or that never fail.
- Investigating the effects of cancellation along different paths once an infection has occurred.
- Comparing the performance of various testing techniques using the paths selected by the techniques and path infection/failure estimates produced by

the DEFA system.

- Discovering anomalies in programs that may not be obvious from an examination of the code, based on observations such as a "fault" that never infects, or an infection that never results in failure.

### **4.3 Programs analyzed using the DEFA system**

A number of different single faults in three programs have been analyzed using DEFA. Each program and its faults are described in the following subsections. The programs were constructed to conform to the restrictions of the DEFA system and, except for program Cancel, compute some useful function. An attempt was made to construct programs in which faults could be inserted that would result in different failure rates, but the programs were not contrived to produce any other behavior. The faults themselves are mostly simple mutation faults, such as a variable substitution, an operator substitution, or an expression off by one. Some faults were chosen without bias, others were chosen as mistakes a programmer might actually make, and still others were chosen as faults that would have low infection or failure rates. This was done in an attempt to observe some interesting semantic behavior -- we make no claims that these faults are typical or representative of any group of faults.

The experiments were performed on 286-based, IBM-compatible PC's using a test harness designed to automate execution of the different parts of the DEFA system. The time required to perform one experiment involving one fault using 20,000 inputs varied from 15 to 30 hours, depending on the program and the fault.

#### **4.3.1 The Triangle program**

The program in Figure 4 accepts three integer inputs and determines whether or not they could be the lengths of the sides of a triangle. If so, the triangle is classified by properties of its angles and sides. Figure 5a shows the different faults tested for this program. Each experiment involved a different fault run with 10,000 or 20,000 randomly generated inputs, with each input consisting of 3 integers in the range -2 .. 50. Overall results for execution rate, infection rate, and failure rate are presented in Figure 5b. A breakdown for each fault of the number of paths that have certain properties is given in Figure 5c.

There are several insights about this program that are prompted by an examination of the experimental results. The most striking observation about Figure 5b is that (except for fault T1) 100% of the infections result in failure. An examination of the program shows why : the path selected determines the value of each output variable independently and exclusively (except for the two paths that



Figure 4. Triangle program

```

-----
{ Triangle program }
{ }
{ Takes 3 integer inputs and determines if they can be the lengths }
{ of the sides of a triangle. If so, classifies the triangle's }
{ properties as (scalene, isosceles, or equilateral) and (obtuse, }
{ right, or acute). Output consists of a bit string with 1's }
{ indicating the properties in the following order : }
{ error scalene isosceles equilateral obtuse right acute }
-----

program triangle;
const
  true = 1;
  false = 0;
var
  a,b,c : integer;
  err, sca, iso, rig, obt, acu, equ : integer;
  pr, po, pa : 0..1;
begin
1  err := false;
2  sca := false;   iso := false;   equ := false;
5  obt := false;   rig := false;   acu := false;
8  readln (a,b,c);

9  if (a <= 0) or (b <= 0) or (c <= 0) then
10   err := true
11 else if (a + b <= c) or (a + c <= b) or (b + c <= a) then
12   err := true
13 else
14   err := false;

14  if err = false then
15   begin
16     if (a = b) and (b = c) then
17       equ := true
18     else if (a = b) or (b = c) or (a = c) then
19       iso := true
20     else if (a <> b) and (b <> c) and (a <> c) then
21       sca := true;

21     pr := ord ((a*a+b*b=c*c) or (a*a+c*c=b*b) or (b*b+c*c=a*a));
22     pa := ord ((a*a+b*b>c*c) and (a*a+c*c>b*b) and (b*b+c*c>a*a));
23     po := ord ((a*a+b*b<c*c) or (a*a+c*c<b*b) or (b*b+c*c<a*a));
24     if pr = 1 then
25       rig := true
26     else if pa = 1 then
27       acu := true
28     else if po = 1 then
29       obt := true;
30   end;
31  writeln (err, sca, iso, equ, obt, rig, acu)
end.

```

**Figure 5. Results of experiments with Triangle program**

**a. Triangle program faults**

fault id	stmt no.	fault description
T1	9	change all $\leq$ to $<$
T2	11	omit: or ( $b + c \leq a$ )
T3	11	change to ( $b + c < a$ )
T4	14	add: or $err = true$
T5	15	change and to or
T6	19	omit: and ( $a < c$ )
T7	21	omit: or ( $a*a+c*c=b*b$ ) or ( $b*b+c*c=a*a$ )
T8	22	change all and to or
T9	23	change ( $a*a+b*b<=c*c$ )
T10	21	change all or to and
T11	21	omit: or ( $b*b+c*c=a*a$ )
T12	21	change ( $a*a+c*c=b*b$ ) to ( $a*a+c*c=b$ )
T13	21	change ( $a*a+b*b=c*c$ ) to ( $a*b+b*b=c*c$ )

**b. Overall execution, infection, and failure statistics**

fault id	no. of tests				exec rate%	infect rate%	failure rate% wrt		
	total	exec fault	caused infect	caused fail			total	exec	inf
T1	20000	20000	1017	0	100.0	5.1	0	0	0
T2	10000	8454	1487	1487	84.5	17.6	14.9	17.6	100.0
T3	10000	8398	76	76	84.0	.9	.76	.9	100.0
T4	20000	20000	11569	11569	100.0	57.8	57.8	57.8	100.0
T5	20000	8469	495	495	42.3	5.8	2.5	5.8	100.0
T6	20000	7724	0	0	38.6	0	0	0	---
T7	20000	8398	13	13	42.0	.16	.065	.16	100.0
T8	10000	4162	2305	2305	41.6	55.4	23.1	55.4	100.0
T9	20000	8439	0	0	42.2	0	0	0	---
T10	10000	4200	7	7	42.0	.17	.07	.17	100.0
T11	10000	4195	0	0	42.0	0	0	0	---
T12	10000	4162	5	5	41.6	.12	.05	.12	100.0
T13	10000	4115	2	2	41.2	.05	.02	.05	100.0

**c. Path properties**

fault id	total num	Properties of individual paths					
		sometimes infect	always infect	never infect	if infected, then		
					sometimes fail	always fail	never fail
T1	8	1	0	7	0	0	1
T2	7	2	0	5	0	2	0
T3	7	2	0	5	0	2	0
T4	13	0	7	6	0	7	0
T5	8	0	2	6	0	2	0
T6	3	0	0	3	0	0	0
T7	7	0	1	6	0	1	0
T8	6	0	2	4	0	2	0
T9	6	0	0	6	0	0	0
T10	6	0	1	5	0	1	0
T11	6	0	0	6	0	0	0
T12	7	0	1	6	0	1	0
T13	8	0	2	6	0	2	0

assign **err:=true**, discussed shortly). The selection of the right path gives its variable the correct value, and the selection of the wrong path gives its variable a wrong value and does not assign the correct value to the variable on the right path. Since a variable is never reassigned another value along any subsequent path, infection of a live variable must necessarily cause failure. Notice that the only assignments to non-output variables are to **pr**, **pa**, and **po**. Due to a limit on expression length in the DEFA system, the predicate expressions that would normally appear in the subsequent **if-else if** statement were computed and stored in these variables. This subtle change in the program changes the semantic behavior so that it is possible for **pa** or **po** to become infected due to a fault, but not cause failure if a previous predicate in the **if-else if** statement is true, since the infected variable is now dead. This possibility did not occur during the testing of fault T9.

Let's look at the one exception where **err:=true** occurs on either of two paths. Notice that fault T1 had a 5.1% infection rate with no failures. When we investigated the fact that this fault had no effect although it seemed like sides of length 0 could slip through and cause a failure, we found that the predicate in line 11 implies the predicate in line 9, making the line 9 predicate redundant. Since the fault inserted in line 9 never sets **err** to true when it should have been false, no failure can occur. This was not noticed when the program was written.

The reason that fault T6 resulted in no infection and no failure is a little

clearer. The omitted condition ( $a < c$ ) is redundant since it is implied by the falsity of ( $a = c$ ) in statement 17. In fact, the entire predicate in statement 19 is redundant, i.e. a triangle that is not a right triangle and not an acute triangle must be an obtuse triangle. The advantage of the redundant predicate versus the "catch-all" else is that a fault in one of the predicates in the **if-else if** structure might result in no assignment, which would make the output easier to identify as erroneous because one of the assignments must be made.

One problem with testing this program for the specified faults is that some of the faults have a very low infection rate. The usual structural coverage testing methods are little help in this situation, since they spread the testing effort among many paths while focused testing on a particular path is required to reveal the fault. Figure 6 contains the error statistics summary and path table for fault T7, which had the lowest infection and failure rate. The fault had an overall failure rate of 0.16% with respect to executions of the fault despite an execution rate for the fault of 42%, and only one of the 7 paths executed resulted in failure. A closer examination of the fault and its path explains the low failure rate and the fact that only one path caused the failure. Fault T7 omitted the other two conditions required to recognize a right triangle, so that roughly two thirds of right triangles were not recognized as such. There are very few of the over 1 million inputs that can be the integer sides of a right triangle, hence the low infection rate. As for the single path, only scalene right triangles can have integer sides, so the faulty path recognized the triangle as scalene,

**Figure 6. Error statistics summary and path table for fault T7**

**Error Set Statistics for Paths**  
-----

Path no.	# of execs	Infection		Failure		Infected or failed						
		no.	rate(%)	no.	rate(%)	Error set size						
						max	avg max	avg avg	avg last	avg#	avg dist	avg c/d
1	4509	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
2	3127	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
3	637	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
4	13	13	100.000	13	100.000	1	1.0	1.0	1.0	0.0	9	0.000
5	98	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
6	8	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
7	6	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0	0.000
tot	8398	13	0.155	13	0.155							

**Execution Paths**  
-----

Path no.	Total execs	Fault hit?	statement execution sequence
1	4509	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 19 20 21 22 23 24 26 28 29 30 31 31 31 31 31 31 31
2	3127	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 19 20 21 22 23 24 26 27 30 31 31 31 31 31 31 31 31
3	637	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 18 21 22 23 24 26 27 30 31 31 31 31 31 31 31 31
4	13	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 19 20 21 22 23 24 26 28 30 31 31 31 31 31 31 31 31
5	98	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 18 21 22 23 24 26 28 29 30 31 31 31 31 31 31 31 31
6	8	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 17 19 20 21 22 23 24 25 30 31 31 31 31 31 31 31 31
7	6	yes	1 2 3 4 5 6 7 8 8 8 9 11 13 14 15 16 21 22 23 24 26 27 30 31 31 31 31 31 31 31 31

Of 7 different paths found during testing, 7 caused execution of the fault  
 Total executions over all paths : 8398  
 Total executions of a path containing the fault : 8398  
 Execution rate for fault : 100.00%

but due to the omitted conditions, did not recognize it as a right triangle.

Clearly, we should concentrate testing on path 4 for this fault at this location. Is this a good path to test for any fault at this location? This question can never be answered in general because we cannot characterize all possible faults, but it seems reasonable to experiment with different faults at a location to see if there are one or two paths that occur frequently and that have relatively high failure rates w.r.t infection. This was done for statement 21 by testing faults T10-T13. Figure 7 summarizes the results of these experiments. Since each experiment was run on a different machine, with paths entered into their respective path tables in arbitrary order, the path numbers for paths with non-zero infection rates have been converted to correspond to the path numbers for T7. Fault T11 never resulted in infection because none of the relatively few inputs that could cause infection for this experiment were randomly generated; therefore, path 4, which always causes a failure by not selecting any angle property for the triangle, was never traversed. Path 4 would definitely be preferred in testing statement 21 for this set of faults, and possibly many other faults as well.

There is one additional observation concerning the maximum error set sizes for fault T13. Path 4 has a maximum error set size of 1, indicating that one variable (**rig**) did not get its correct value. The maximum error set size of 2 for path 6 implies that a triangle was mislabeled as a right triangle and not labeled as the appropriate

Figure 7. Experiments with various faults in statement 21

Error Set Statistics for Paths											
		Infection				Failure				Infected or failed	
		no.		rate(%)		no.		rate(%)		Error set size	
Path	# of	no.	rate(%)	no.	rate(%)	max	avg	avg	avg	avg#	avg
no.	execs					max	max	avg	last	canc	dist
											c/d
<b>T10</b>											
	2281	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	1547	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	302	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	60	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
4	7	7	100.000	7	100.000	1	1.0	1.0	1.0	0.0	9 0.000
	3	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
tot	4200	7	0.167	7	0.167						
<b>T11</b>											
	2228	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	1573	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	331	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	56	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	5	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	2	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
tot	4195	0	0.000	0	0.000						
<b>T12</b>											
	2297	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	296	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	1503	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	51	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	7	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
4	5	5	100.000	5	100.000	1	1.0	1.0	1.0	0.0	9 0.000
	3	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
tot	4162	5	0.120	5	0.120						
<b>T13</b>											
	2251	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	1499	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	295	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	6	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	55	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
	7	0	0.000	0	0.000	0	0.0	0.0	0.0	0.0	0 0.000
6	1	1	100.000	1	100.000	2	2.0	1.9	2.0	0.0	9 0.000
4	1	1	100.000	1	100.000	1	1.0	1.0	1.0	0.0	9 0.000
tot	4115	2	0.049	2	0.049						

kind of triangle.

We previously noted that structural coverage techniques would perform poorly in identifying low infection rate faults in this program, in particular with respect to a fault in statement 21, where only one path seems to have much chance of revealing an error. Consider the strongest coverage criterion, all-paths, and the strongest data flow coverage criterion, all-du-paths. The control flow graph for the Triangle program has 51 possible paths, of which only 8 are feasible (equilateral triangles must be acute, right triangles with integer sides must be scalene). Of the 51 possible paths, 48 include statement 21, and 6 of the 8 feasible paths include statement 21. Both all-paths and all-du-paths require coverage of all 48 paths to test this statement, and must deal with the large number of infeasible paths in this problem. Even if the infeasible paths are eliminated, these techniques have no reason to prefer any of the 6 feasible paths, and so, assuming equal coverage of the paths, would require roughly 6 times as much testing to reveal an error for a fault in statement 21 as compared with a technique that selects path 4 as the best path to test.

Little attention has been paid so far to the error set information gained in these experiments. Most of the error set statistics for paths with respect to a particular fault were very similar due to the fact that infections caused failure 100% of the time. The error set activity in the programs of the next sections is more interesting. However, dynamic error flow analysis of the Triangle program has yielded



some valuable insights into the semantics of the program and demonstrated that for a particular fault some paths are better to test than others.

#### **4.3.2 The Digitseq program**

The program in Figure 8 processes a 24-digit positive integer and returns the length of the longest sequence of even digits if the integer is even, and the length of the longest sequence of odd digits if the integer is odd. Leading zeroes are not counted as part of the integer. The integer is input as three unsigned 8-digit values stored in three long integer variables. Each experiment involved a different fault run with 20,000 randomly generated inputs, with each input consisting of 3 integers in the range 0 .. 99999999.

This program has a large number of paths (around  $10^{19}$ ), so when first tested the same path never appeared twice before the path table was filled, and all remaining paths were grouped together. To address this problem, various parts of the program were written as procedures so the trace could be turned off for the procedure body and only the net effect of the procedure on the data state recorded upon return from the procedure.

The location of the fault determines how such modularization is done. Clearly,

Figure 8. Digitseq program

```

-----
{ DigitSequence program }
{ Implements the following specification : }
{ }
{ The program processes an unsigned integer of up to 24 digits. }
{ If the integer is even, the program finds the length of the }
{ longest consecutive sequence of even digits within the integer, }
{ and if the integer is odd, the program finds the length of the }
{ longest consecutive sequence of odd digits within the integer. }
{ The program may only use simple integer (longint) variables. }
{ Input will consist of 3 positive integers of no more than }
{ 8 digits each representing, respectively, the high-order 8 digits, }
{ middle-order 8 digits, and low-order 8 digits of the 24-digit }
{ unsigned integer. Output will be an integer in the range 1-24 }
{ as determined above. }
-----

```

```

program DigitSeq;

```

```

var
  a, b, c : longint;           { holds 24-digit integer }
  d : longint;                 { done flag }
  s : longint;                 { flag : 1 - odd, 0 - even }
  p : longint;                 { current part of number : 0 - low, 1 - mid, 2 - high }
  x : longint;                 { holds current part of number }
  n : longint;                 { holds current digit }
  z : longint;                 { length of seq. of zeroes }
  L : longint;                 { length of longest sequence }
  w : longint;                 { length of current sequence }
  i : longint;

```

Figure 8. (cont.)

```
begin
1   d := 0; p := 0; w := 0; L := 0; z := 0;
6   read (a); read (b); read (c);

9   if (c mod 2) = 0 then           { determine even or odd }
10  s := 0
11  else
11  s := 1;

12  while d <> 1 do
12  begin
13  if p = 0 then begin           { get next 8-digit portion }
14  x := c; p := p + 1; end
17  else if p = 1 then begin
18  x := b; p := p + 1; end
21  else if p = 2 then begin
22  x := a; p := p + 1; end;

25  for i := 1 to 8 do           { search current portion }
25  begin
26  n := x mod 10;
27  x := x div 10;

28  if (n mod 2) = s then       { start counting sequence }
29  w := w + 1
30  else
30  begin
31  if w > L then               { new longest sequence }
31  L := w;
32  w := 0;
32  end;

33  if n = 0 then               { length of sequence of zeroes }
34  z := z + 1
35  else
35  z := 0;
35  end;

37  if p > 2 then               { done }
38  d := 1;
38  end;

40  if (s = 0) and (w > L) and (n = 0) then { adjust for leading 0's }
41  w := w - z;
42  if w > L then               { last sequence was longest }
43  L := w;
44  writeln (L)
end.
```

we want to avoid putting the fault in a module, along with any code directly affected by the fault. When the fault occurs near the end of the program, the code that is executed prior to the fault and not executed again is a good candidate for modularization. In some cases it is desirable to "collapse" part of a path containing a decision, especially if that part of the path is not expected to use infected variables. The presence of decisions within loops is the main source of the combinatorial explosion of paths, and sometimes loops can be modularized with a great reduction in the number of paths. All of these methods were used in the modularization of Digitseq.

The disadvantage of modularization is that it loses some error flow information. Error set activity that takes place inside a module is only summarized, and distance-related measures can be affected. Further investigation is necessary to determine the impact of modularization on error flow information and to develop appropriate guidelines for doing modularization. However, judicious use of this technique can reduce the number of paths to a manageable level.

The Appendix contains the listings for the modularized versions of Digitseq. The experimental results in Figure 9 are based on these versions. Figure 9a shows the different faults tested for this program, Figure 9b gives the overall results for execution rate, infection rate, and failure rate, and Figure 9c gives a breakdown of path properties.

**Figure 9.** Results of experiments with Digitseq program

**a. Digitseq program faults**

fault id	stmt no.	fault description	proc version
T1	40	change to (s = 1)	1
T2	40	change to (w > 1)	1
T3	40	change to (n = 1)	1
T4	41	change to w - 2	1
T5	37	change to p >= 2	1
T6	27	change to x div 100	4
T7	26	change to x mod 9	4
T8	26	change to x mod 5	4
T9	27	change to x div 5	4
T10	27	change to x div 2	4
T11	9	change to (c mod 2) = 1	4
T12	2	change to p := 1	4
T13	3	change to w := 1	4
T14	22	change to x := b	5

**b. Overall execution, infection, and failure statistics**

fault id	no. of tests				failure rate% wrt		failure rate% wrt		
	total	exec fault	caused infect	caused fail	exec rate%	infect rate%	total	exec	inf
T1	20000	20000	117	117	100.0	.59	.59	.59	100.0
T2	20000	20000	382	0	100.0	1.9	0	0	0
T3	20000	20000	114	114	100.0	.57	.57	.57	100.0
T4	20000	105	83	41	.53	79.0	.21	39.0	49.4
T5	20000	20000	20000	5718	100.0	100.0	28.6	28.6	28.6
T6	20000	20000	20000	17500	100.0	100.0	87.5	87.5	87.5
T7	20000	20000	20000	16322	100.0	100.0	81.6	81.6	81.6
T8	20000	20000	20000	16338	100.0	100.0	81.7	81.7	81.7
T9	20000	20000	20000	16480	100.0	100.0	82.4	82.4	82.4
T10	20000	20000	20000	15853	100.0	100.0	79.3	79.3	79.3
T11	20000	20000	20000	16301	100.0	100.0	81.5	81.5	81.5
T12	20000	20000	20000	6405	100.0	100.0	32.0	32.0	32.0
T13	20000	20000	20000	4588	100.0	100.0	22.9	22.9	22.9
T14	20000	20000	20000	6590	100.0	100.0	33.0	33.0	33.0

**c. Path properties**

fault id	Properties of individual paths						
	total num	sometimes infect	always infect	never infect	if infected, then		
					sometimes fail	always fail	never fail
T1	4	1	0	3	0	1	0
T2	6	1	0	5	0	0	1
T3	4	1	0	3	0	1	0
T4	2	2	0	0	1	1	0
T5	6	0	6	0	4	2	0
T6	4	0	4	0	4	0	0
T7	6	0	6	0	6	0	0
T8	6	0	6	0	6	0	0
T9	6	0	6	0	6	0	0
T10	5	0	5	0	5	0	0
T11	6	0	6	0	6	0	0
T12	6	0	6	0	6	0	0
T13	6	0	6	0	3	0	3
T14	5	0	5	0	2	3	0

Unlike the Triangle program, the Digitseq program exhibits different path failure rates with respect to infection for a number of faults. Figure 10 contains the results for a few such faults. Note that in most of the experiments (D5-D14) there were faults that had paths with 100% execution and infection rates, yet their failure rates varied from 23% to 88%. This indicates that cancellation is indeed a problem for testing these faults. Different paths through the same fault can have a wide variation in cancellation activity, as shown in Figure 10. This indicates that some paths through a fault may be better for testing than others. There is also a difference in the maximum average error set size statistic for different paths through the same fault in many cases. The general trend in these experiments was that for a given fault this statistic was higher for paths with higher failure rates, as demonstrated in Figure 10a.

Another statistic of interest in Figure 10 is "average cancellation per unit distance". This statistic is obtained by averaging the ratio for each test of the number of cancellations to distance, where distance has been defined in section 4.1, in an attempt to give a normalized measure of cancellation activity. Normalization is necessary in order to compare paths that have a few cancellations over a short distance with paths that may have a number of cancellations over a long distance. It is intuitive that a path with a lower failure rate w.r.t. infection should exhibit more cancellation activity than one with a higher failure rate. The experimental results support this intuition.

Figure 10a. Digitseq experimental results

Error Set Statistics for Paths												
						Infected or failed						
Path	# of no. execs	Infection		Failure		Error set size						
		no.	rate(%)	no.	rate(%)	max	avg max	avg avg	avg last	avg# canc	avg dist	avg c/d
D4												
1	78	62	79.487	20	25.641	2	1.3	1.1	1.3	0.0	2	0.000
2	27	21	77.778	21	77.778	2	2.0	1.5	2.0	0.0	3	0.000
tot	105	83	79.048	41	39.048							
D5												
1	9053	9053	100.000	2360	26.069	5	3.0	2.2	3.0	0.0	4	0.000
2	9328	9328	100.000	2395	25.675	5	3.0	2.2	3.0	0.0	4	0.000
3	790	790	100.000	428	54.177	5	4.0	2.8	3.5	0.5	5	0.092
4	673	673	100.000	379	56.315	5	4.0	2.8	3.5	0.4	5	0.087
5	76	76	100.000	76	100.000	5	4.8	3.7	4.8	0.1	6	0.013
6	80	80	100.000	80	100.000	5	4.8	3.5	4.7	0.1	5	0.023
tot	20000	20000	100.000	5718	28.590							
D7												
1	580	580	100.000	497	85.690	4	3.9	2.3	1.9	11.8	109	0.137
2	9065	9065	100.000	7576	83.574	4	3.9	2.3	1.5	12.0	110	0.173
3	9502	9502	100.000	7550	79.457	4	3.9	2.2	1.3	12.6	104	0.219
4	577	577	100.000	473	81.976	4	3.9	2.3	1.8	12.1	107	0.139
5	135	135	100.000	109	80.741	4	3.9	2.3	2.8	12.0	107	0.157
6	141	141	100.000	117	82.979	4	4.0	2.4	2.9	11.7	115	0.123
tot	20000	20000	100.000	16322	81.610							
D8												
1	9608	9608	100.000	7629	79.403	4	3.8	2.1	2.0	12.8	90	0.314
2	9176	9176	100.000	7676	83.653	4	3.8	2.2	2.1	11.3	98	0.220
3	554	554	100.000	491	88.628	4	3.8	2.2	2.4	11.2	99	0.159
4	421	421	100.000	329	78.147	4	3.8	2.1	2.2	12.6	94	0.202
5	127	127	100.000	120	94.488	4	3.9	2.3	3.2	11.1	98	0.156
6	114	114	100.000	93	81.579	4	3.8	2.3	2.8	11.1	98	0.164
tot	20000	20000	100.000	16338	81.690							
D12												
1	9055	9055	100.000	3013	33.274	6	5.7	3.8	0.3	9.5	100	0.096
2	9136	9136	100.000	3011	32.958	6	5.8	3.8	0.3	9.7	100	0.097
3	927	927	100.000	187	20.173	6	5.7	3.8	0.2	9.6	100	0.096
4	704	704	100.000	126	17.898	6	5.7	3.7	0.2	9.7	100	0.097
5	86	86	100.000	24	27.907	6	5.8	3.8	0.5	9.5	101	0.094
6	92	92	100.000	44	47.826	6	5.9	3.8	0.9	9.4	100	0.094
tot	20000	20000	100.000	6405	32.025							

Figure 10b. Digitseq experimental results

Error Set Statistics - Detail														
	Infected, no failure							Infected and failed						
	Error set size							Error set size						
Path no.	max	avg max	avg avg	avg last	avg# canc	avg dist	avg c/d	max	avg max	avg avg	avg last	avg# canc	avg dist	avg c/d
<b>D4</b>														
1	1	1.0	1.0	1.0	0.0	2	0.000	2	2.0	1.3	2.0	0.0	2	0.000
2	0	0.0	0.0	0.0	0.0	0	0.000	2	2.0	1.5	2.0	0.0	3	0.000
total	no. -	42		rate % -		40.000		no. -	41		rate % -		39.048	
<b>D5</b>														
1	4	2.7	2.0	2.7	0.0	4	0.000	5	3.8	2.6	3.8	0.0	4	0.000
2	4	2.7	2.0	2.7	0.0	4	0.000	5	3.8	2.6	3.8	0.0	4	0.000
3	5	4.0	2.7	3.0	1.0	5	0.200	5	3.9	3.0	3.9	0.0	5	0.000
4	5	4.0	2.7	3.0	1.0	5	0.200	5	4.0	3.0	4.0	0.0	5	0.000
5	0	0.0	0.0	0.0	0.0	0	0.000	5	4.8	3.7	4.8	0.1	6	0.013
6	0	0.0	0.0	0.0	0.0	0	0.000	5	4.8	3.5	4.7	0.1	5	0.023
total	no. -	14282		rate % -		71.410		no. -	5718		rate % -		28.590	
<b>D7</b>														
1	4	3.9	2.3	1.0	12.5	117	0.118	4	3.9	2.3	2.0	11.7	108	0.141
2	4	3.8	2.0	0.6	12.8	69	0.397	4	3.9	2.4	1.6	11.8	118	0.129
3	4	3.7	1.9	0.6	13.5	62	0.492	4	3.9	2.3	1.6	12.4	115	0.148
4	4	3.9	2.3	0.9	13.3	115	0.132	4	3.9	2.3	2.0	11.8	106	0.140
5	4	3.8	2.0	1.8	14.0	58	0.365	4	4.0	2.4	3.1	11.5	119	0.107
6	4	4.0	2.4	2.1	13.6	120	0.132	4	3.9	2.4	3.1	11.3	114	0.121
total	no. -	3678		rate % -		18.390		no. -	16322		rate % -		81.610	
<b>D8</b>														
1	4	3.6	1.8	1.3	13.4	35	0.861	4	3.9	2.2	2.2	12.7	104	0.172
2	4	3.7	1.9	1.3	12.0	43	0.650	4	3.9	2.2	2.3	11.2	109	0.136
3	4	3.8	2.2	1.4	11.9	108	0.165	4	3.8	2.2	2.5	11.1	97	0.159
4	4	3.8	2.2	1.5	12.9	104	0.163	4	3.8	2.1	2.4	12.5	92	0.212
5	4	4.0	2.2	2.1	13.1	105	0.150	4	3.9	2.3	3.2	11.0	98	0.156
6	4	3.4	2.1	2.2	11.0	53	0.342	4	3.9	2.3	3.0	11.1	108	0.123
total	no. -	3662		rate % -		18.310		no. -	16338		rate % -		81.690	
<b>D12</b>														
1	6	5.7	3.7	0.0	9.8	99	0.099	6	5.8	3.9	1.0	9.1	101	0.090
2	6	5.8	3.7	0.0	9.9	99	0.100	6	5.8	3.9	1.0	9.2	101	0.091
3	6	5.7	3.7	0.0	9.9	100	0.099	6	5.8	3.9	1.0	8.8	102	0.086
4	6	5.7	3.7	0.0	9.9	100	0.099	6	5.8	3.8	1.0	9.0	102	0.088
5	6	5.8	3.7	0.0	9.9	100	0.098	6	6.0	3.9	2.0	8.5	103	0.083
6	6	5.8	3.7	0.0	10.1	99	0.102	6	6.0	3.9	2.0	8.7	102	0.085
total	no. -	13595		rate % -		67.975		no. -	6405		rate % -		32.025	



Figure 10b shows a detailed breakdown of error set statistics for those tests that infected but did not fail versus those that infected and failed. We observe that for a particular path the group of tests that ultimately failed generally show a higher average maximum error set size than those that resulted in coincidental correctness. Also, the same group showed a lower average c/d than the coincidentally correct group, indicating that the lower cancellation activity allowed more errors to propagate to failure.

### **4.3.3 The Cancel program**

The program in Figure 11 accepts 3 integer inputs and computes an arbitrary function. This program uses a number of arithmetic operations that exhibit expression cancellation, such as integer division, modulus, absolute value, squaring, and potential multiplication by zero. Each experiment involved a different fault run with 20,000 randomly generated inputs, with each input consisting of 3 integers in the range -50 .. 50. Figure 12a shows the different faults tested for this program, Figure 12b gives the overall results for execution rate, infection rate, and failure rate, and Figure 12c gives a breakdown of path properties.

The trends observed in the Digitseq experiments relating the average maximum error set size to a path's failure rate still hold in the Cancel experiments,

Figure 11. Cancel program

```

-----
{ Cancel program }
{ }
{ Takes 3 integer inputs and computes an arbitrary function using }
{ statements with different potentials for resistance and }
{ cancellation. }
-----
program cancel;

var
  a, b, c, d, e, f, g, h, x, y, z : longint;
begin
  1 readln (a, b, c);
  2 d := 0; e := 0; f := 0; g := 0; h := 0;
  7 if a > c then
    begin
  8 d := c*c - a*a;
  9 e := b * (abs(a) - abs(c));
  10 if d + e > 0 then
  11 f := trunc(sqrt(d+e))
    else
  12 f := trunc(sqrt(abs(d+e)));
  13 g := a - c;
  14 h := abs(g div 5) + 1;
  15 while g > 0 do
    begin
  16 d := d + e mod h;
  17 g := g - h
    end;
  18 if d div (abs(b)+1) > c*c then
  19 d := (d - c) * a
    end
    else
    begin
  20 d := (c-a) * (b-a) * (b-c) * (b-10);
  21 if d > 0 then
    begin
  22 e := b * d + (b - a*a) * c;
  23 if d + e <> 0 then
  24 f := d * e div (d + e);
  25 g := 1;
  26 while g <= 5 do
    begin
  27 g := g + 1;
  28 c := c + 2;
  29 a := a - c
    end;
  30 h := (c + a) div 2
    end
    else
    begin
  31 e := abs(b * d - a*a);
  32 f := e * (a + b + c);
  33 if abs(f - e) < a*a then
  34 h := (d + a) div 10
    else
  35 h := (d + b) div 10
    end
    end;
  36 x := b * (g - h);
  37 y := abs(f * (g - a));
  38 z := (e * h) div (abs(c)+1);
  39 writeln (x, y, z)
end.

```

**Figure 12. Results of experiments with Cancel program**

**a. Cancel program faults**

fault id	stmt no.	fault description
C1	8	change to a+a
C2	9	change to b * (a - c)
C3	15	change to g >= 0
C4	19	change to (d - b) * a
C5	24	change to e div d + e
C6	26	change to g < 5
C7	29	missing stmt
C8	32	change to (a + b - c)
C9	34	change to div 100

**b. Overall execution, infection, and failure statistics**

fault id	no. of tests				exec rate%	infect rate%	failure rate% wrt		
	total	exec fault	caused infect	caused fail			total	exec	inf
C1	20000	9908	9822	8962	49.5	99.1	44.8	90.5	91.2
C2	20000	9872	7287	7261	49.4	73.8	36.3	73.6	99.6
C3	20000	9994	1655	1655	50.0	16.6	8.3	16.6	100.0
C4	20000	0	0	0	0	---	0	---	---
C5	20000	4839	4839	4827	24.2	100.0	24.1	99.8	99.8
C6	20000	4809	4809	4808	24.0	100.0	24.0	99.98	99.98
C7	20000	4832	4832	4778	24.2	100.0	23.9	98.9	98.9
C8	20000	5421	5380	5258	27.1	99.2	26.3	97.0	97.7
C9	20000	32	30	30	.16	93.8	.15	93.8	100.0

**c. Path properties**

fault id	total num	Properties of individual paths					
		sometimes infect	always infect	never infect	if infected, then		
					sometimes fail	always fail	never fail
C1	11	10	1	0	10	1	0
C2	10	10	0	0	10	0	0
C3	8	6	2	0	6	2	0
C4	0	0	0	0	0	0	0
C5	1	0	1	0	1	0	0
C6	1	0	1	0	1	0	0
C7	1	0	1	0	1	0	0
C8	2	1	1	0	1	1	0
C9	1	1	0	0	1	0	0

**Figure 13. Fault C2 in Cancel program**

**Error Set Statistics for Paths**

Path no.	# of execs	Infection		Failure		Infected or failed						
		no.	rate(%)	no.	rate(%)	Error set size						
						max	avg max	avg avg	avg last	avg#	avg canc	avg dist
1	797	730	91.593	723	90.715	5	4.3	2.5	4.3	0.0	24	0.000
2	3626	3494	96.360	3494	96.360	5	4.8	2.8	4.8	0.0	27	0.000
3	809	167	20.643	167	20.643	5	4.3	2.5	4.3	0.0	24	0.000
4	3719	2403	64.614	2403	64.614	5	4.7	2.8	4.7	0.0	27	0.000
5	279	251	89.964	241	86.380	5	3.9	2.3	3.9	0.0	21	0.000
6	83	73	87.952	72	86.747	4	3.7	2.3	3.7	0.0	18	0.000
7	101	90	89.109	82	81.188	4	3.1	2.1	3.1	0.0	15	0.000
8	101	15	14.851	15	14.851	4	3.5	2.1	3.5	0.0	15	0.000
9	273	54	19.780	54	19.780	5	4.1	2.4	4.1	0.0	21	0.000
10	84	10	11.905	10	11.905	4	3.9	2.4	3.9	0.0	18	0.000
tot	9872	7287	73.815	7261	73.551							

**Execution Paths**

Path no.	Total execs	Fault hit?	statement execution sequence
1	797	yes	1 1 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
2	3626	yes	1 1 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 15 16 17 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
3	809	yes	1 1 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
4	3719	yes	1 1 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 15 16 17 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
5	279	yes	1 1 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
6	83	yes	1 1 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 15 16 17 15 18 36 37 38 39 39 39
7	101	yes	1 1 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 15 18 36 37 38 39 39 39
8	101	yes	1 1 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 15 18 36 37 38 39 39 39
9	273	yes	1 1 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 15 16 17 15 16 17 15 18 36 37 38 39 39 39
10	84	yes	1 1 1 2 3 4 5 6 7 8 9 10 12 13 14 15 16 17 15 16 17 15 18 36 37 38 39 39 39

although not as clearly since several of the experiments only traversed one path. The average c/d statistic is usually zero in most of the experiments, but this is probably due to the fact that it only measures assignment cancellation and not expression cancellation.

We will use these experiments to show the utility of dynamic error flow analysis in discovering new testing methods by comparing the performance of coverage-based testing techniques against a technique that selects paths for testing using error set information. We will also demonstrate its usefulness in comparing other testing methods against each other. For both demonstrations we will use fault C2. Figure 13 gives the breakdown by paths for the statistics from this experiment and a description of the paths.

Our proposed testing technique, which we will refer to as ES1, selects for testing the path(s) with the largest observed average maximum error set size. We will compare its performance in detecting this fault with that of all-paths and all-du-paths. We proceed as follows :

- 1) Identify the paths to test according to all-paths, all-du-paths, and ES1 with respect to the given fault by presuming we know or suspect the fault is at the analyzed location.

- 2) For the purpose of comparing the testing techniques, we assume that we have a way to randomly select input data that drives a particular path. We must then also assume away the infeasible paths problem for all-paths and all-du-paths, which is a source of additional cost in these techniques.
  
- 3) Determine how many tests we need to run using each technique in order to achieve 99.999% confidence of observing erroneous output if the fault is present.

The Cancel program has a total of 28 possible paths, of which 24 contain the fault location, statement 9. The loop at statement 15 can be shown to execute between 0 and 5 times. The all-du-paths strategy identifies 10 paths, since the loop at statement 15 will only be executed 0, 1, or 2 times and the assignment at statement 19 is a dead assignment. The loop at statement 26 is really one path since it is always executed 5 times. Of the 10 all-du-paths, 6 contain the fault location. Using the path table in Figure 13, we find the path number for each path under consideration in each testing strategy, with the following results :

**all-paths -- Of the 24 paths :**

**4 are infeasible since the loop at statement 15 cannot execute 0 times**

**10 paths (found in the path table, Figure 13) are feasible.**

**10 are semantically equivalent to the previous 10 paths since the value assigned to *d* in statement 19 has no effect on output**

**all-du-paths -- Of the 6 paths :**

**2 are infeasible, as above**

**4 are feasible - paths 6,7,8,and 10**

**ES1 -- An examination of Figure 13 shows that path 2 has the highest average maximum error set size.**

We begin the testing by choosing a random input that drives a path selected by the testing method under consideration. For testing methods that choose multiple paths, we presume there is no reason to prefer any path over another, so we stipulate that the tests must be distributed among all paths so that no path has ever currently been tested more than one additional time as compared with all other paths in the group. Our testing procedure assures this by ordering the paths in the group as a cycle with the first path following the last path, and testing each path in turn. The testing process stops whenever an error is observed or the desired confidence level

has been attained. Of course, any testing method can get lucky and observe an error early in the process, but from a probabilistic standpoint will reveal errors based on the actual probability of failure for the path driven.

The above testing procedure is performed, and (assuming no errors are observed) the estimated failure rates for each path (Figure 13) are used to compute a 99.999% confidence that the fault is not present by

$$P_C = (1-p_1)*(1-p_2)* \dots *(1-p_n)*(1-p_1)* \dots \text{ until } P_C \leq .00001,$$

where  $p_i$  is the estimated probability of failure for path  $i$ ,  $n$  is the number of paths in the group, and  $P_C$  is the probability that correct output was observed on every test.

The results of this comparison for fault C2 are given in Figure 14. Note that the ordering of the paths can cause a wide variation in the number of tests required by all-paths. Testing strategy ES1 is the clear favorite, which is not surprising since the path with the largest average maximum error set size also had the highest failure rate in this instance. While this is not universally true, it has occurred enough in our limited experimentation to make this error flow characteristic a strong candidate for use in a testing technique. We have performed the above comparison for all faults in the Cancel program whose experiments identified more than one path. The results are summarized in Figure 14. Observe that all-du-paths performed the best for fault C3. For this fault, ES1 could not distinguish any subset of the paths. It is interesting



to note that all paths for C3 had 100% failure w.r.t. infection. As in the Triangle program, this situation did not lead to significantly different error set activity.

We can also use this technique to compare the performance of other existing testing methods, many of which are not comparable by other means. The outcome of the comparison is dependent on the programs and faults chosen, so this method cannot definitively pronounce one technique superior to another. However, consistently favorable outcomes for one technique over another using a wide variety of programs and faults gives an empirical basis for preferring the former. We may also investigate the marginal performance of a superior but more costly technique against other techniques.

The results in Figure 14 that were used to evaluate ES1 also allow us to compare the performance of all-paths versus all-du-paths. The results of the comparison are mixed -- all-paths performed better in the best case but worse in the worst case. The techniques were fairly even otherwise in this very limited comparison.

#### **4.4 Comments on experimental results**

The experimental results described above show the utility of dynamic error flow information in analyzing programs and execution paths, and comparing the

**Figure 14.** Comparison of testing techniques for fault C2

fault	all-paths				all-du-paths				ES1			
	paths	path# order	best case	worst case	paths	path# order	best case	worst case	paths	path# order	best case	worst case
C1	1-11	5	1	7	5,11	5	5	6	9,10	1	1	2
C2	1-12	7	6	12	6,7, 8,10	10	10	12	2	4	4	4
C3	1-8	6	1	7	6,8	1	1	1	1-8	6	1	7
C8	1,2	2	1	2	1,2	2	1	2	2	1	1	1

performance of other testing methods. As we have seen in the above examples, we can learn much about a program by examining the tables of error flow statistics for various faults. A path where an infection never causes failure is indicative of a redundant computation that could be removed. Infrequently executed paths may not give statistics representative of their behavior and should be investigated more thoroughly. Paths that always infect are good candidates for testing, especially if accompanied by a reasonable failure rate. Those paths that sometimes infect and sometimes fail are indicative of complex semantics involving resistance and cancellation. Error set statistics also provide useful insight into a program's behavior. The maximum error set size achieved in any test for a path is a good indication of the number of variables that are the target of assignments in a chain of data flow along the path. The average maximum error set size, which we have associated with failure rate in our experiments, gives an idea of the typical degree to which an infected variable propagates to other variables. The distance statistic gives a feel for the execution distance from fault to output, which might not be easy to discern from the program's code. Finally, the average cancellation per unit distance indicates the degree of assignment cancellation activity along the path.

The error flow statistics presented show apparent correlation with path failure rate in several instances, and we would clearly prefer to test paths through a potential fault that have high failure rates rather than those with low failure rates. Unfortunately, we cannot make many statistical claims as a result of this

experimentation, for a variety of reasons [Joh91]. The primary problem is that of having a representative sample on which statistical claims can be built. We can only write an infinitesimal number of the programs (i.e. computable functions) that can be written, so it is impossible to ever claim that a set of programs constitutes a representative sample. The same is true of faults in programs. With respect to a given fault in a given program, we can claim that the execution, infection, and failure rates observed experimentally for each path closely estimate the actual rates for the input domain from which the random samples were drawn, provided each path is executed a sufficient number of times. The statistics for average maximum error set size, average final error set size, and average number of cancellations are reliable estimates within a path, but comparing these measures between paths raises complex statistical issues that we have not yet addressed. It is not even clear that distance-related measures within the same path are statistically meaningful, since the underlying distribution of distances with respect to a path is not known.

At present we only report the results of these experiments and observe some intuitively appealing trends that we think bear further investigation. We believe, however, that error flow statistics, such as average maximum error set size, will continue to show a relationship to failure rate and provide a basis for the development of a family of error flow testing methods.

#### 4.5 Comments on dynamic error flow analysis

Experiments in dynamic error flow analysis measure the effect on a program's behavior of known faults. Software testing attempts to reveal unknown faults, so the application of error flow analysis to testing must address this issue. One possible testing technique based on dynamic error flow analysis would be to create several mutants for each program statement and perform the analysis on each mutant. This approach is similar to fault sensitivity analysis, but the product of the analysis would be an optimal set of code-covering paths to be tested, rather than the location sensitivity information produced by fault sensitivity analysis.

A central issue in all testing methods that use the mutation technique is the degree to which mutant faults mimic the effect of actual faults on program behavior, and in practice they have proven effective in this respect [DeM78, Off89]. We believe that error flow characteristics derived from mutant faults will generally reflect the error flow characteristics of any faults actually present, making dynamic error flow analysis an effective tool for investigating program behavior.

Due to the cost of dynamic analysis, it would be very desirable to develop static error flow analysis strategies that give a close approximation of the error flow characteristics of a program. Static analysis raises undecidability issues that limit its ability to determine semantic behavior in the general case, although in many

instances useful semantic information can be derived statically. One limitation of static analysis mentioned in Chapter 1 involves the infeasible path problem. Clearly, static analysis can identify as infeasible a path through unreachable code, i.e. an unlabeled FORTRAN statement preceded immediately by a GOTO. More sophisticated analysis could possibly determine that the preceding predicates along a path force a subsequent predicate to be always true or always false. Other information from the program specification or from assertions in the code might allow a similar logical deduction.

Perhaps the greatest limitation of static error flow analysis concerns the determination of coincidental correctness properties. As discussed in Section 3.4, complete characterization of coincidental correctness requires knowledge of the distribution of possible data states at each location in the code, which is generally unattainable by static analysis. The degree to which statically derived information can approximate coincidental correctness properties remains an open question in the development of static error flow analysis. As an alternative to pure static analysis, a hybrid static/dynamic technique might be developed that allows dynamically derived estimates to be used to compensate for insufficient statically derived information.

## 5.0 Estimating error flow behavior through static analysis

Static error flow analysis attempts to estimate a program's error flow behavior through syntactic and semantic analysis of the program's code. Static analysis is much less costly than dynamic analysis, and is preferable if it can produce reasonable estimates of the actual behavior. This chapter presents the underlying concepts of static error flow analysis, and presents some preliminary exploration into one possible static error flow testing model : the threshold model.

### 5.1 Static analysis techniques

Standard flow graph-theoretic techniques, such as data flow analysis, form the basis of our static analysis. These are augmented with semantic information that is used to capture potential data state characteristics. Our model of static error flow analysis estimates the actual error sets a program might produce with *static error sets*, which we will simply refer to as *error sets* in this chapter. The *error degree* at a given point in the program is the size of the corresponding error set. To facilitate construction of the error sets we will represent the program as a *control/data (CD) flowgraph*. A CD flowgraph is a standard program control flowgraph augmented with

directed arcs to indicate data flow. Nodes contain executable statements, or their statement numbers. Unlabeled arcs represent control flow paths between statements. An arc labeled with a variable  $v$  represents a data flow path from the definition of  $v$  to a reference to  $v$  that uses that definition. For two nodes  $a$  and  $b$  in graph  $G$ ,  $a$  *dominates*  $b$  in  $G$  iff every path that reaches  $b$  must first go through  $a$ .

Our strategy in error flow testing is to presume that a particular statement contains a fault that infects the succeeding data state, and then track the spread of the infection via data flow analysis along various paths in the program. Data flow analysis is particularly appropriate here because the  $dr$  pair is the mechanism by which an infection propagates from one variable to another. As we follow a particular path, the error degree may increase through propagation, decrease through cancellation, or remain the same. If we can find a set of paths that are very likely to propagate the error to output, then testing one or more of those paths and observing correct output gives us confidence that the original statement is correct. This conclusion may be false if we have overestimated the error degree because the path would be less likely than expected to propagate an error, which could result in coincidentally correct output. Execution with different input might in fact reveal the error. We may achieve confidence in the entire program by applying the above strategy to each statement in the program. Figure 1 shows a hypothetical error flow testing system.



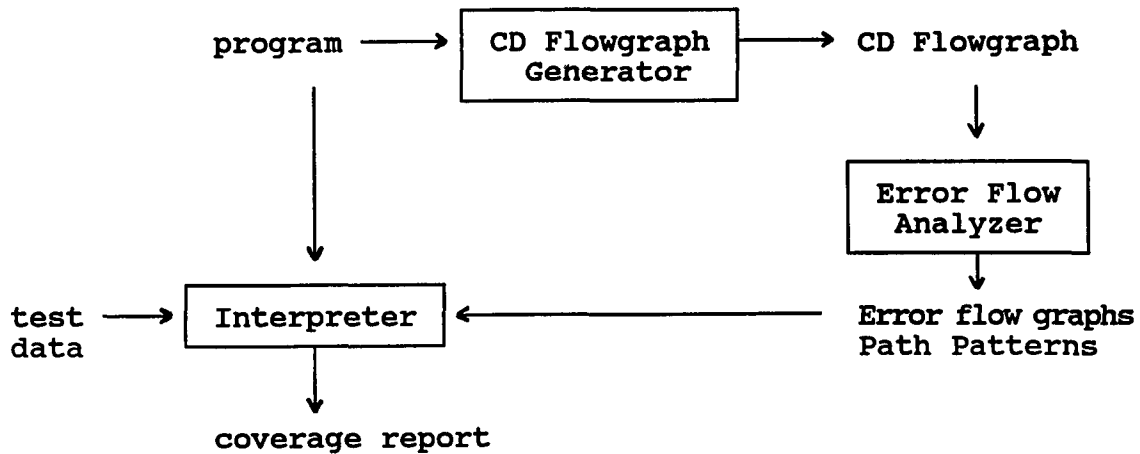


Figure 1.

Let's examine the static approximation of error sets in more detail. For the moment we will ignore the cancellation issue. A variable is placed in an error set when it is assigned an incorrect value and removed when its value is no longer incorrect. A statement *inherits* an error set from its executional predecessor and possibly modifies it by adding or deleting a variable. We could determine error sets for paths strictly through control flow analysis, but this might involve passing the error set through many statements or along many paths that leave it unchanged. Since data flow is the model for propagation of an infection, it would seem more efficient to simply follow data flow arcs from the infection and add the newly infected variables to the corresponding error sets. We advocate this approach, but observe that data flow alone is not sufficient to correctly determine an error set. The reason is simple : assignments that could result in addition or deletion of variables in the

error set may lie along a control path between the definition of an error and a subsequent reference. Of course, we need not worry that the variable defined as an error will leave the error set of the associated reference due to an intervening assignment. That is prohibited because the paths selected to test this *dr* pair must be definition-clear for that variable. Other variables, however, could be affected, as demonstrated by the following example. Error sets are enclosed in [].

<u>stmt</u>		
		[a]
1	b := a + 1	[a,b]
2	c := b * 2	[a,b,c]
3	a := 10	[b,c]
4	d := b + 5	[b,c,d]

It is clear that the error set [b,c,d] after statement 4 is the correct one. If we had followed data flow alone, we would have the reference to **b** in statement 4 inherit the error set of the definition of **b** in statement 1, and add the target variable **d** in statement 4, giving the error set after statement 4 as [a,b,d].

Predicates are a particular problem in determining error sets. A different error set may result from taking a different path, and the path taken can only be known upon execution. We certainly want to follow and create error sets for all *dr* pairs that propagate an error along different paths, but what about predicates that lie along the

path between  $d$  and  $r$  ? Suppose we replace statement 3 in the previous example with

```
3.1  if P then
3.2    a := 10
      else
3.3    a := a + 1
```

Depending on the path taken,  $a$  will either be present or absent in the error set following statement 4. If we create error sets for all possible paths, we gain accuracy in our estimation of the error sets at the expense of efficiency. Clearly, we desire some tradeoff that will give us an efficient way of obtaining reasonably accurate error sets.

Cancellation is perhaps the biggest problem we face in constructing error sets. We must consider the possibility that an assignment statement whose expression involves variables in error may still produce the correct result due to this phenomenon, causing the target variable to be removed (or not added) to the subsequent error set. We must take cancellation into account to avoid overestimating error sets, which might lead us to incorrect conclusions about error flow behavior. Cancellation in predicates is worst of all, and very common since predicates map a potentially  $n$ -dimensional input space to [false, true]. Again, we don't know until execution which branch is taken, and whether or not it is the correct branch. Note that a predicate may manifest an error in two ways : a fault in the predicate expression or a correct predicate evaluated with at least one erroneous argument. Let's examine the consequences of executing a predicate under either of these

circumstances. Sometimes the correct branch may be taken due to cancellation, i.e. the incorrect expression produces the same result as the correct expression or the effects of the infected variables are cancelled by the expression. In this case no additional variables become infected or uninfected due to the problem with the predicate since it has no detrimental effect on the subsequent execution path. The above method of determining error sets through data flow and/or control flow still works. However, suppose the wrong branch is taken. Now the program counter is corrupted and things can get bad rather quickly. Any assignments made along the wrong path are not supposed to be made, and any assignments on the correct path are not being made when they should. This situation should result in rapid growth of the error sets along either path. The problems are how to statically estimate the proportion of the time the wrong path is taken in order to decide which error sets apply, and how to construct the resulting error sets by comparing the execution of two separate paths.

Error flow testing has several important characteristics that make it an attractive testing strategy :

#### Static computation of propagation paths

The CD flowgraph can be created using relatively efficient data flow algorithms, and then used to construct error sets that determine the set of paths that have the highest degree of error propagation for a particular infection point.

### Automatable

Since all of this computation is driven by the original program structure, it can be automated to deliver an optimal set of test paths to the tester. Other portions of the procedure, such as matching the paths actually tested against those required, are also amenable to automation.

### Errors in the data state are easy to describe and track

Syntactic descriptions of faults, such as those in fault-based testing, can be difficult to classify or describe. The manifestation of a fault as an error in the data state is a simple binary description : right or wrong.

### Some missing code faults can be analyzed

We can analyze the effect of a missing assignment statement at any point in the program by doing an error flow analysis for each variable in the corresponding data state. Although this adds considerably to the expense of testing, it gives a capability not possible in most testing methods. Analysis of missing predicates, however, seems intractable due to the combinatorics of the possible paths.

We have described the general strategy of error flow testing and discussed some of the problems. Recall that the goals of error flow testing are to approximate the actual error sets that would be observed during program execution, to do so with

static analysis to the extent possible, and to use the resulting error sets to make positive statements about program correctness. Implementation of an error flow testing strategy requires the formulation of a model that is consistent with these goals. Many models may be developed that achieve these goals with varying degrees of effectiveness. Some considerations in constructing an error flow model include the representation chosen for error sets, the amount of path analysis to be undertaken, any simplifying assumptions and their impact, how to handle cancellation, and the manner in which test results are interpreted. The next section describes a particular model for error flow testing called the *threshold model*.

## 5.2 The threshold model of error flow testing

The error flow testing procedure we will describe is based on the fundamental assumption that once the error degree has reached some threshold, it is very unlikely that the infection would be cancelled by the remaining computation. This is consistent with our experimental results involving tests based on the average maximum error set size (ES1). We apply our error flow analysis procedure to each statement in the program by postulating an infection at that point. In the simple language used in our examples we only have two types of statements to consider : assignments and predicates. The predicates for **if** and **while** are treated similarly and we assume **input/output** statements are correct.

If the infection point is an assignment statement, we put the target variable in the error set and follow the data flow to all references, considering each  $dr$  pair in turn (Figure 2a). In the interest of preventing the overestimation of error sets, we will assume that any predicates lying on a path between a definition  $d$  and a reference  $r$  result in the correct branch being taken regardless of whether the predicate contains a fault or references infected variables. Furthermore, we will disregard the effect on an error set of any statements guarded by such predicates. We will, however, consider the impact on an error set of any assignment statement guaranteed to be executed between  $d$  and  $r$  (Figure 2b). These statements are (control) dominators of  $r$  in the subgraph with start node  $d$  and final node  $r$ . Ordering these statements by the dominance relation gives the control sequence in which we use the statements to produce successive error sets between  $d$  and  $r$ . The resulting error set that precedes  $r$  is then modified by the action of  $r$ . If  $r$  is an assignment statement, we add its target variable to the error set and follow the data flow to all references to that variable, repeating the above procedure. If  $r$  is a predicate, we assume the correct branch is taken and terminate that particular chain of data flow.

If the infection point is a predicate statement, treating it the same as predicates in the above case would essentially ignore the statement, since all error sets would remain null. We therefore assume that the wrong branch is taken and add to the error set the target variable of every unguarded assignment statement in the scope of the predicate until we reach the threshold or leave the scope of the

predicate. For **if-else** statements this means both branches. As above, we choose to ignore nested predicates. If we arrive at the end of the predicate's scope and the resulting error set has not reached the threshold, we handle each variable in the error set as an infected assignment and apply the previous procedure.

A trace of the above construction of error sets defines an *error flow graph* for an infection at a particular statement. This procedure is formally specified by the algorithms in Figure 2. Nodes in the error flow graph consist of a statement and its succeeding error set. The graph is essentially tree-structured with some arcs looping back when a previously encountered node is produced. Some arcs are labeled  $\bar{m}$  to prohibit a path through node  $m$  which would redefine the variable defined by a particular *dr* pair. Terminal nodes either satisfy the error threshold or have no subsequent data flow. We use this graph to find a *path pattern* for each error set that satisfies the error threshold. A path pattern is a regular expression that matches one or more execution paths. We will augment the usual regular expression notation with the following :

$a \longrightarrow b$  means a path from  $a$  to  $b$  that does not include  $b$

$\bar{x}$

$a \longrightarrow b$  means a path from  $a$  to  $b$  that does not include  $b$  or  $x$



For example, the path pattern

$$\overline{10}$$
$$3 \longrightarrow 8 \longrightarrow ( 5 \mid 15 )$$

would match the execution paths 1 2 3 5 6 8 3 6 8 5 and 2 3 8 9 15

but not 2 3 6 8 10 11 15 or 2 3 8 10 5 8 9 5.

Identification of input data that will drive a particular path is undecidable in general. Path patterns are thus of limited help in identifying input test data. However, they could be used to determine how many patterns have been matched by execution of a particular set of input data, and this could be automated. Only output produced by executing the desired paths would be subjected to the expensive task of output verification.

There are some improvements that can be made in the algorithm in Figure 2. It is possible to represent the CD flowgraph nodes as basic blocks rather than individual statements. This could avoid some of the time spent looking for control dominators and chasing arcs. Also, our algorithm considers each reference reached by data flow individually, when in fact one of them may dominate another. This results in producing overlapping path patterns at additional expense. We could also reduce the path patterns to a minimum form since they are regular expressions.

**Figure 2.**

In the following,  $ErrorSet_i$  denotes the error set after the execution of  $stmt_i$ . EFG is the error flow graph under construction. These algorithms assume that the CD flowgraph contains null-action end-of-predicate nodes.

**2(a).** Algorithm to propagate infection through assignment

procedure Propagate ( CDFlow, Current EFG node  $\langle c, ErrorSet_{Current} \rangle$  )

```
if  $c$  is an assignment  $v:=e$  then
  Refs := set of nodes in CDFlow reached by a data flow arc from  $c$ 
  for each node  $n$  in Refs do
    Create node New in EFG with an arc from Current
    Label the arc with  $m$  for any other node  $m$  that has a
      definition for  $v$  that reaches  $n$ 
    ModifyES ( CDFlow,  $v$ ,  $ErrorSet_{Current}$ ,  $n$ ,  $ErrorSet_{New}$  )
    New EFG node :=  $\langle n, ErrorSet_{New} \rangle$ 
    if New EFG node has previously appeared then
      Redirect its arc to previously appearing node
      Remove new EFG node
    else if size (  $ErrorSet_{New}$  ) < threshold then
      Propagate ( New EFG node )
else
  null
```

**2(b).** Algorithm to modify error set along a  $dr$  path

procedure ModifyES ( CDFlow, S, ESin, F, ESout )

```
DomList := all assignment stmts (excluding S) that dominate F
           in the control flow subgraph of CDFlow with start node S
           and final node F, ordered from most dominating to
           least dominating
ESout := ESin
for successive elements  $v:=e$  of DomList do
  if  $e$  contains at least one variable in ESout then
    ESout := ESout  $\cup$   $\{v\}$ 
  else
    ESout := ESout -  $\{v\}$ 
if F is an assignment  $v:=e$  then
  ESout := ESout  $\cup$   $\{v\}$ 
```

**2(c). Algorithm to construct an error flow graph**

**procedure Errorflow (i)**

**Select stmt<sub>i</sub> (node i in the CD Flowgraph) as the infection point**

**if stmt<sub>i</sub> is an assignment v:=e then**

**ErrorSet<sub>i</sub> := [v]**

**Root of EFG := <stmt<sub>i</sub>, ErrorSet<sub>i</sub>>**

**Propagate ( EFG root )**

**else if stmt<sub>i</sub> is a predicate then**

**stmt<sub>e</sub> := end-of-predicate node for stmt<sub>i</sub>**

**for each control arc leaving stmt<sub>i</sub> do**

**P := subgraph of CDFlow defined by all control paths that follow  
this control arc from stmt<sub>i</sub> to stmt<sub>e</sub>**

**for each node n in P that dominates stmt<sub>e</sub> do**

**if n is an assignment v:=e then**

**ErrorSet<sub>i</sub> := ErrorSet<sub>i</sub> ∪ [v]**

**exit if size(ErrorSet<sub>i</sub>) ≥ threshold**

**exit if size(ErrorSet<sub>i</sub>) ≥ threshold**

**root of EFG := <stmt<sub>i</sub>, ErrorSet<sub>i</sub>>**

**if size(ErrorSet<sub>i</sub>) < threshold then**

**for each v in ErrorSet<sub>i</sub> do**

**Find node n such that n defines v and there exist data flow  
arcs from n that reach beyond the scope of stmt<sub>i</sub>**

**A := all assignment stmt nodes outside the scope of stmt<sub>i</sub>  
reached by the definition in n**

**for each a in A do**

**Create node New in EFG with an arc from root of EFG**

**ModifyES ( CDFlow, stmt<sub>e</sub>, ErrorSet<sub>i</sub>, a, ErrorSet<sub>New</sub> )**

**New EFG node := <a, ErrorSet<sub>New</sub>>**

**if size(ErrorSet<sub>New</sub>) < threshold then**

**Propagate ( New EFG node )**

**else**

**null**

### 5.3 An example of threshold testing

The sorting program in Figure 3 is taken from [Las83]. The sorting algorithm is basically an ascending selection sort. The fault is a missing assignment to  $r3$  at statement 7, which should reset  $r3$  to point to the first element in the unsorted subarray. An error may occur if the inner **while** does not result in an assignment to  $r3$ , which happens if the first element of the unsorted subarray is already the smallest. The value of  $r3$  is that from a previous iteration and causes the (unsaved) array element it references to be overwritten. Note that cancellation may occur in two ways. First, if the unsorted subarray is the entire array, and the first element is already smallest, then  $r3$  has the proper value (0) due to the initialization in statement 2. Second, if the array elements are not unique, the error is cancelled if  $a[r1] = a[r3]$ , since  $a[r3]$  is overwritten with the same value.

Figure 4 shows the CD flowgraph for the program. The program contains no output statement, so we assume that the sorted array is output prior to **halt**. An array variable presents a particular problem for data flow analysis. Each element may act like an individual variable in terms of definition or reference. However, since subscripts may not be known until run-time, we cannot statically identify the elements involved. For example, an assignment to  $A[j]$  may or may not cancel a previous infection of  $A[i]$  depending on whether or not  $i=j$ . We will assume that the entire array is infected if one element becomes infected, and that the infection cannot be

cancelled by assignment to an array element. Figure 5 shows the error flow graph for an infection of  $r_0$  at statement 5 for an arbitrary threshold of 3 errors. The resulting path pattern tells us that we must avoid executing statement 10 before reaching statement 14. After that, any path will reveal the error. This is consistent with our intuition about the crucial way variable  $r_0$  is used in the program. Note that cancellation could occur if we did go through statement 10 first since this assignment to  $r_0$  would cancel the previous infection. This would require the predicate in statement 9 to take the correct branch even though it referenced an infected variable.

The actual fault in the program is a missing code fault, the missing assignment of  $r_3$  at statement 7. We first assume the presence of this statement as the infection point, and must compute the new data flow for this new statement. In this case, the arc in the CD flowgraph from 2 to 15 labeled  $r_3$  is deleted and replaced with an arc from 7 to 15 labeled  $r_3$ . Notice that there is now no reference to the value of  $r_3$  defined in statement 2, so statement 2 may be deleted (if included, it does no harm but is a dd anomaly). Figure 6 shows the resulting error flow graph and the path pattern generated. Inspection of the program will convince the reader that any path matching this path pattern will indeed reveal the error, except under the aforementioned cancellation circumstances.

Now consider a path along which the infection of  $r_3$  at statement 7 does not

reach the threshold (Figure 7). Since the infection is cancelled by a subsequent assignment before it can spread, this path will not reveal the error. This path does not appear in Figure 6 because our error flow graph construction algorithm specifically avoids this situation.

**Figure 3.** Sorting program

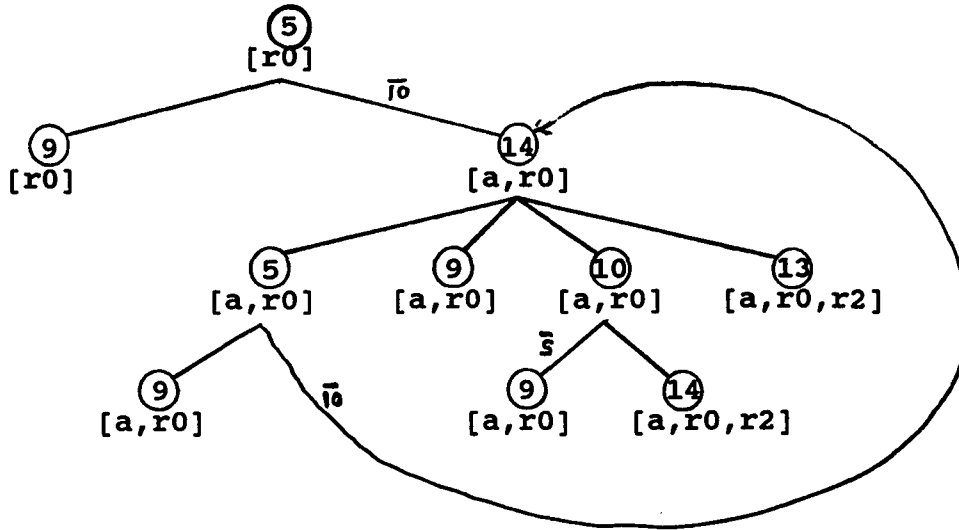
<u>stmt #</u>		
1	input (n,a)	
2	r3 := 0	
3	r1 := 0	
4	while r1 < n do	
5	r0 := a[r1]	
6	r2 := r1 + 1	
*7	{r3 := r1}	
8	while r2 <= n do	
9	if a[r2] < r0 then	
10	r0 := a[r2]	
11	r3 := r2	
	endif	
12	r2 := r2 + 1	
	endwhile	
13	r2 := a[r1]	
14	a[r1] := r0	
15	a[r3] := r2	
16	r1 := r1 + 1	
	endwhile	
	halt	

a - array[0..n]  
r0 - hold smallest element on this iteration  
r1 - marks start of unsorted subarray  
r2 - subscript of successive el.in unsorted subarray starting at r1+1  
r3 - points to smallest element found in unsorted subarray  
r2 - reused as temp for swapping

**Figure 4.** Matrix representation of CD flowgraph for Sorting program.  
 Each entry (x,y) represents a data flow arc from x to y for that variable.  
 A control flow arc is indicated by an asterisk (\*).

	1	2	3	4	5	6	8	9	10	11	12	13	14	15	16	17
1		*		n	a		n	a	a			a				
2			*											r3		
3				r1*	r1	r1						r1	r1		r1	
4					*											*
5						*		r0					r0			
6							r2*	r2	r2	r2	r2					
8								*				*				
9									*		*					
10								r0		*			r0			
11											*			r3		
12							r2*	r2	r2	r2	r2					
13													*	r2		
14					a			a	a			a		*		
15					a			a	a			a			*	
16				r1*	r1	r1						r1	r1		r1	
17																

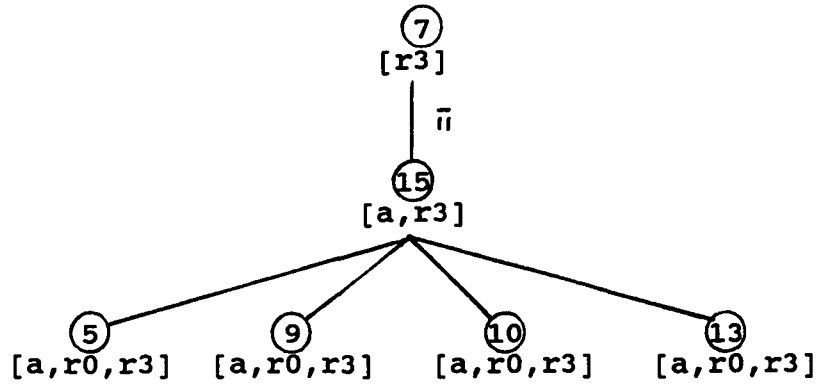
Figure 5. Error flow graph for infection of r0 at stmt 5



Path pattern :  $5 \xrightarrow{\overline{10}} 14 \longrightarrow ( 10 \longrightarrow 14 \mid 13 )$



**Figure 6.** Error flow graph for infection of r3 at stmt 7



Path pattern : 7  $\xrightarrow{ī}$  15  $\longrightarrow$  (5 | 9 | 10 | 13)

**Figure 7.** Path for infection at stmt 7 that does not reach threshold



## 5.4 Discussion of the threshold model

The threshold model has all of the advantages of error flow testing previously mentioned. Issues that need to be considered in further development of the model include :

### Arbitrary error threshold

The choice of the error threshold is a very subjective one. Choosing an error threshold that is too low does not allow us to do enough analysis to find the most likely error-revealing paths. By setting a high error threshold, the error flow analysis takes longer and may result in finding no paths that reach the threshold for some statements. The set of variables in most programs is partitioned into disjoint sets of variables which influence each other. If one of these sets is small, an infection of one of the variables might never spread beyond the threshold. In this case, we choose the path(s) that contain an error set of the greatest error degree encountered.

### Infeasible or infrequently executed path patterns

Since this method relies on static analysis, it may generate path patterns for infeasible or infrequently executed paths. Identifying infeasible paths through static analysis is undecidable, so this method suffers equally with other static methods. Lack of knowledge about execution frequencies of paths does mean testing time might be spent on infrequently executed paths, although some might argue that these

are the hiding places of the most insidious program bugs and that time testing them is well spent.

*No quantitative measure of confidence*

Although we can say that we have tested for an infection of every statement in the program, and for each statement we tested all paths that exceeded a certain threshold, this does not lead us to a quantitative measure of confidence.

*Overestimation of some error sets*

The algorithm used to construct error sets may overestimate the size of some error sets by including variables that are often coincidentally correct. When an overestimated error set exceeds the threshold, the path selected will be more likely than expected to produce correct output in the presence of a fault, thus we may misinterpret correct (error cancelled) output as an indication that the tested statement is correct when it is actually incorrect. We have tested predicate statements by assuming they always take the wrong branch. This is clearly not the case because of cancellation, but our representation of error sets does not let us record facts like "takes the wrong branch half the time".

*Handles cancellation implicitly rather than explicitly*

This model does not explicitly deal with cancellation problems, but rather relies on the "hope" that if enough different paths are executed enough times each

to show the correctness of a given statement, that an error, if present, would eventually slip through uncanceled. This is not an unreasonable approach for software that needs only be moderately reliable. However, for highly reliable software we would like to deal with the cancellation problem in a much stronger way.

Although the threshold model does not explicitly handle the issue of cancellation, we can envision other models that would. One model we are currently investigating is a probabilistic model of error flow testing that would allow us to make quantifiable statements about the probable correctness of a program. This model would represent error sets as fuzzy sets, where each element is associated with its probability of being a member of the set. Test paths would be chosen for those error sets with the highest overall probability of revealing an error.

## **6.0 Summary and future directions**

We have introduced error flow analysis as a new method of program analysis that investigates the semantic behavior of a program at the level of data state transitions. Error flow analysis addresses some of the deficiencies of past syntactically-based methods of program analysis. In particular, error flow analysis distinguishes not only various program paths, but also properties of those paths as they relate to program failure. The phenomena of resistance and cancellation along paths and their relationship to the fault/failure model have been described and investigated as central issues in error flow analysis, although these phenomena have been mostly ignored by past coverage-oriented methods. We have described two types of error flow analysis : dynamic error flow analysis and static error flow analysis. Dynamic error flow analysis is used to analyze the execution behavior of a program that contains injected faults as compared to the execution behavior of a correct program using the same input. Static error flow analysis attempts to approximate the actual error flow behavior of a program through an analysis of its code.

A formal model of error flow in computer programs has been developed based on mathematical functions and the fault/failure model. This model views

program statements as functions that map a prior data state into a subsequent data state, and it is used to examine the conditions that lead to infection and propagation of data state errors. We have defined and discussed syntactic closeness and semantic closeness as a means of comparing a faulty program with a correct program, and introduced error sets for tracking their data state differences. We also discussed coincidental correctness in the context of functions and described the difficulty of determining this behavior for functions in composition due to a lack of knowledge about their domains.

Experiments in dynamic error flow analysis were performed on several programs using a system constructed for this purpose : the DEFA system. This prototype system, which analyzes programs written in a subset of Pascal, compares the execution traces of syntactically close correct and faulty programs over many inputs, accumulating statistics on error flow behavior as well as execution, infection, and failure rates. The preliminary experiments suggest one statistic, the average maximum error set size, as a likely candidate for selecting paths with high failure rates. The performance of test data selected on this basis was compared with the performance of test data selected by the all-paths and all-du-paths criteria, and found to be superior in revealing selected errors by requiring fewer tests than the other strategies.

Static error flow analysis is based on traditional flow graph-theoretic

techniques, such as data flow analysis, augmented with semantic analysis to the extent possible, and is appealing because it may significantly reduce the cost of testing. Like other static methods, it suffers from problems such as the determination of infeasible paths and the possibility of coincidental correctness. The extent to which coincidental correctness properties can be estimated by static methods is an open question, and we are continuing our research on this subject. We have suggested one possible static testing method, called the threshold model of testing, which is based on the average maximum error set size statistic of the dynamic experiment. This model, which attempts to identify a promising set of test paths in a program, constructs error sets on the basis of data flow and makes no explicit attempt to handle coincidental correctness. We plan to experiment with this model and compare its performance against existing structural coverage testing methods. We also plan to investigate other models that incorporate any coincidental correctness properties that can be determined through static or static/dynamic analysis. The investigation of static error flow analysis is just beginning, and more experience and experimentation are necessary to find ways to overcome its inherent problems.

Although our primary emphasis in developing error flow analysis has been on the development of testing techniques, error flow analysis has other potential applications. We have demonstrated its potential use as a method of comparing the performance of existing testing techniques. We have also demonstrated its use as an investigative tool in understanding the subtleties in existing programs by making small

changes to a program and observing the effect. Other potential applications include its use as a measure of program complexity based on error flow characteristics rather than traditional syntactic measures, and its potential use as a debugging tool.

This research has suggested a new technique in program analysis and testing, with many possible directions for future research. In addition to those mentioned above, one topic to be immediately addressed is the enhancement of the DEFA system. This system is a significant product of our research, and a fully functional version might be useful to other researchers. It would also allow us to investigate a wider variety of programs. More experiments of the type performed in our research are necessary to gain a deeper understanding of the behavior of programs and errors. We expect our experience with the DEFA system to suggest characteristics of error flow behavior that will serve as a basis for a family of error flow testing strategies.



## APPENDIX

```

program DigitSeq;
var
  a, b, c : longint;           { holds 24-digit integer }
  d : longint;                 { done flag }
  s : longint;                 { flag : 1 - odd, 0 - even }
  p : longint;                 { current part of number : 0 - low, 1 - mid, 2 - high }
  x : longint;                 { holds current part of number }
  n : longint;                 { holds current digit }
  z : longint;                 { length of seq. of zeroes }
  L : longint;                 { length of longest sequence }
  w : longint;                 { length of current sequence }
  i : longint;

procedure next ( var p, x : longint );
begin
  traceoff;
  if p = 0 then begin          { get next 8-digit portion }
    x := c; p := p + 1 end
  else if p = 1 then begin
    x := b; p := p + 1 end
  else if p = 2 then begin
    x := a; p := p + 1 end;
  traceon
end;

procedure search ( var n, x, w, L, z : longint );
begin
  traceoff;
  for i := 1 to 8 do          { search current portion }
  begin
    n := x mod 10;
    x := x div 10;
    if (n mod 2) = s then     { start counting sequence }
      w := w + 1
    else
      begin
        if w > L then         { new longest sequence }
          L := w;
          w := 0
        end;

    if n = 0 then             { length of sequence of zeroes }
      z := z + 1
    else
      z := 0
    end;
  traceon
end;

procedure count ( var w, L : longint );
begin
  traceoff;
  if (n mod 2) = s then       { start counting sequence }
    w := w + 1
  else
    begin
      if w > L then           { new longest sequence }
        L := w;
        w := 0
      end;
  traceon
end;

```

```

procedure zero ( var z : longint );
begin
  traceoff;
  if n = 0 then
    z := z + 1
  else
    z := 0;
  traceon
end;

```

Version 1 -----

```

begin { main }
d := 0; p := 0; w := 0; L := 0; z := 0;
read (a); read (b); read (c);

if (c mod 2) = 0 then
  s := 0
else
  s := 1;

while d <> 1 do
begin
  next (p, x);
  search (n, x, w, L, z);

  if p > 2 then
    d := 1
  end;

if (s = 0) and (w > L) and (n = 0) then
  w := w - z;
if w > L then
  L := w;
writeln (L)
end.

```

Version 2 -----

```

begin { main }
d := 0; p := 0; w := 0; L := 0; z := 0;
read (a); read (b); read (c);

if (c mod 2) = 0 then
  s := 0
else
  s := 1;

while d <> 1 do
begin
  next (p, x);
  for i := 1 to 8 do
begin
  n := x mod 10;
  x := x div 10;

  count (w, L);

  if n = 0 then
    z := z + 1
  else
    z := 0
end;
end;

```

```

        if p > 2 then                                { done }
            d := 1
        end;

        if (s = 0) and (w > L) and (n = 0) then      { adjust for leading 0's }
            w := w - z;
            if w > L then                              { last sequence was longest }
                L := w;
            writeln (L)
        end.

```

Version 3 -----

```

begin { main }
    d := 0; p := 0; w := 0; L := 0; z := 0;
    read (a); read (b); read (c);

    if (c mod 2) = 0 then                            { determine even or odd }
        s := 0
    else
        s := 1;

    while d <> 1 do
        begin
            next (p, x);

            for i := 1 to 8 do                        { search current portion }
                begin
                    n := x mod 10;
                    x := x div 10;

                    if (n mod 2) = s then            { start counting sequence }
                        w := w + 1
                    else
                        begin
                            if w > L then          { new longest sequence }
                                L := w;
                                w := 0
                            end;

                            zero (z);
                        end;

                    if p > 2 then                    { done }
                        d := 1
                    end;

                    if (s = 0) and (w > L) and (n = 0) then { adjust for leading 0's }
                        w := w - z;
                    if w > L then                    { last sequence was longest }
                        L := w;
                    writeln (L)
                end.
end.

```

Version 4 -----

```

begin { main }
    d := 0; p := 0; w := 0; L := 0; z := 0;
    read (a); read (b); read (c);

    if (c mod 2) = 0 then                            { determine even or odd }
        s := 0
    else
        s := 1;

```

```

while d <> 1 do
  begin
    next (p, x);

    for i := 1 to 8 do
      begin
        n := x mod 10;
        x := x div 10;

        count (w, L);
        zero (z)
      end;

      if p > 2 then
        d := 1
      end;

      if (s = 0) and (w > L) and (n = 0) then
        w := w - z;
      if w > L then
        L := w;
      writeln (L)
    end.

```

Version 5 -----

```

begin { main }
  d := 0; p := 0; w := 0; L := 0; z := 0;
  read (a); read (b); read (c);

  if (c mod 2) = 0 then
    s := 0
  else
    s := 1;

  while d <> 1 do
    begin
      if p = 0 then begin
        x := c; p := p + 1 end
      else if p = 1 then begin
        x := b; p := p + 1 end
      else if p = 2 then begin
        x := a; p := p + 1 end;

      for i := 1 to 8 do
        begin
          n := x mod 10;
          x := x div 10;

          count (w, L);
          zero (z)
        end;

        if p > 2 then
          d := 1
        end;

        if (s = 0) and (w > L) and (n = 0) then
          w := w - z;
        if w > L then
          L := w;
        writeln (L)
      end.

```

## BIBLIOGRAPHY

- [Cla89] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria", *IEEE Transactions on Software Engineering* (November 1989), pp. 1318-1332.
- [DeM78] R. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer* (April 1978), pp. 34-41.
- [Dur84] J. W. Duran and S. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering* (July 1984), pp. 179-183.
- [Fos76] L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability", *ACM Computing Surveys* (September 1976), pp. 305-330.
- [Goo75] John B. Goodenough and Susan Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering* (June 1975), pp. 156-173.
- [Hec77] Matthew S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland (1977).
- [Hoa69] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM* (October 1969), pp. 576-585.
- [How85] W. E. Howden, "The Theory and Practice of Functional Testing", *IEEE Software* (September 1985), pp. 6-18.
- [How86] W. E. Howden, "A Functional Approach to Program Testing and Analysis", *IEEE Transactions on Software Engineering* (October 1986), pp. 997-1005.
- [Hua75] J. C. Huang, "An Approach to Program Testing", *ACM Computing Surveys* (Sept. 1975), pp. 113-128.
- [Joh91] Robert Johnson, Sharon Navard, James Davenport. Division of Statistics, Department of Mathematical Sciences, Virginia Commonwealth University. Personal communication.
- [Kor87] Bogdan Korel, "The Program Dependence Graph in Static Program Testing", *Information Processing Letters* (January 1987), pp. 103-108.

- [Las83] J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy", *IEEE Transactions on Software Engineering* (May 1983), pp. 347-354.
- [Mil74] E. Miller, M. Paige, J. Benson, and W. Wisehart, "Structural Techniques of Program Validation", *Digest of Papers COMPCON 74* (Spring 1974), pp. 161-164.
- [Mil90] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas, "Estimating the Probability of Failure When Testing Reveals No Failures", *IEEE Transactions on Software Engineering* (to appear).
- [Mor81] L. J. Morell and R. G. Hamlet, *Error Propagation and Elimination in Computer Programs*, University of Maryland TR-1065, Department of Computer Science (July, 1981).
- [Mor84] L. J. Morell, *A Theory of Error-Based Testing*, University of Maryland TR-1395, Department of Computer Science (August, 1984). PhD Thesis.
- [Mor87] L. J. Morell, "A Model for Code-Based Testing Schemes", *Proceedings of the Fifth Annual Pacific Northwest Software Quality Conference*, (October 1987), pp. 309-326.
- [Mor88] L. J. Morell, "Theoretical Insights into Fault-Based Testing", *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis (TAV2)* (July 19-21, 1988), pp. 45-62.
- [Mor90] L. J. Morell, "A Theory of Fault-Based Testing", *IEEE Transactions on Software Engineering* (August 1990), pp. 844-857.
- [Nta84] S. Ntafos, "On Required Element Testing", *IEEE Transactions on Software Engineering* (November 1984), pp. 795-803.
- [Off89] A. Jefferson Offutt, "The Coupling Effect : Fact or Fiction", *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification (TAV 3)* (Dec. 13-15, 1989), pp. 131-140.
- [Pod89] A. Podgurski and L. A. Clarke, "The Implications of Program Dependences for Software Testing, Debugging, and Maintenance", *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification (TAV 3)* (Dec. 13-15, 1989), pp. 168-178.

- [Pod90] A. Podgurski and L. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Transactions on Software Engineering* (September 1990), pp. 965-979.
- [Rap85] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering* (April 1985), pp. 367-375.
- [Voa90] J. Voas, *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs*, College of William and Mary, Department of Computer Science (May 1990). PhD Thesis.
- [Wey80] Elaine J. Weyuker and Thomas J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains", *IEEE Transactions on Software Engineering* (May 1980), pp. 236-246.
- [Wey84] E. J. Weyuker, "The Complexity of Data Flow Criteria for Test Data Selection", *Information Processing Letters* (August 1984), pp. 103-109.
- [Whi80] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing", *IEEE Transactions on Software Engineering* (May 1980), pp. 247-257.
- [Woo88] M. R. Woodward and K. Halewood, "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues", *Second Workshop on Software Testing, Verification, and Analysis (TAV2)* (July 1988), Banff, Alberta, Canada, pp. 152-158.
- [Zei88] Steven Zeil, "Selectivity of Data-Flow and Control-Flow Path Criteria", *Second Workshop on Software Testing, Verification, and Analysis (TAV2)* (July 1988), Banff, Alberta, Canada, pp. 216-222.

## VITA

### Branson (Buz) Wayne Murrill

Born in Richmond, Virginia, February 20, 1952. Graduated from the College of William and Mary in June, 1974 with a B.S. in Mathematics/Computer Science. Earned a Master of Computer Science degree from the University of Virginia in December, 1977.

Dr. Murrill has held several positions in industry and government as a computer professional, and is currently an Assistant Professor of Computer Science in the Department of Mathematical Sciences at Virginia Commonwealth University, where he has worked for the past 7 years.

He lives in Mechanicsville, Va. with his wife, Linda, and their daughters Jenny, Kathy, and Sara.