(2)

RL-TR-92-181
In-House Report
September 1992

HIGH LEVEL DESIGN FOR DISTRIBUTED APPLICATION INSTRUMENTATION

Vaughn T. Combs, Cheryl L. Blake, 1/Lt, USAF

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, New York

RL-TR-92-181 has been reviewed and is approved for publication.

APPROVED: [illegible]

ANTHONY F. SNYDER, Chief
C2 Systems Division

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1992	3. REPORT TYPE AND DATES COVERED In-House

4. TITLE AND SUBTITLE HIGH LEVEL DESIGN FOR DISTRIBUTED APPLICATION INSTRUMENTATION	5. FUNDING NUMBERS PE - 62702F PR - 5581 TA - 28 WU - 17
6. AUTHOR(S) Vaughn T. Combs, Cheryl L. Blake, 1/Lt, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Road Griffiss AFB NY 13441-4505	8. PERFORMING ORGANIZATION REPORT NUMBER RL-TR-92-181
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Road Griffiss AFB NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Cheryl Blake/C3AB (315) 330-2158
1/Lt, USAF

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper describes the high level design for a distributed application instrumentation package. The instrumentation package provides monitoring tools to aid application designers in developing and understanding the behavior of their applications within an object-oriented distributed environment. The package consists of a general query-based system for the collection of events that occur within an application and a display system for processing and presenting the events. These events are collected through the use of probes which register events with a distributed database. Calls to these probes are embedded within the code to be instrumented in order to mark the occurrence of a specific event. The instrumentation package will provide a library of pre-defined probes based on events that adhere to the object/thread model (e.g. thread entrances into objects, thread exits from objects, etc.) and will also allow for user-defined probes. The display to be provided will represent object/thread interactions. The design is built flexibly so as to permit a wide range of events and displays to be used with the package.

14. SUBJECT TERMS Distributed Computing, Software Instrumentation, Heterogeneous Computing	15. NUMBER OF PAGES 60
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

In the past decade, many specialized software testing philosophies and testing techniques have evolved. Some of these testing philosophies are time-consuming and impractical while others are practical but only for small programs. Automating the testing process is a goal that needs to be attained for testing to be cost-effective and practical for any size or type of software program. Two testing techniques have been explored under Rome Laboratory (C3CB) R&D programs. These programs developed automated testing tools which support these techniques. One testing technique is mutation analysis and the other technique is decision-to-decision path analysis. A comparison of these testing techniques via two test tools was performed. This technical report describes the testing process and the results of this comparison.

2.0 BACKGROUND

Testing is that phase in the software life cycle where a program is symbolically or physically executed with the intent of gaining confidence in its correctness.[1] The basic problem, therefore, is finding a test selection criterion, a rule to select sets of test cases that will constitute a reliable test. Early in the 1970's, as part of software testing's theoretical foundations, J. B. Goodenough and S. L. Gerhart defined the concept of a reliable test which they believed was sufficient for verifying <u>any</u> program's correctness. As it turned out their ambitious theory was not practical in the real world.

Real world software testers began implementing test strategies which work on particular classes of programs or particular classes of errors. There are two classes of testing strategies: non-traditional and traditional. The most widely used testing strategies are traditional (also called manual) test strategies which include deskchecking, code walkthroughs, and inspections. Non-traditional test strategies are more extensive. TABLE 1 shows the non-traditional strategies in a matrix format. Non-traditional strategies include structure dependent and structure independent test strategies, both of which may be categorized as either deterministic or random strategies.

---

[1] A. Goel, Syracuse University Publication, September 1987.

Structure dependent testing is based on the structural properties of the program code. Structure independent testing is based on the specifications (requirements) of the program, and is not concerned with the design or code structure. Deterministic testing is performed by taking into account the structure and/or specifications of the program during test case selection. Random testing assumes that all test input is created equal. It assumes that any input is as good, for testing purposes, as any other input.

	Structure Dependent (White Box)	**Structure Independent (Black Box)**
Deterministic	Statement Testing Branch Testing Path Testing Structured Testing Symbolic Testing Domain Testing Mutation Testing	Equivalence Partitioning Boundary Value Analysis Cause-Effect Graphing Design-Based Functional Testing
Random	Randomized Partition Testing	SIAD Tree Testing

NON-TRADITIONAL TESTING STRATEGIES

TABLE 1

Since it is virtually impossible to test a program with all possible inputs to see if it produces the correct outputs, current research efforts have concentrated on testing strategies which are more practical than such brute force methods. Test strategies must be reasonable in effort, in order to be cost effective. For these reasons, testing strategies which select only a small subset of the entire possible input domain are being pursued. While these testing techniques cannot guarantee program correctness, they provide the tester with a higher level of confidence in the program. Thus, as stated by E. W. Dijkstra and generally accepted as a maxim, "testing can only be used to detect the presence of errors, never their absence."

The two testing strategies examined here are structure dependent strategies.

3.0 TESTING STRATEGIES

3.1 MUTATION TESTING

Mutation testing is based on the "competent programmer hypothesis", it assumes the program under test has been written by a skilled (i.e., competent) programmer.[1] It then follows that the program would be almost correct, and differ from a truly correct program by only a few small errors. Mutation testing allows a tester to gauge whether a set of test data is adequate to detect those errors. This strategy makes a series of minor changes to a program being tested, creating a series of programs known as mutant programs. Each program is the same as the original program except for a single syntactic change. This minor change can be in the form of constant replacement, arithmetic, relational, logical or logical operator replacement, statement deletion, and statement addition (i. e., Return, Continue, or a Trap statement).[2]

The process consists of determining the expected output for each test case of the original program, generating a set of mutant programs, determining the mutant output for each test case, and comparing the mutant output with the expected output for each test case and mutant. If the mutant's output is different from the expected output for a test case, then that mutant is said to be discovered and "killed" by that test case and the mutant is said to be "dead." Otherwise, it remains "alive" but may still be killed by a subsequent test case. Live mutants provide important test information. A mutant may remain alive for one of three reasons:

[1] R. A. DeMillo, et al., Purdue University/University of Florida, Software Engineering Research Center, "An Overview of the Mothra Software Testing Environment," SERC-TR-3-P.

[2] R. A. DeMillo, et al., Purdue University/University of Florida, Software Engineering Research Center, "The Mothra Software Testing Environment," User's Manual, SERC-TR-4-P.

a. The test da᷊ is inadequate. This means that the test data may not have covered (i.e., exercised) that portion of the program.

Example: In the test program (ref, page 14) the input test data "(3,3,5)" and "(5,3,3)" each exercise different parts of the program. If one is not included in the test data, part of the program will not be exercised.

b. The mutant program is equivalent to the original program. If the mutant program and the original program always produce the same output, there is no way for the test data to distinguish between the original program and the mutant program. It may mean that the programs are essentially the same.

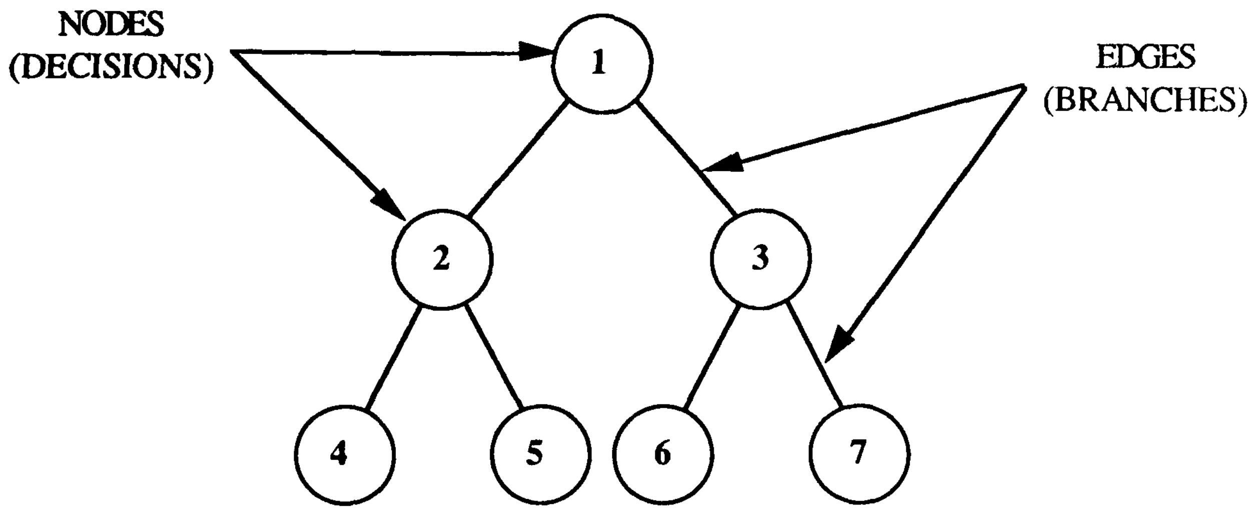
Example: Consider a program that first checks all input variables to make sure they are greater than zero, and exits if the inputs are negative. One mutant type is the "absolute value insertion" (ref, Table 2). This mutant operator replaces each variable in a program by the absolute value of the variable. However, since the original program is designed such that that particular variable is always positive, this mutant will always produce the same output as the original program, and is therefore equivalent.

c. There exists an error in the program. If the . ɔut of the original program and the mutant program is the same and the test data has exercised that portion of code, and the mutant program is not equivalent to the original program, then an error is uncovered.

Example: See Figures 10, 11, 14, and 17 for mutants remaining which remain after test case execution and are not equivalent to the original program.

3.2 DECISION TO DECISION PATH (DD-PATH) TESTING - BRANCH TESTING

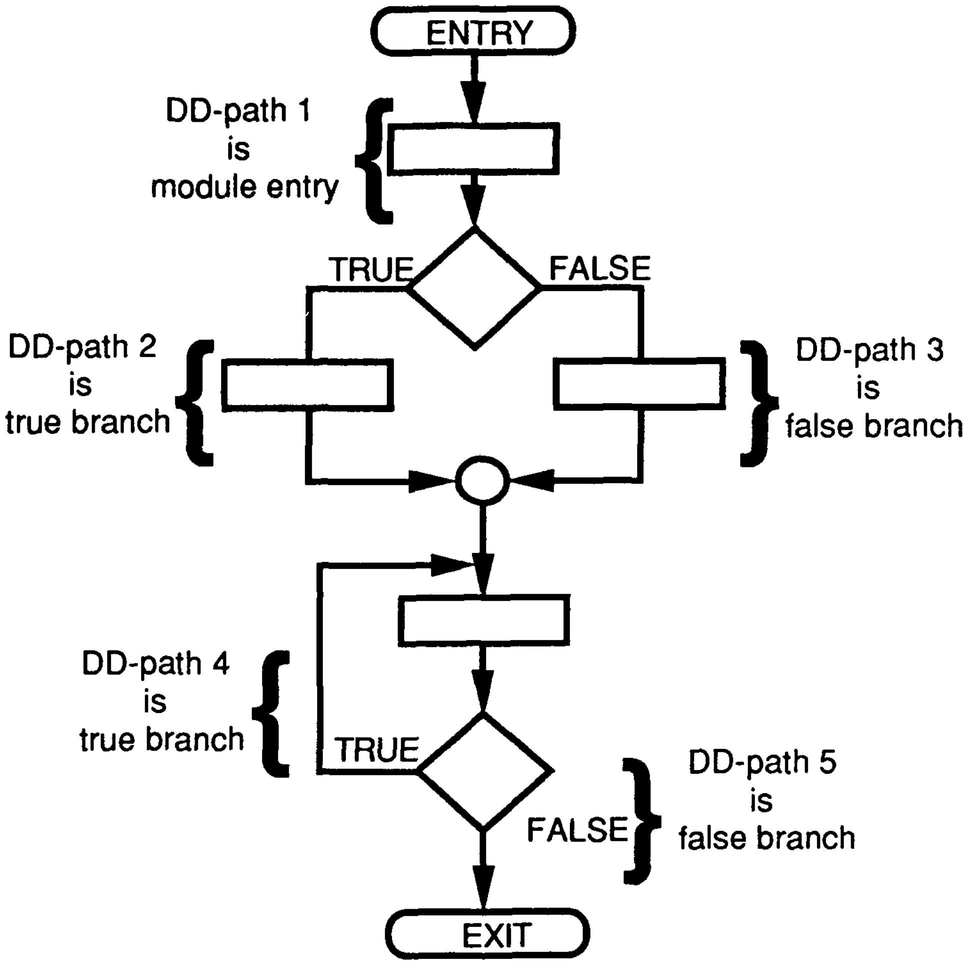
DD-PATH analysis uses a very simple structure dependent test criteria strategy; i.e., cover all the edges (branches) of the program's directed flow graph (ref FIGURE 1). This insures that not only every branch of the program will be executed at least once, but that every statement will also be executed. DD-PATH testing is frequently called Branch testing. DD-PATH testing insures the execution of every loop and control flow at least once. It involves inserting statements or routines into the program to be tested to record properties of the executing program. It does not affect the functional behavior of the program. A decision point is either the entry point of a module, or a place where more than one possible path, or decision occurs. The path that is followed from one decision point to the next is called a decision-to-decision path[1] (ref FIGURE 2).



DIRECTED FLOW GRAPH
FIGURE 1

Path testing is a more stringent test criterion than DD-PATH testing. Path testing covers all the edge-to-edge transitions in the directed flow graph. This means that path testing covers all possible ways to traverse from the start of the program to its ending statement. Most times it is impossible to test all the paths or combinations of branches in a large program. Programs with many loops may have an infinite number of paths. However, it is possible to test all DD-PATHs. Analytic information consists of a listing of the DD-PATHs of the program undergoing analysis and the number of times each DD-PATH is executed when the program is executed. For more effective testing, all DD-PATHs and as many paths or branch combinations as possible should be tested. The goal is to increase the amount of code tested.

[1] General Research Corporation, "RXVP80, The Verification and Validation System for Fortran," User's Manual, 1985, 5-9.


ENTRY
DD-path 1 is module entry
TRUE
FALSE
DD-path 2 is true branch
DD-path 3 is false branch
DD-path 4 is true branch
TRUE
DD-path 5 is false branch
FALSE
EXIT

DECISION-TO-DECISION PATH EXAMPLE

FIGURE 2

4.0 AUTOMATED TESTING TOOLS

Manually testing software can be a very tiresome, time consuming, costly, and error-prone task. Since a great deal of manhours are traditionally expended in program testing, automated testing tools are gaining acceptance, especially in space and military applications. According to E. Miller in "Structurally Based Automatic Program Testing,"[1] most application programs written in FORTRAN can be tested minimally thorough with a relatively small number of test cases. A test is minimally thorough if each and every branch in

[1] Miller, E. F., et al., "Structurally Based Automatic Program Testing," EASCON-74, Washington D.C., October 1974.

its directed flow graph is traversed at least once during the test.[1] Two automated testing systems, which can be used to provide minimally thorough testing, have been developed under sponsorship of Rome Laboratory and are described in the following sections.

4.1 MOTHRA

MOTHRA is a mutation-based testing system that allows a tester to perform mutation analysis on a program (ref FIGURE 3). The tester chooses the classes and types of mutations (ref TABLE 2) to be performed and the test strength desired (i. e., percentage of selected mutants that will be enabled). The tester also supplies the test cases to be used as test input. The system executes the test data on the original program and the mutant program and compares the output. If the output resulting from the mutant program is different from the original output then the mutant is considered dead.[2] If the outputs are the same, the mutant is not detected and is considered alive. The objective is to kill all of the mutants.

MOTHRA supports three super-classes of mutation analyses:

a. Statement analysis. Mutated statement and control structures are introduced into the program code. These mutants test for traditional statement analysis, testing that all statements are executed and that each statement has an effect.

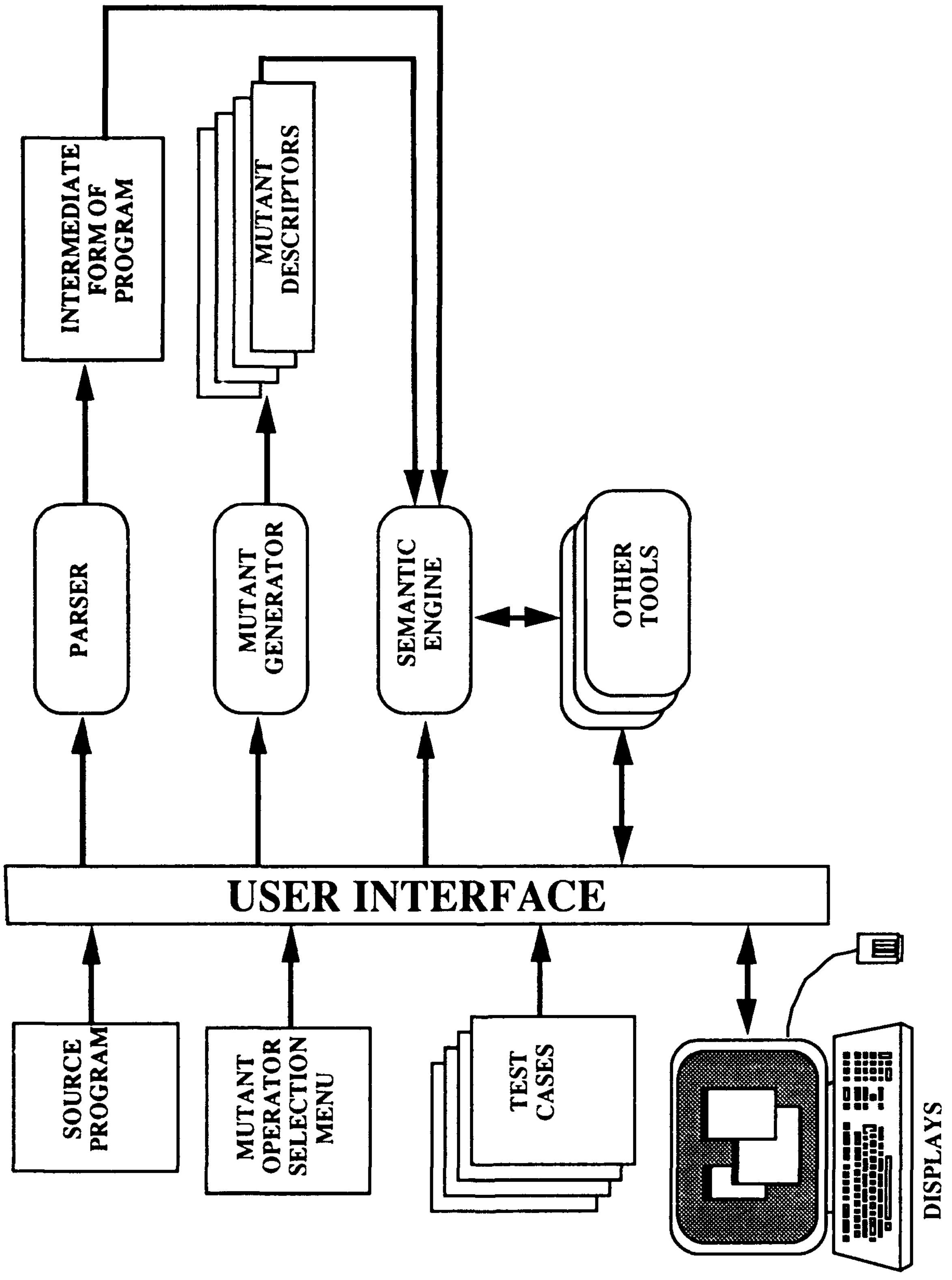
b. Predicate and domain analysis. Mutated expressions, arithmetic operators, and constants are introduced into the program code. These mutants test predicate boundaries and data domains.

c. Coincidental correctness. Scalar variables, array references, and constants are replaced with other values. These mutated programs detect errors that are undetected by other testing strategies when, due to the nature of the test data, the program just happens to (coincidentally) produce the correct output results.

---

[1] Huang, J. C., "An Approach to Program Testing," ACM Computing Surveys, September 1975, 113-128.

[2] R. A. DeMillo, et al., Purdue University/University of Florida, Software Engineering Research Center, "An Overview of the Mothra Software Testing Environment," SERC-TR-3-P, 1-3.



THE MUTATION PROCESS

FIGURE 3

Abbreviations for MUTANT TYPES are:	
aar	array reference for array reference replacement
abs	absolute value insertion
acr	array reference for constant replacement
aor	arithmetic operator replacement
asr	array reference for scalar variable replacement
car	constant for array reference replacement
cnr	comparable array name replacement
crp	constant replacement
csr	constant for scalar replacement
der	DO statement end replacement
dsa	data statement alterations
glr	goto label replacement
lcr	logical connector replacement
ror	relational operator replacement
rsr	return statement replacement
san	statement analysis (replacement by TRAP)
sar	scalar variable for array reference replacement
scr	scalar for constant replacement
sdl	statement deletion
src	source constant replacement
svr	scalar variable replacement
uoi	unary operator insertion

Abbreviations for MUTANT CLASSES:	
ary	array mutations (aar,car,cnr,sar)
con	constant-related mutations (acr,scr,src)
ctl	control structure mutants (glr,der,rsr)
dmn	domain perturbations (abs,crp,dsa,uoi)
opm	operator mutants (aor)
prd	operand mutants (lcr,ror)
scl	scalar mutants (asr,csr,svr)
stm	statement mutants (san,sdl)

Abbreviations for MUTANT SUPER CLASSES:	
all	all the mutants
cca	coincidental correctness analysis (ary,scl,opm,con)
pda	predicate and domain analysis (dmn,prd)
sal	statement analysis (stm, ctl)

MOTHRA MUTANT OPERATORS

TABLE 2

MOTHRA has evolved from previous work in mutation systems. The first was PIMS in 1979, a FORTRAN subset prototype, EXPER an experimental vehicle in 1980, CMS.1 a COBOL system in 1981 and FMS.3 an enhancement of EXPER in 1983. MOTHRA is designed to allow the testing of software at all test stages in the development process. It can accommodate units ranging from 10 to 100,000,000 lines of code. It currently supports FORTRAN 77, and follow-on work is planned to support the Ada programming language. An attractive feature of mutation analysis is that it includes statement and branch coverage, as it performs mutation analysis. In addition, the mutation score of a particular program (i. e., dead mutants/total # of mutants) indicates the adequacy of the data used to test the program, and is also a potential predictor of operational reliability. A potential problem of mutation analysis systems is the amount of disk storage and manpower required for the testing of the programs. However, MOTHRA allows the user to choose a subset of mutants that is very manageable and still adequate for testing purposes.

MOTHRA was developed by Georgia Institute of Technology (with a subcontract to Purdue University), under the sponsorship of Rome Laboratory contract F30602-85-C-0255. MOTHRA currently runs under 4.3BSD UNIX[1], System V UNIX, and Ultrix-32 V3.0[2].

4.2 RXVP80[3]

Research EXportable Verification Program for the 80's (RXVP80) is a software testing tool used to test and verify FORTRAN programs. RXVP80 can analyze FORTRAN 66, FORTRAN 77 and most FORTRAN extensions to the standards.

RXVP80 performs static as well as dynamic analysis of programs. Static analyses are those which do not require execution of the user's program, but which collect information on the structure of the program. Static analyses provides information on control structure, symbol usage, calling hierarchy, as well as unreachable code.

Dynamic analyses require execution of the user's program and provides run-time execution coverage information. As part of execution coverage analysis (ref FIGURE 4), the user's source code is

[1] UNIX is a trademark of Bell Laboratories.

[2] Ultrix and Ultrix-32 are trademarks of Digital Equipment Corporation.

[3] RXVP80 is a trademark of General Research Corporation, Santa Barbara, CA.

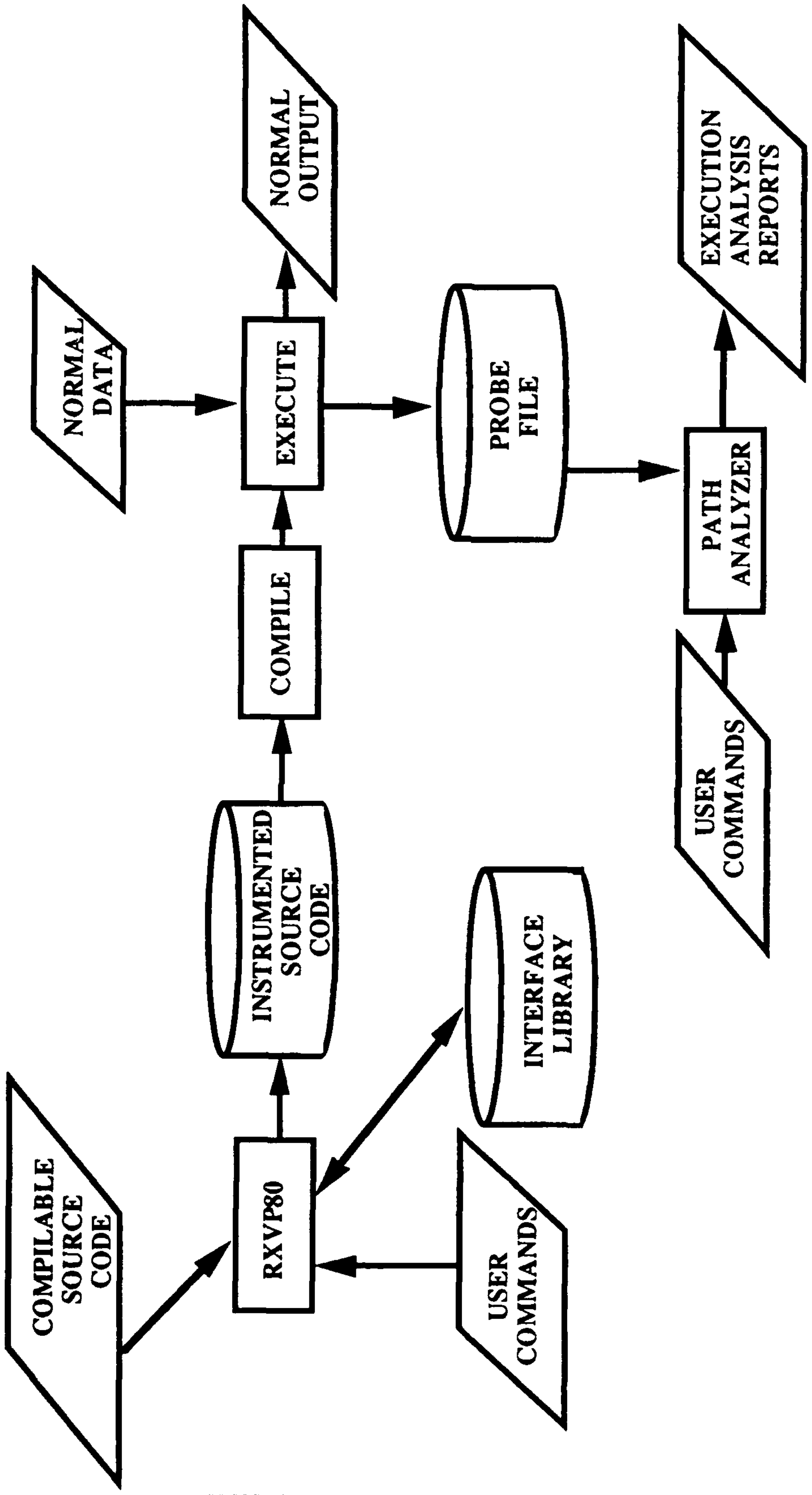
instrumented (i.e., software probes are inserted) with statements that trace the execution of the program. The execution of this instrumented program produces a set of data that trace the DD-PATHs and/or statements executed during the test run. A number of reports that show the extent of program testing is then produced from the data. The information provided indicates the thoroughness of the tests, including which DD-PATHs are taken, which DD-PATHs are not taken, and how often each DD-PATH is traversed. RXVP80 provides the capability to test for 100% branch and statement coverage of a program.

The dynamic analysis portion of the RXVP80 was used in analysis of the test program and its performance was matched against both MOTHRA's statement and predicate & domain analyses capabilities as these strategies were similar in their proposed detection of errors.

RXVP80 is a commercial product from General Research Corporation (Santa Barbara, CA) that resulted from a Rome Laboratory (C3CB) effort entitled "FORTRAN Automated Verification System" (FAVS), contract F30602-76-C-0436.

5.0 THE TEST PROGRAM

The test program used for automated testing by MOTHRA and RXVP80 determines the type of triangle (scalene, isosceles, or equilateral) from the data that is entered. The program user must provide the length of the three sides of the triangle as integer inputs. The program checks for negative integers or zero in the input and, if found, it determines the input to be "not a valid triangle." A triangle program was chosen because of its popularity in the software testing literature. The triangle classification program in "Theories of Program Testing & the Application of Revealing Subdomains" by Weyuker and Ostrand is widely used to validate software testing techniques. The test program used in this experiment is somewhat different. In this program, the input does not have to be in ascending or descending order. For example, the input can be entered as, (3,3,5), (5,3,3), or (3,5,3). Because this program allows any order, to test this program you must test all combinations of input! Even if the tester does not immediately see this at first glance, by using these tools it becomes apparent that, for example, (3,3,5) does not exercise the same statements as (5,3,3). Thus, test



EXECUTION COVERAGE ANALYSIS
FIGURE 4

cases must be built from information learned after exercising the tools and examining the results.

The triangle subroutine (ref FIGURE 5) is a correct program -- it contains no errors. Therefore, this program was input to both test tools and the output was examined. This was the "control program." Next, three types of errors were introduced into the program: a domain error, a computation error, and missing statement error. Then each test tool was used on each of the control program's variants (i.e., errors). The types of errors introduced into the variants and the results of testing them are discussed in the following sections.

6.0 CORRECT PROGRAM

6.1 MOTHRA - STATEMENT ANALYSIS MUTANTS

Using MOTHRA the first class of mutants selected was the STATEMENT ANALYSIS class, which includes statement mutants (statement replaced by TRAP, and statement deletion) and control structure mutants (GOTO label replacement, DO statement end replacement, and return statement replacement). All mutants that belong to the statement analysis class (i. e., the test strength was 100) were enabled. Using 35 test cases (ref FIGURE 6) on the correct program, all mutants were killed except one. Examination of this mutant program revealed that it was essentially the same as the original program and, therefore, the mutant program was "equivalenced" (ref TABLE 3). To equivalence a mutant program means to declare it to be functionally the same as the original program. For the triangle program, MOTHRA's replacement of "GOTO 110" with the RETURN statement did not change the program since, at that point in the program, the variable "MATCH" already had the correct return value, making the mutant equivalent to the original program. Thus, all mutants were accounted for and, as expected, the presence of any errors was not detected.

6.2 MOTHRA - PREDICATE & DOMAIN MUTANTS

Using MOTHRA, the second class of mutants selected was the PREDICATE AND DOMAIN class, which includes domain perturbations (absolute value insertion, constant replacement, data statement alterations, and unary operator insertion) and operand mutants (logical connector replacement, and relational operator replacement).

```
      SUBROUTINE TRIANGLE(I,J,K,MATCH)

      integer  i,j,k,match

C     MATCH is output from the subroutine:
C     MATCH = 1 IF THE TRIANGLE IS SCALENE
C     MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C     MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C     MATCH = 4 IF NOT A TRIANGLE

C     After a quick confirmation that it's a legal
C     triangle, detect any sides of equal length
      IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
      MATCH=0
      IF (I.NE.J) GOTO 10
      MATCH=MATCH+1
10    IF (I.NE.K) GOTO 20
      MATCH=MATCH+2
20    IF (J.NE.K) GOTO 30
      MATCH=MATCH+3
30    IF (MATCH.NE.0) GOTO 100

C     Confirm it's a legal triangle before declaring it to be scalene
      IF (I+J.LE.K)  GOTO 500
      IF (J+K.LE.I)  GOTO 500
      IF (I+K.LE.J)  GOTO 500
      MATCH=1
      Return

C     Confirm it's a legal triangle before declaring
C     it to be isosceles or equilateral
100   IF (MATCH.NE.1) GOTO 200
      IF (I+J.LE.K) GOTO 500
110   MATCH=2
      RETURN
200   IF (MATCH.NE.2) GOTO 300
      IF (I+K.LE.J)  GOTO 500
      GOTO 110
300   IF (MATCH.NE.3) GOTO 400
      IF (J+K.LE.I)  GOTO 500
      GOTO 110
400   MATCH=3
      RETURN

C     Can't fool this program, that's not a triangle
500   MATCH=4
      RETURN
      END
```

CORRECT TRIANGLE PROGRAM

FIGURE 5

Test Cases for triangle.tc.

Values for case 1.	Values for case 2.
I 3	I 2
J 4	J 2
K 5	K 2
Values for case 3.	Values for case 4.
I 2	I 5
J 2	J 4
K 1	K 3
Values for case 5.	Values for case 6.
I 0	I -1
J 0	J 0
K 0	K 3
Values for case 7.	Values for case 8.
I 4	I 2
J 3	J 3
K 2	K 4
Values for case 9.	Values for case 10.
I 2	I 6
J 4	J 8
K 2	K 10
Values for case 11.	Values for case 12.
I -1	I -1
J -1	J 2
K -1	K -1
Values for case 13.	Values for case 14.
I 2	I 2
J 1	J 0
K 2	K 2

INITIAL SET OF TEST CASES

FIGURE 6

Values for case 15.
I 2
J 2
K -1

Values for case 16.
I 1
J 2
K 2

Values for case 17.
I 1
J 1
K 1

Values for case 18.
I 0
J 1
K 2

Values for case 19.
I 1
J 2
K 0

Values for case 20.
I 1
J -2
K 5

Values for case 21.
I 3
J 3
K 2

Values for case 22.
I 1
J 5
K 5

Values for case 23.
I 7
J 5
K 5

Values for case 24.
I 10
J 5
K 5

Values for case 25.
I 5
J 5
K 10

Values for case 26.
I 5
J 10
K 5

Values for case 27.
I 0
J 0
K 2

Values for case 28.
I 2
J 0
K 0

INITIAL SET OF TEST CASES

FIGURE 6 (continued)

Values for case 29.
I 1
J 1
K 2

Values for case 30.
I 3
J 3
K 3

Values for case 31.
I 0
J 3
K 7

Values for case 32.
I 1
J 5
K 9

Values for case 33.
I -1
J 5
K 1

Values for case 34.
I 1
J 0
K 5

Values for case 35.
I -20
J -20
K -20

INITIAL SET OF TEST CASES

FIGURE 6 (continued)

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
glr	126	0	0.0	0	128
rsr	38	0	0.0	1	37
san	36	0	0.0	0	36
sdl	41	0	0.0	0	41
TOTALS	243	0	0.0	1	242

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	166	0	0.0	1	165
dmn	0	0	0.0	0	0
opm	0	0	0.0	0	0
prd	0	0	0.0	0	0
scl	0	0	0.0	0	0
stm	77	0	0.0	0	77

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	243	0	0.0	1	242
cca	0	0	0.0	0	0
pda	0	0	0.0	0	0
sal	243	0	0.0	1	242

MOTHRA - STATEMENT ANALYSIS MUTANTS (CORRECT PROGRAM)

TABLE 3

Using the same 35 test cases as in the statement analysis mutants test run described in section 6.1 above, 108 mutants remained alive. The mutants were then examined and it was found that many of them could be equivalenced. For example, the absolute value insertion mutant type (abs) was equivalenced at several points in the program since at those points, negative values were impossible due to the structure of the code. Thirty-one mutants were then left remaining. With these few remaining mutants, it was much easier to see which statements were not being executed. Additional test cases were then added and the remaining mutants were killed (ref TABLE 4). This confirmed the expected output, since it was known that the program was correct.

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
abs	126	0	0.0	77	49
crp	29	0	0.0	0	29
lsr	13	0	0.0	1	12
ror	99	0	0.0	4	95
uoi	83	0	0.0	2	81
TOTALS	350	0	0.0	84	266

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	0	0	0.0	0	0
dmn	238	0	0.0	79	159
opm	0	0	0.0	0	0
prd	112	0	0.0	5	107
scl	0	0	0.0	0	0
stm	0	0	0.0	0	0

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	350	0	0.0	84	266
cca	0	0	0.0	0	0
pda	350	0	0.0	84	266
sal	0	0	0.0	0	0

MOTHRA - DOMAIN & PREDICATE MUTANTS (CORRECT PROGRAM)

TABLE 4

6.3 RXVP80

Using RXVP80, an output report was created which identified all the DD-PATHs in the triangle program (ref FIGURE 7). A DD-PATH tree (ref FIGURE 8) was manually created to aid understanding of the triangle program and determine exactly how RXVP80 created the DD-PATHs. On entry to a function or subroutine, the entry point is always DD-PATH 1. For IF statements, the TRUE branch is assigned an even number DD-PATH and the FALSE branch an odd number DD-PATH.

By using RXVP80 on the triangle program, it was found that the original 35 test cases exercised 100% of the program DD-PATHs, and that the output from each test case was correct (as it should be, since this is a correct program). Thus, as was expected, no errors were found.

7.0 DOMAIN ERROR

An error was introduced into the correct program, and both the MOTHRA and RXVP80 were used to see if they would detect the error. The triangle program was modified such that a domain error was created. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the program.[1] The error was created by modifying the predicate on line 18 of the triangle subroutine (ref FIGURE 9). Line 18, IF (I .NE. J) GOTO 10, was changed to: IF (-I .NE. J) GOTO 10.

7.1 MOTHRA - STATEMENT ANALYSIS MUTANTS

The domain error was created via the VAX/ULTRIX editor. The program was then entered into the MOTHRA system. The statement analysis class was selected, and all mutants belonging to that class were enabled (i. e., the test strength was 100).

When the test cases were entered, test case 3, test input (2,2,1) gave an incorrect output. The triangle program output identified the triangle as scalene when it should have been isosceles. This incorrect

[1] White, Cohen, and Zeil, "A Domain Strategy for Computer Program Testing," Computer Program Testing, September 1981, 103-113.

```
DD-PATH DEFINITIONS            SUBROUTINE TRIANGLE ( I , J , K , MATCH )
  STMT NEST  LINE   SOURCE...
------------------------------------------------------------------------------------------------------------------------
              1
              2
   1          3         SUBROUTINE TRIANGLE(I,J,K,MATCH)
                                                                        ** DDPATH   1 IS PROCEDURE ENTRY
   2          5         INTEGER I,J,K,MATCH
              6
              7   C     MATCH is output from the subroutine:
              8   C     MATCH = 1 IF THE TRIANGLE IS SCALENE
              9   C     MATCH = 2 IF THE TRIANGLE IS ISOSCELES
             10   C     MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
             11   C     MATCH = 4 IF NOT A TRIANGLE
             12
             13   C     After a quick confirmation that it's a legal
             14   C     triangle, detect any sides of equal length
   3         16         IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
                                                                        ** DDPATH   2 IS TRUE BRANCH
                                                                        ** DDPATH   3 IS FALSE BRANCH
   5         17         MATCH=0
   6         18         IF (-I.NE.J) GOTO 10
                                                                        ** DDPATH   4 IS TRUE BRANCH
                                                                        ** DDPATH   5 IS FALSE BRANCH
   8         19         MATCH=MATCH+1
   9         20  10     IF (I.NE.K) GOTO 20
                                                                        ** DDPATH   6 IS TRUE BRANCH
                                                                        ** DDPATH   7 IS FALSE BRANCH
  11         21         MATCH=MATCH+2
  12         22  20     IF (J.NE.K) GOTO 30
                                                                        ** DDPATH   8 IS TRUE BRANCH
                                                                        ** DDPATH   9 IS FALSE BRANCH
  14         23         MATCH=MATCH+3
  15         24  30     IF (MATCH.NE.0) GOTO 100
             25
             26   C     Confirm it's a legal triangle before declaring
             27   C     it to be scalene
                                                                        ** DDPATH  10 IS TRUE BRANCH
                                                                        ** DDPATH  11 IS FALSE BRANCH
  17         29         IF (I+J.LE.K)  GOTO 500
                                                                        ** DDPATH  12 IS TRUE BRANCH
                                                                        ** DDPATH  13 IS FALSE BRANCH
  19         30         IF (J+K.LE.I)  GOTO 500
                                                                        ** DDPATH  14 IS TRUE BRANCH
                                                                        ** DDPATH  15 IS FALSE BRANCH
  21         31         IF (I+K.LE.J)  GOTO 500
                                                                        ** DDPATH  16 IS TRUE BRANCH
                                                                        ** DDPATH  17 IS FALSE BRANCH
  23         32         MATCH=1
  24         33         RETURN
             34
------------------------------------------------------------------------------------------------------------------------
```

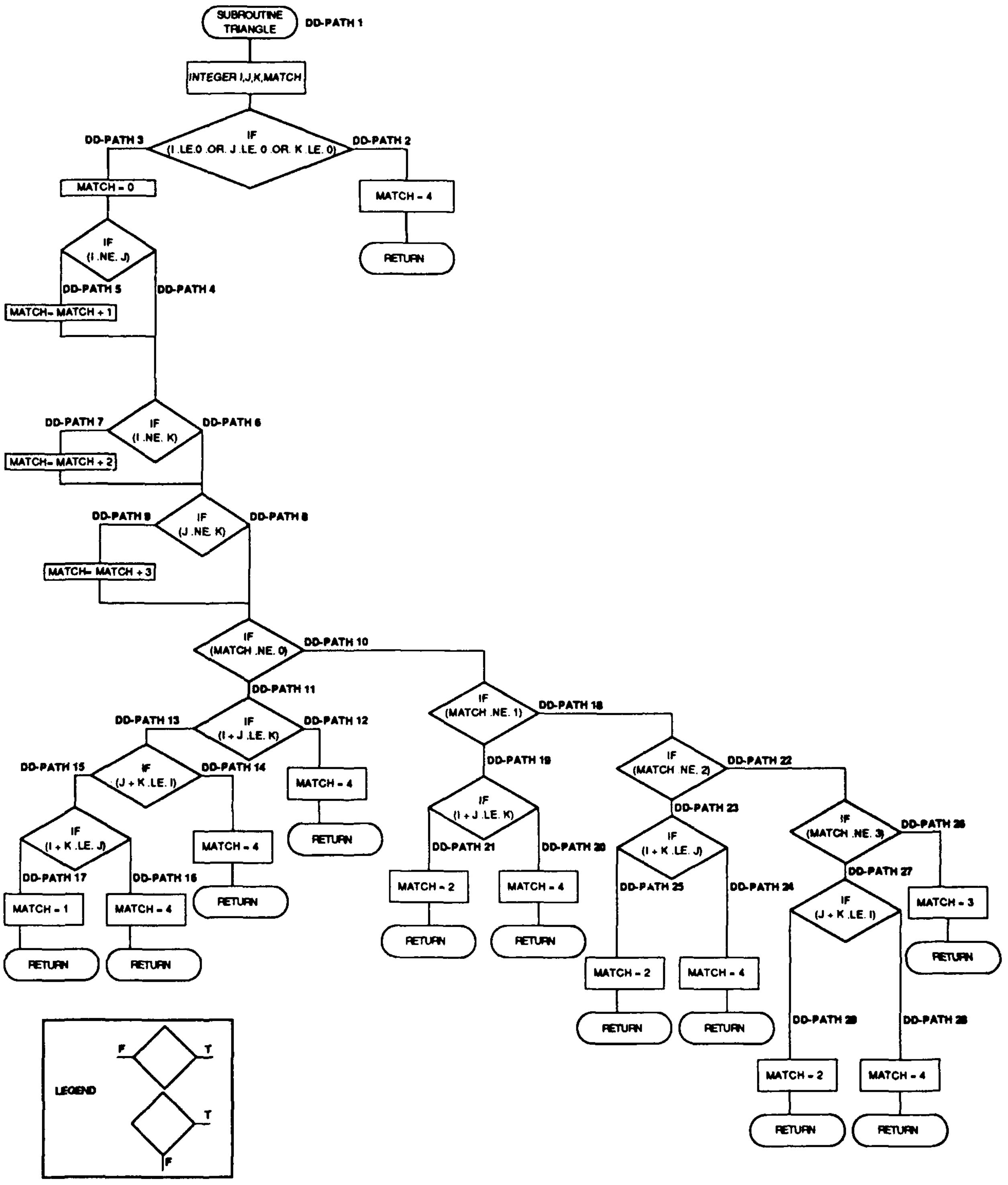
TRIANGLE PROGRAM DD-PATHS

FIGURE 7

```
-------------------------------------------------------------------------------------------------------
DD-PATH DEFINITIONS            SUBROUTINE TRIANGLE ( I , J , K , MATCH )
  STMT NEST  LINE   SOURCE...
-------------------------------------------------------------------------------------------------------
             35   C      Confirm is't a legal triangle before declaring
             36   C      it to be isosceles or equilateral
  25         38   100    IF (MATCH.NE.1) GOTO 200
                                                                     ** DDPATH  18 IS TRUE BRANCH
                                                                     ** DDPATH  19 IS FALSE BRANCH
  27         39          IF (I+J.LE.K) GOTO 500
                                                                     ** DDPATH  20 IS TRUE BRANCH
                                                                     ** DDPATH  21 IS FALSE BRANCH
  29         40   110    MATCH=2
  30         41          RETURN
  31         42   200    IF (MATCH.NE.2) GOTO 300
                                                                     ** DDPATH  22 IS TRUE BRANCH
                                                                     ** DDPATH  23 IS FALSE BRANCH
  33         43          IF (I+K.LE.J)  GOTO 500
                                                                     ** DDPATH  24 IS TRUE BRANCH
                                                                     ** DDPATH  25 IS FALSE BRANCH
  35         44          GOTO 110
  36         45   300    IF (MATCH.NE.3) GOTO 400
                                                                     ** DDPATH  26 IS TRUE BRANCH
                                                                     ** DDPATH  27 IS FALSE BRANCH
  38         46          IF (J+K.LE.I)  GOTO 500
                                                                     ** DDPATH  28 IS TRUE BRANCH
                                                                     ** DDPATH  29 IS FALSE BRANCH
  40         47          GOTO 110
  41         48   400    MATCH=3
  42         49          RETURN
             50
             51   C      Can't fool this program, thats not a triangle
  43         53   500    MATCH=4
  44         54          RETURN
  45         55          END
-------------------------------------------------------------------------------------------------------
```

TRIANGLE PROGRAM DD-PATHS

FIGURE 7 (continued)



DD-PATH TREE

FIGURE 8

```
      SUBROUTINE TRIANGLE(I,J,K,MATCH)

      integer   i,j,k,match

C     MATCH is output from the subroutine:
C     MATCH = 1 IF THE TRIANGLE IS SCALENE
C     MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C     MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C     MATCH = 4 IF NOT A TRIANGLE

C     After a quick confirmation that it's a legal
C     triangle, detect any sides of equal length
      IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
      MATCH=0
      IF (-I.NE.J) GOTO 10 -- DOMAIN ERROR
      MATCH=MATCH+1
10    IF (I.NE.K) GOTO 20
      MATCH=MATCH+2
20    IF (J.NE.K) GOTO 30
      MATCH=MATCH+3
30    IF (MATCH.NE.0) GOTO 100

C     Confirm it's a legal triangle before declaring it to be scalene
      IF (I+J.LE.K)  GOTO 500
      IF (J+K.LE.I)  GOTO 500
      IF (I+K.LE.J)  GOTO 500
      MATCH=1
      Return

C     Confirm it's a legal triangle before declaring it to be isosceles or
C     equilateral
100   IF (MATCH.NE.1) GOTO 200
      IF (I+J.LE.K) GOTO 500
110   MATCH=2
      RETURN
200   IF (MATCH.NE.2) GOTO 300
      IF (I+K.LE.J)  GOTO 500
      GOTO 110
300   IF (MATCH.NE.3) GOTO 400
      IF (J+K.LE.I)  GOTO 500
      GOTO 110
400   MATCH=3
      RETURN

C     Can't fool this program, that's not a triangle
500   MATCH=4
      RETURN
      END
```

TRIANGLE PROGRAM (DOMAIN ERROR)

FIGURE 9

output immediately indicates there is an error. MOTHRA was then used to help pinpoint the error.

After inputting the original 35 test cases plus several additional test cases, 18 mutants were still alive (ref TABLE 5). The same mutant described in section 6.1 was equivalenced. Next, possible reasons for the remaining mutants were examined. Examination of Figure 10 shows that there are two groups of remaining live mutants (mutants are identified in the program via the "#" symbol). One group represents the mutants that are created to replace the statement: IF (I+J .LE. K) GOTO 500. As shown in Figure 10, the mutant statements for each original statement are displayed beneath the statement they replace. When MOTHRA executes, it replaces the original statement with one mutant statement. Thus, as many new programs are created as mutant statements. None of these were killed. Further examination shows that this statement was never executed because of the statement directly above it: IF (MATCH .NE. 1) GOTO 200. MATCH was never being set to 1 so the program execution was always jumping to statement 200. To find the reason for this, the other group of mutants, those that were created to replace the statement: MATCH = MATCH + 1, were checked. It was clear that this statement was not being executed. The statement directly above it was examined: IF (-I .NE. J) GOTO 10. What was happening was that the program was always going to statement 10 because there was an error in the IF statement. The error was found!

7.2 MOTHRA - PREDICATE & DOMAIN MUTANTS

Using MOTHRA, the predicate and domain class of mutants was selected. Using the same test cases as those input to the correct program previously mutated using the predicate and domain class, three inputs gave incorrect outputs. Obviously the program was in error and MOTHRA was used to try to find the error. After numerous test case inputs, fifty-three mutants were remaining (ref TABLE 6). The live mutants were examined (ref FIGURE 11). The first set of mutants replaced the statement: MATCH = MATCH + 1. It was clear that this statement was not being executed. To find a reason for this, the code was examined. The previous statement: IF (-I .NE. J) GOTO 10 indicated that the program execution was going to statement 10 and not executing the MATCH = MATCH + 1 statement.

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
glr	128	8	6.3	0	120
rsr	38	4	10.5	0	34
san	36	3	8.3	0	34
sdl	41	3	7.3	0	38
TOTALS	243	18	7.4	0	225

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	166	12	7.2	0	154
dmn	0	0	0.0	0	0
opm	0	0	0.0	0	0
prd	0	0	0.0	0	0
scl	0	0	0.0	0	0
stm	77	6	7.8	0	71

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	243	18	7.4	0	225
cca	0	0	0.0	0	0
pda	0	0	0.0	0	0
sal	243	18	7.4	0	225

MOTHRA - STATEMENT ANALYSIS MUTANTS
DOMAIN ERROR

TABLE 5

```
        SUBROUTINE TRIANGLE(I,J,K,MATCH)

        integer  i,j,k,match

C       MATCH is output from the subroutine:
C       MATCH = 1 IF THE TRIANGLE IS SCALENE
C       MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C       MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C       MATCH = 4 IF NOT A TRIANGLE

C       After a quick confirmation that it's a legal
C       triangle, detect any sides of equal length
        IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
        MATCH=0
        IF (-I.NE.J) GOTO 10 -- DOMAIN ERROR
        MATCH=MATCH+1
#   rsr     134   #     RETURN
#   san     170   #   *** TRAP ***
#   sdl     208   #    CONTINUE
10      IF (I.NE.K) GOTO 20
        MATCH=MATCH+2
20      IF (J.NE.K) GOTO 30
        MATCH=MATCH+3
30      IF (MATCH.NE.0) GOTO 100

C       Confirm it's a legal triangle before declaring it to be scalene
        IF (I+J.LE.K)  GOTO 500
        IF (J+K.LE.I)  GOTO 500
        IF (I+K.LE.J)  GOTO 500
        MATCH=1
        Return

C       Confirm it's a legal triangle before declaring it to be isosceles
C       or equilateral
100     IF (MATCH.NE.1) GOTO 200
        IF (I+J.LE.K) GOTO 500
#   rsr     152   #     RETURN
#   san     188   #   *** TRAP ***
#   sdl     227   #    CONTINUE
#   rsr     153   #     IF ((I + J) .LE. K) RETURN
#   san     189   #     IF ((I + J) .LE. K) *** TRAP ***
#   sdl     228   #     IF ((I + J) .LE. K) CONTINUE
#   glr     73    #     IF ((I + J) .LE. K) GO TO 400
#   glr     74    #     IF ((I + J) .LE. K) GO TO 300
#   glr     75    #     IF ((I + J) .LE. K) GO TO 200
#   glr     76    #     IF ((I + J) .LE. K) GO TO 110
```

MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS
DOMAIN ERROR

FIGURE 10

```
#   glr     77    #     IF ((I + J) .LE. K) GO TO 100
#    glr     78    #      IF ((I + J) .LE. K) GO TO 30
#    glr     79    #      IF ((I + J) .LE. K) GO TO 20
#    glr     80    #      IF ((I + J) .LE. K) GO TO 10
110       MATCH=2
          RETURN
200       IF (MATCH.NE.2) GOTO 300
          IF (I+K.LE.J)  GOTO 500
          GOTO 110
#    rsr     159   #      RETURN
300       IF (MATCH.NE.3) GOTO 400
          IF (J+K.LE.I)  GOTO 500
          GOTO 110
400       MATCH=3
          RETURN

C         Can't fool this program, that's not a triangle
500       MATCH=4
          RETURN
          END
```

MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS DOMAIN ERROR

FIGURE 10 (continued)

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
abs	126	27	21.4	21	78
crp	29	2	6.9	0	27
lsr	13	0	0.0	0	13
ror	99	11	11.1	0	88
uoi	83	13	15.7	0	70
TOTALS	350	53	15.1	21	276

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	0	0	0.0	0	0
dmn	238	42	17.6	21	175
opm	0	0	0.0	0	0
prd	112	11	9.8	0	101
scl	0	0	0.0	0	0
stm	0	0	0.0	0	0

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	350	53	15.1	21	276
cca	0	0	0.0	0	0
pda	350	53	15.1	21	276
sal	0	0	0.0	0	0

MOTHRA - PREDICATE & DOMAIN MUTANTS
DOMAIN ERROR

TABLE 6

```
         SUBROUTINE TRIANGLE(I,J,K,MATCH)

         integer  i,j,k,match
C        MATCH is output from the subroutine:
C        MATCH = 1 IF THE TRIANGLE IS SCALENE
C        MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C        MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C        MATCH = 4 IF NOT A TRIANGLE
C        After a quick confirmation that it's a legal
C        triangle, detect any sides of equal length
         IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
         MATCH=0
         IF (-I.NE.J) GOTO 10 -- DOMAIN ERROR
         MATCH=MATCH+1
#   abs 17    #     MATCH = NEGABS (MATCH) + 1
#   abs 18    #     MATCH = ZPUSH (MATCH) + 1
#   uoi 283   #     MATCH = ( - MATCH) + 1
#   crp 135   #     MATCH = ZPUSH (MATCH) + 1
#   abs 20    #     MATCH = NEGABS (MATCH + 1)
#   abs 21    #     MATCH = ZPUSH (MATCH + 1)
#   uoi 284   #     MATCH = - (MATCH + 1)
#   uoi 285   #     MATCH = ++ (MATCH + 1)
#   uoi 286   #     MATCH = - - (MATCH + 1)
10       IF (I.NE.K) GOTO 20
         MATCH=MATCH+2
#   abs 29    #     MATCH = NEGABS (MATCH) + 2
#   uoi 291   #     MATCH = ( - MATCH) + 2
#   abs 33    #     MATCH = ZPUSH (MATCH + 2)
20       IF (J.NE.K) GOTO 30
         MATCH=MATCH+3
#   abs 45    #     MATCH = ZPUSH (MATCH + 3)
30       IF (MATCH.NE.0) GOTO 100
C        Confirm it's a legal triangle before declaring it to be scalene
         IF (I+J.LE.K)  GOTO 500
         IF (J+K.LE.I)  GOTO 500
         IF (I+K.LE.J)  GOTO 500
         MATCH=1
         Return
C        Confirm it's a legal triangle before declaring it to be isosceles
C        or equilateral
100  IF (MATCH.NE.1) GOTO 200
#   abs    83    #    100   IF (NEGABS (MATCH) .NE. 1) GO TO 200
#   abs    84    #    100   IF (ZPUSH (MATCH) .NE. 1) GO TO 200
#   uoi    321   #    100   IF (( - MATCH) .NE. 1) GO TO 200
#   uoi    322   #    100   IF (( ++ MATCH) .NE. 1) GO TO 200
#   crp    144   #    100   IF (MATCH .NE. 0) GO TO 200
#   uoi    324   #    100   IF (MATCH .NE. (- 1)) GO TO 200
#   ror    235   #    100   IF (MATCH .GT. 1) GO TO 200
```

MOTHRA - "LIVE" PREDICATE & DOMAIN MUTANTS
DOMAIN ERROR

FIGURE 11

```
#   ror 236   #     100   IF (MATCH .GE. 1) GO TO 200
#    ror 237    #       100   IF (.TRUE.) GO TO 200
          IF (I+J.LE.K) GOTO 500
#    abs 86     #        IF ((NEGABS (I) + J) .LE. K) GO TO 500
#    abs 87     #        IF ((ZPUSH (I) + J) .LE. K) GO TO 500
#    uoi 325    #       IF (((- I) + J) .LE. K) GO TO 500
#    abs 88     #        IF ((I + NEGABS (J)) .LE. K) GO TO 500
#    abs 90     #        IF ((I + ZPUSH(J)) .LE. K) GO TO 500
#    abs 92     #        IF (NEGABS (I - J) .LE. K) GO TO 500
#    abs 93     #        IF (ZPUSH(I + J) .LE. K) GO TO 500
#    uoi 326    #       IF ((- (I + J)) .LE. K) GO TO 500
#    uoi 327    #       IF ((++ (I + J)) .LE. K) GO TO 500
#    uoi 328    #       IF ((- - (I + J)) .LE. K) GO TO 500
#    abs 95     #        IF ( (I + J) .LE. NEGABS (K)) GO TO 500
#    abs 96     #        IF ((I + J) .LE. ZPUSH (K)) GO TO 500
#    uoi 329    #       IF ((I + J) .LE. (- K)) GO TO 500
#    ror 238    #       IF ((I + J) .LT. K) GO TO 500
#    ror 239    #       IF ((I + J) .EQ. K) GO TO 500
#    ror 240    #       IF ((I + J) .NE. K) GO TO 500
#    ror 241    #       IF ((I + J) .GT. K) GO TO 500
#    ror 242    #       IF ((I + J) .GE. K) GO TO 500
#    ror 243    #       IF (.TRUE.) GO TO 500
110       MATCH=2
          RETURN
200       IF (MATCH.NE.2) GOTO 300
#    abs 99     # 200   IF (ZPUSH (MATCH) .NE. 2) GO TO 300
#    ror 247 # 200   IF (MATCH .GT. 2) GO TO 300
          IF (I+K.LE.J)  GOTO 500
#    abs 102    #       IF ((ZPUSH (I) + K) .LE. J) GO TO 500
#    abs 105    #       IF ((I + ZPUSH(K)) .LE. J) GO TO 500
#    abs 108    #       IF (ZPUSH(I + K) .LE. J) GO TO 500
#    abs 111    #       IF ((I + K) .LE. ZPUSH (J)) GO TO 500
          GOTO 110
300       IF (MATCH.NE.3) GOTO 400
#    abs 114    #   300   IF (ZPUSH (MATCH) .NE. 3) GO TO 400
#    ror 259    #   300   IF (MATCH .GT. 3) GO TO 400
          IF (J+K.LE.I)  GOTO 500
#    abs 117    #       IF ((ZPUSH (J) + K) .LE. I) GO TO 500
#    abs 120    #       IF ((J + ZPUSH(K)) .LE. I) GO TO 500
#    abs 123    #       IF (ZPUSH(J + K) .LE. I) GO TO 500
#    abs 126    #       IF ((J + K) .LE. ZPUSH (I)) GO TO 500
          GOTO 110
400       MATCH=3
          RETURN
C         Can't fool this program, that's not a triangle
500       MATCH=4
          RETURN
          END
```

MOTHRA - "LIVE" PREDICATE & DOMAIN MUTANTS
DOMAIN ERROR

FIGURE 11 (continued)

The next large group of mutants which were not killed were examined. These replaced the statement: IF (I+J .LE. K) GOTO 500. Again, by the large number of mutants remaining in this group, it was clear that this statement was not being executed. Looking at the previous statement: IF (MATCH .NE. 1) GOTO 200 indicated that MATCH was not being set to 1. MATCH was not set to 1 because, as indicated by the first group of mutants, the statement MATCH = MATCH + 1 was not executed. For each group of mutants, the flow of execution returned to the same point and it was finally noticed that the IF statement: IF (-I .NE. J) GOTO 10 was in error. It should have been: IF (I .NE. J) GOTO 10!

Overall, it was found that mutating for predicate and domain mutants provided too much information for this small non-critical testing experiment. It was easier to locate an error (domain error) when statement analysis mutants were created. However, the same conclusions were drawn as to what caused the error in the program - it just took the test personnel much longer.

7.3 RXVP80

The same domain error was created by modifying the subroutine triangle via the VAX/VMS editor. The program was then input to RXVP80, instrumented (insertion of software probes) for DD-PATH coverage and run against the previous set of test cases. After additional test cases were added, only 86% of the paths had been executed. RXVP80 identified DD-PATHs 5, 19, 20 and 21 as paths that were not executed (ref FIGURE 12).

This led to investigation of the DD-PATH report which identifies each DD-PATH, ref FIGURE 7. DD-PATH 5 is the FALSE branch of: IF (-I .NE. J) GOTO 10, which meant that for none of the test cases was -I = J. On first examination of the program, it appears that inputting (2,-2,2) for the sides of the triangle would cause this DD-PATH to be traversed. However, closer examination revealed that a previous statement checked for negative input values. If negative input was encountered, then the program control flow jumps to a statement at the end of the program. The result is that this DD-PATH could never be exercised, and that the assignment statement on this DD-PATH which sets MATCH to 1 would not occur. DD-PATHs 19, 20, and 21 were also not exercised. DD-PATH 19, 20, and 21 emerge from the false branch of: IF (MATCH .NE. 1) GOTO 200. This means that for

```
------------------------------------------------------------------------------------------------------------
                                          PATH ANALYZER CUMULATIVE DETAILED REPORT
FOR MODULE TRIANGLE

------------------------------------------------------------------------------------------------------------
CUMULATIVE RESULTS OF    39 TEST CASES

------------------------------------------------------------------------------------------------------------
DD-PATH |    NUMBER    |   NUMBER OF EXECUTIONS (NORMALIZED TO MAXIMUM)      | DD-PATH | NUMBER OF   |
NUMBER  | NOT EXECUTED | .--------20.--------40.--------60.--------80.------100 | NUMBER  | TRAVERSALS  |
------------------------------------------------------------------------------------------------------------
        |              |                                                     |         |             |
   1    |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  |    1    |         39  |
   2    |              | XXXXXXXXXXXXXXXXXXXXX                               |    2    |         17  |
   3    |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXX                        |    3    |         22  |
   4    |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXX                        |    4    |         22  |
        |   (    5)    |                                                     |         |             |
   6    |              | XXXXXXXXXXXXXXXXXXXXX                               |    6    |         17  |
   7    |              | XXXXXX                                              |    7    |          5  |
   8    |              | XXXXXXXXXXXXXXXXXXX                                 |    8    |         15  |
   9    |              | XXXXXXXX                                            |    9    |          7  |
  10    |              | XXXXXXXXXXXX                                        |   10    |         10  |
  11    |              | XXXXXXXXXXXXXXX                                     |   11    |         12  |
  12    |              | XXXXX                                               |   12    |          4  |
  13    |              | XXXXXXXXXX                                          |   13    |          8  |
  14    |              | X                                                   |   14    |          1  |
  15    |              | XXXXXXXX                                            |   15    |          7  |
  16    |              | X                                                   |   16    |          1  |
  17    |              | XXXXXXX                                             |   17    |          6  |
  18    |              | XXXXXXXXXXXX                                        |   18    |         10  |
        |   (   19)    |                                                     |         |             |
 ....   |     ....     |                                                     |  ....   |             |
        |   (   21)    |                                                     |         |             |
  22    |              | XXXXXXXX                                            |   22    |          7  |
  23    |              | XXX                                                 |   23    |          3  |
  24    |              | XX                                                  |   24    |          2  |
  25    |              | X                                                   |   25    |          1  |
  26    |              | XX                                                  |   26    |          2  |
  27    |              | XXXXXX                                              |   27    |          5  |
  28    |              | X                                                   |   28    |          1  |
  29    |              | XXXXX                                               |   29    |          4  |
------------------------------------------------------------------------------------------------------------
                                             TOTAL NUMBER OF DD-PATH TRAVERSALS =        228
TOTAL NUMBER OF DD-PATHS NOT EXECUTED =    4
TOTAL NUMBER OF DD-PATHS EXECUTED     =   25
     ** PERCENT EXECUTED  86.21 **
```

DD-PATH EXECUTION
(DOMAIN ERROR)

FIGURE 12

these DD-PATHs never to be executed, MATCH must not equal 1 at any point in the program. Looking back again at FIGURE 8, MATCH never equals 1 because DD-PATH 5 was never exercised by the test data. Thus DD-PATHS 19, 20, and 21 these can never be exercised because DD-PATH 5 was never taken. Upon close examination of the predicate of DD-PATH 5: IF (-I .NE. J) the error was realized. It should be: IF (I .NE. J)!

8.0 MISSING STATEMENT ERROR

The triangle program was modified, such that statement 17, MATCH = 0 was removed from the program (ref FIGURE 13).

8.1 MOTHRA

Using MOTHRA the statement analysis class of mutants was selected. There were 242 mutants created for the triangle subroutine. Using the original set of test cases, test case 1, test input (3,4,5) gave an incorrect output on the changed program. The output showed that the triangle was "isosceles", however the correct output for these values is a "scalene" triangle. Twelve other test cases gave incorrect outputs. At the end of 23 test cases, 71 mutants remained alive (ref TABLE 7).

In checking the live mutants, it was evident that certain statements were not being executed and thus the mutants could not be killed (ref FIGURE 14). These statements followed the false branch of the predicate: IF (MATCH .NE. 0) GOTO 100, which means that if MATCH = 0 the control flow goes to the statement IF (I+J .LE. K) GOTO 500. However, his statement was never being executed. Thus, it was clear that there was a problem in the program such that MATCH was never set to zero and the missing statement (i. e., MATCH = 0) was found.

8.2 RXVP80

The modified triangle program was input to RXVP80 along with the same test cases as previously entered. The DD-PATH execution report showed that 100% execution coverage was obtained (ref FIGURE 15)! The reason was that VMS was automatically initializing memory (and therefore MATCH), to zero, and the missing statement error was not found. The static analysis capability of RXVP80 was then used to identify SET/USE errors. SET/USE errors occur when

```
        SUBROUTINE TRIANGLE(I,J,K,MATCH)

        integer   i,j,k,match

C       MATCH is output from the subroutine:
C       MATCH = 1 IF THE TRIANGLE IS SCALENE
C       MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C       MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C       MATCH = 4 IF NOT A TRIANGLE

C       After a quick confirmation that it's a legal
C       triangle, detect any sides of equal length
        IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
        MATCH = 0     -- MISSING STATEMENT
        IF (I.NE.J) GOTO 10
        MATCH=MATCH+1
10      IF (I.NE.K) GOTO 20
        MATCH=MATCH+2
20      IF (J.NE.K) COTO 30
        MATCH=MATCH+3
30      IF (MATCH.NE.0) GOTO 100

C       Confirm it's a legal triangle before declaring it to be scalene
        IF (I+J.LE.K)  GOTO 500
        IF (J+K.LE.I)  GOTO 500
        IF (I+K.LE.J)  GOTO 500
        MATCH=1
        Return

C       Confirm it's a legal triangle before declaring
C       it to be isosceles or equilateral
100     IF (MATCH.NE.1) GOTO 200
        IF (I+J.LE.K) GOTO 500
110     MATCH=2
        RETURN
200     IF (MATCH.NE.2) GOTO 300
        IF (I+K.LE.J)  GOTO 500
        GOTO 110
300     IF (MATCH.NE.3) GOTO 400
        IF (J+K.LE.I)  GOTO 500
        GOTO 110
400     MATCH=3
        RETURN

C       Can't fool this program, that's not a triangle
500     MATCH=4
        RETURN
        END
```

TRIANGLE PROGRAM (MISSING STATEMENT)

FIGURE 13

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
glr	128	40	31.3	0	88
rsr	37	10	27.0	0	27
san	37	10	27.0	0	27
sdl	40	11	27.5	0	29
TOTALS	242	71	29.3	0	171

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	165	50	30.3	0	115
dmn	0	0	0.0	0	0
opm	0	0	0.0	0	0
prd	0	0	0.0	0	0
scl	0	0	0.0	0	0
stm	77	21	27.3	0	56

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	242	71	29.3	0	171
cca	0	0	0.0	0	0
pda	0	0	0.0	0	0
sal	242	71	29.3	0	171

MOTHRA - STATEMENT ANALYSIS MUTANTS
MISSING STATEMENT ERROR

TABLE 7

```
          SUBROUTINE TRIANGLE(I,J,K,MATCH)

          integer   i,j,k,match

C         MATCH is output from the subroutine:
C         MATCH = 1 IF THE TRIANGLE IS SCALENE
C         MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C         MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C         MATCH = 4 IF NOT A TRIANGLE
C         After a quick confirmation that it's a legal
C         triangle, detect any sides of equal length
          IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
          MATCH = 0  --  MISSING STATEMENT
          IF (I.NE.J) GOTO 10
          MATCH=MATCH+1
10        IF (I.NE.K) GOTO 20
          MATCH=MATCH+2
20        IF (J.NE.K) GOTO 30
          MATCH=MATCH+3
30        IF (MATCH.NE.0) GOTO 100

C         Confirm it's a legal triangle before declaring it to be scalene
          IF (I+J.LE.K)  GOTO 500
   #  rsr  142   #      RETURN
   #  san  179   #   ***  TRAP  ***
   #  sdl  216   #     CONTINUE
   #  rsr  143   #      IF ((I + J) .LE. K) RETURN
   #  san  180   #      IF ((I + J) .LE. K) *** TRAP ***
   #  sdl  217   #      IF ((I + J) .LE. K) CONTINUE
   #  glr  41    #      IF ((I + J) .LE. K) GO TO 400
   #  glr  42    #      IF ((I + J) .LE. K) GO TO 300
   #  glr  43    #      IF ((I + J) .LE. K) GO TO 200
   #  glr  44    #      IF ((I + J) .LE. K) GO TO 110
   #  glr  45    #      IF ((I + J) .LE. K) GO TO 100
   #  glr  46    #      IF ((I + J) .LE. K) GO TO 30
   #  glr  47    #      IF ((I + J) .LE. K) GO TO 20
   #  glr  48    #      IF ((I + J) .LE. K) GO TO 10
          IF (J+K.LE.I)  GOTO 500
   #  rsr  144   #      RETURN
   #  san  181   #   ***  TRAP  ***
   #  sdl  218   #     CONTINUE
   #  rsr  145   #      IF ((J + K) .LE. I) RETURN
   #  san  182   #      IF ((J + K) .LE. I) *** TRAP ***
   #  sdl  219   #      IF ((J + K) .LE. I) CONTINUE
   #  glr  49    #      IF ((J + K) .LE. I) GO TO 400
   #  glr  50    #      IF ((J + K) .LE. I) GO TO 300
   #  glr  51    #      IF ((J + K) .LE. I) GO TO 200
   #  glr  52    #      IF ((J + K) .LE. I) GO TO 110
```

MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS
MISSING STATEMENT ERROR

FIGURE 14

```
   #   glr   53    #      IF ((J + K) .LE. I) GO TO 100
   #   glr   54    #      IF ((J + K) .LE. I) GO TO 30
   #   glr   55    #      IF ((J + K) .LE. I) GO TO 20
   #   glr   56    #      IF ((J + K) .LE. I) GO TO 10
       IF (I+K.LE.J)  GOTO 500
   #   rsr   146   #      RETURN
   #   san   183   #   ***  TRAP  ***
   #   sdl   220   #     CONTINUE
   #   rsr   147   #      IF ((I + K) .LE. J) RETURN
   #   san   184   #       IF ((I + K) .LE. J) *** TRAP ***
   #   sdl   221   #      IF ((I + K) .LE. J) CONTINUE
   #   glr   57    #      IF ((I + K) .LE. J) GO TO 400
   #   glr   58    #      IF ((I + K) .LE. J) GO TO 300
   #   glr   59    #      IF ((I + K) .LE. J) GO TO 200
   #   glr   60    #      IF ((I + K) .LE. J) GO TO 110
   #   glr   61    #      IF ((I + K) .LE. J) GO TO 100
   #   glr   62    #      IF ((I + K) .LE. J) GO TO 30
   #   glr   63    #      IF ((I + K) .LE. J) GO TO 20
   #   glr   64    #      IF ((I + K) .LE. J) GO TO 10
       MATCH=1
   #   rsr   148   #      RETURN
   #   san   185   #   ***  TRAP  ***
   #   sdl   222   #     CONTINUE
       Return
   #   sdl   223   #     CONTINUE
C      Confirm it's a legal triangle before declaring it to be isosceles
C      or equilateral
100    IF (MATCH.NE.1) GOTO 200
       IF (I+J.LE.K) GOTO 500
110    MATCH=2
       RETURN
200    IF (MATCH.NE.2) GOTO 300
       IF (I+K.LE.J)  GOTO 500
   #   rsr   156   #      RETURN
   #   san   193   #   ***  TRAP  ***
   #   sdl   232   #     CONTINUE
   #   rsr   157   #      IF ((I + K) .LE. J) RETURN
   #   san   194   #       IF ((I + K) .LE. J) *** TRAP ***
   #   sdl   233   #      IF ((I + K) .LE. J) CONTINUE
   #   glr   89    #      IF ((I + K) .LE. J) GO TO 400
   #   glr   90    #      IF ((I + K) .LE. J) GO TO 300
   #   glr   91    #      IF ((I + K) .LE. J) GO TO 200
   #   glr   92    #      IF ((I + K) .LE. J) GO TO 110
   #   glr   93    #      IF ((I + K) .LE. J) GO TO 100
   #   glr   94    #      IF ((I + K) .LE. J) GO TO 30
   #   glr   95    #      IF ((I + K) .LE. J) GO TO 20
   #   glr   96    #      IF ((I + K) .LE. J) GO TO 10
```

MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS
MISSING STATEMENT ERROR

FIGURE 14 (continued)

```
       GOTO 110
   #   glr   97    #     GO TO 500
```

```
    #   glr   98    #      GO TO 400
    #   glr   99    #      GO TO 300
    #   glr   100   #      GO TO 200
    #   glr   101   #      GO TO 100
    #   glr   102   #      GO TO 30
    #   glr   103   #      GO TO 20
    #   glr   104   #      GO TO 10
    #   rsr   158   #       RETURN
    #   san   195   #   ***  TRAP  ***
    #   sdl   234   #      CONTINUE
300      IF (MATCH.NE.3) GOTO 400
         IF (J+K.LE.I)  GOTO 500
         GOTO 110
400      MATCH=3
         RETURN
C        Can't fool this program, that's not a triangle
500      MATCH=4
         RETURN
         END
```

**MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS
MISSING STATEMENT ERROR**

FIGURE 14 (continued)

```
===========================================================================================================
                                           PATH ANALYZER CUMULATIVE DETAILED REPORT
FOR MODULE TRIANGLE

-----------------------------------------------------------------------------------------------------------
CUMULATIVE RESULTS OF    37 TEST CASES

-----------------------------------------------------------------------------------------------------------
 DD-PATH |   NUMBER     |   NUMBER OF EXECUTIONS (NORMALIZED TO MAXIMUM)    | DD-PATH | NUMBER OF      |
 NUMBER  | NOT EXECUTED | .-------20.-------40.-------60.-------80.------100 | NUMBER  | TRAVERSALS     |
-----------------------------------------------------------------------------------------------------------
         |              |                                                   |         |                |
    1    |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX |    1    |            37  |
    2    |              | XXXXXXXXXXXXXXXXXXXX                              |    2    |            15  |
    3    |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXXX                     |    3    |            22  |
    4    |              | XXXXXXXXXXXXXXXXXXXXXX                            |    4    |            17  |
    5    |              | XXXXXX                                            |    5    |             5  |
    6    |              | XXXXXXXXXXXXXXXXXXXXXX                            |    6    |            17  |
    7    |              | XXXXXX                                            |    7    |             5  |
    8    |              | XXXXXXXXXXXXXXXXXXXX                              |    8    |            15  |
    9    |              | XXXXXXXXX                                         |    9    |             7  |
   10    |              | XXXXXXXXXXXXXXXXX                                 |   10    |            13  |
   11    |              | XXXXXXXXXXXX                                      |   11    |             9  |
   12    |              | XX                                                |   12    |             2  |
   13    |              | XXXXXXXXX                                         |   13    |             7  |
   14    |              | X                                                 |   14    |             1  |
   15    |              | XXXXXXXX                                          |   15    |             6  |
   16    |              | X                                                 |   16    |             1  |
   17    |              | XXXXXX                                            |   17    |             5  |
   18    |              | XXXXXXXXXXXXX                                     |   18    |            10  |
   19    |              | XXXX                                              |   19    |             3  |
   20    |              | XX                                                |   20    |             2  |
   21    |              | X                                                 |   21    |             1  |
   22    |              | XXXXXXXXX                                         |   22    |             7  |
   23    |              | XXXX                                              |   23    |             3  |
   24    |              | XX                                                |   24    |             2  |
   25    |              | X                                                 |   25    |             1  |
   26    |              | XX                                                |   26    |             2  |
   27    |              | XXXXXX                                            |   27    |             5  |
   28    |              | X                                                 |   28    |             1  |
   29    |              | XXXXX                                             |   29    |             4  |
-----------------------------------------------------------------------------------------------------------
                                       TOTAL NUMBER OF DD-PATH TRAVERSALS =         225
TOTAL NUMBER OF DD-PATHS NOT EXECUTED =   0
TOTAL NUMBER OF DD-PATHS EXECUTED     =  29
     ** PERCENT EXECUTED 100.00 **
```

**DD-PATH EXECUTION
(MISSING STATEMENT ERROR)**

FIGURE 15

variables are used before being assigned a value. None were found. RXVP80 reports showed that MATCH was set to a value via the statements MATCH = MATCH + 1, MATCH = MATCH + 2, and MATCH = MATCH + 3 and therefore, as far as RXVP80 was concerned, MATCH was set before being used. This shows a limitation of the SET/USE capability of RXVP80!

9.0 COMPUTATION ERROR

The triangle program was modified such that statement 19, MATCH = MATCH + 1 was changed to MATCH = MATCH - 1 (ref FIGURE 16).

9.1 MOTHRA

Using MOTHRA, the class of mutants that belong to statement analysis was selected. There were 243 mutants created for the triangle subroutine. At the end of the set of test case inputs, 15 mutants remained alive. Looking at each mutant, one was equivalenced since MATCH was always equal to 2 at that point in the program. There were then 14 live mutants remaining (ref TABLE 8). Examination of the location of the mutants in the program (ref FIGURE 17), showed that this was the result of the false branch of the predicate: IF (MATCH .NE. 1) never being executed (i.e., MATCH is never equal to 1). Therefore, it was necessary to check statements where MATCH is assigned values. Checking back in the program it was found that at:

line # 23, MATCH = MATCH + 3
line # 21, MATCH = MATCH + 2
line # 19, MATCH = MATCH - 1
line # 17, MATCH = 0

Since MATCH is originally set to zero, it never gets set to 1 on any of the subsequent branches. It was determined that MATCH could be set to one if: the false branch of IF (I .NE. J) is taken and MATCH = MATCH - 1, resulting in MATCH = -1 and then the false branch of IF (I .NE. K) is taken and MATCH = MATCH + 2 resulting in MATCH = 1. However, for these two conditions to be false, I would be equal to J and I would be equal to K, which means that J would be equal to K and the false branch of IF (J .NE. K) would be taken and MATCH = MATCH + 3 would be executed resulting in MATCH = 2.

```
          SUBROUTINE TRIANGLE(I,J,K,MATCH)

      integer   i,j,k,match

C     MATCH is output from the subroutine:
C     MATCH = 1 IF THE TRIANGLE IS SCALENE
C     MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C     MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C     MATCH = 4 IF NOT A TRIANGLE

C     After a quick confirmation that it's a legal
C     triangle, detect any sides of equal length
      IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
      MATCH=0
      IF (I.NE.J) GOTO 10
      MATCH=MATCH-1    -- COMPUTATION ERROR
10    IF (I.NE.K) GOTO 20
      MATCH=MATCH+2
20    IF (J.NE.K) GOTO 30
      MATCH=MATCH+3
30    IF (MATCH.NE.0) GOTO 100

C     Confirm it's a legal triangle before declaring it to be scalene
      IF (I+J.LE.K)  GOTO 500
      IF (J+K.LE.I)  GOTO 500
      IF (I+K.LE.J)  GOTO 500
      MATCH=1
      Return

C     Confirm it's a legal triangle before declaring
C     it to be isosceles or equilateral
100   IF (MATCH.NE.1) GOTO 200
      IF (I+J.LE.K) GOTO 500
110   MATCH=2
      RETURN
200   IF (MATCH.NE.2) GOTO 300
      IF (I+K.LE.J)  GOTO 500
      GOTO 110
300   IF (MATCH.NE.3) GOTO 400
      IF (J+K.LE.I)  GOTO 500
      GOTO 110
400   MATCH=3
      RETURN

C     Can't fool this program, that's not a triangle
500   MATCH=4
      RETURN
      END
```

TRIANGLE PROGRAM (COMPUTATION ERROR)

FIGURE 16

TYPE	GENERATED	LIVE	%LIVE	EQUIV	DEAD
glr	128	8	6.3	0	120
rsr	38	2	5.3	1	35
san	36	2	5.6	0	34
sdl	41	2	4.9	0	39
TOTALS	243	14	5.8	1	228

CLASS	GENERATED	LIVE	%LIVE	EQUIV	DEAD
ary	0	0	0.0	0	0
con	0	0	0.0	0	0
ctl	166	10	6.0	1	155
dmn	0	0	0.0	0	0
opm	0	0	0.0	0	0
prd	0	0	0.0	0	0
scl	0	0	0.0	0	0
stm	77	4	5.2	0	73

SUPERCL	GENERATED	LIVE	%LIVE	EQUIV	DEAD
all	243	14	5.8	1	228
cca	0	0	0.0	0	0
pda	0	0	0.0	0	0
sal	243	14	5.8	1	228

**MOTHRA - STATEMENT ANALYSIS MUTANTS
COMPUTATION ERROR**

TABLE 8

```
        SUBROUTINE TRIANGLE(I,J,K,MATCH)

        integer  i,i,k,match

C       MATCH is output from the subroutine:
C       MATCH = 1 IF THE TRIANGLE IS SCALENE
C       MATCH = 2 IF THE TRIANGLE IS ISOSCELES
C       MATCH = 3 IF THE TRIANGLE IS EQUILATERAL
C       MATCH = 4 IF NOT A TRIANGLE

C       After a quick confirmation that it's a legal
C       triangle, detect any sides of equal length
        IF (I .LE. 0 .OR. J .LE. 0 .OR. K .LE. 0) GOTO 500
        MATCH=0
        IF (I.NE.J) GOTO 10
        MATCH=MATCH-1 --  COMPUTATION ERROR
10      IF (I.NE.K) GOTO 20
        MATCH=MATCH+2
20      IF (J.NE.K) GOTO 30
        MATCH=MATCH+3
30      IF (MATCH.NE.0) GOTO 100

C       Confirm it's a legal triangle before declaring it to be scalene
        IF (I+J.LE.K)  GOTO 500
        IF (J+K.LE.I)  GOTO 500
        IF (I+K.LE.J)  GOTO 500
        MATCH=1
        Return

C       Confirm it's a legal triangle before declaring it to be isosceles
C       or equilateral
100     IF (MATCH.NE.1) GOTO 200
        IF (I+J.LE.K) GOTO 500
   #  rsr  152  #    RETURN
   #  san  188  #  *** TRAP ***
   #  sdl  227  #   CONTINUE
   #  rsr  153  #    IF ((I + J) .LE. K) RETURN
   #  san  189  #    IF ((I + J) .LE. K) *** TRAP ***
   #  sdl  228  #    IF ((I + J) .LE. K) CONTINUE
   #  glr  73   #    IF ((I + J) .LE. K) GO TO 400
   #  glr  74   #    IF ((I + J) .LE. K) GO TO 300
   #  glr  75   #    IF ((I + J) .LE. K) GO TO 200
   #  glr  76   #    IF ((I + J) .LE. K) GO TO 110
   #  glr  77   #    IF ((I + J) .LE. K) GO TO 100
   #  glr  78   #    IF ((I + J) .LE. K) GO TO 30
   #  glr  79   #    IF ((I + J) .LE. K) GO TO 20
   #  glr  80   #    IF ((I + J) .LE. K) GO TO 10
110     MATCH=2
```

**MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS
COMPUTATION ERROR**

FIGURE 17

```
        RETURN
200     IF (MATCH.NE.2) GOTO 300
        IF (I+K.LE.J)  GOTO 500
        GOTO 110
300     IF (MATCH.NE.3) GOTO 400
        IF (J+K.LE.I)  GOTO 500
        GOTO 110
400     MATCH=3
        RETURN

C       Can't fool this program, that's not a triangle
500     MATCH=4
        RETURN
        END
```

MOTHRA - "LIVE" STATEMENT ANALYSIS MUTANTS COMPUTATION ERROR

FIGURE 17 (continued)

Therefore, it was clear that MATCH never stays equal to 1 for this program. Upon closer examination, it was seen that this was a result of one of the assignment statements being in error and it was found that MATCH = MATCH - 1 on line 19. It should be MATCH = MATCH + 1, therefore the error was found.

9.2 RXVP80

The modified triangle program was input to RXVP80, using the same set of test cases in addition to several new test cases. The DD-PATH execution report showed that DD-PATHs 19, 20, and 21 were not executed (ref FIGURE 18). Examination of the DD-PATHs not traversed showed that this was the result of DD-PATH 19 never being taken. This meant that MATCH was never equal to one. As in the MOTHRA system, the statements where MATCH is assigned a value were examined. These statements were as follows:

```
MATCH = MATCH + 3 on DD-PATH 9
MATCH = MATCH + 2 on DD-PATH 7
MATCH = MATCH - 1 on DD-PATH 5
MATCH = 0         on DD-PATH 3
```

For the reasons stated in the previous MOTHRA section, it was clear that one of the assignment statements (i.e., MATCH = MATCH - 1) was in error. Again, the error was found.

10.0 CONCLUSION

Both tools performed very well. It was felt that RXVP80 was easier to learn and use, and easier to find the error types that were created. The DD-PATH concept is very clear and the graphical chart that was manually created aided understanding of this test technique. The DD-PATH concept has been used on many large-scale software development efforts and has proven its usefulness.

MOTHRA is a newer and much more complex system (but a more powerful testing system). It was more difficult to learn and use and very time consuming to equivalence mutants. With so many mutants being created, especially for the Predicate and Domain mutant class, using MOTHRA was difficult to detect the errors. Except for the Missing Statement Error, RXVP80 found the errors and displayed them to the tester in a more understandable manner. Overall, while mutation testing certainly ..as the potential to surpass

```
=========================================================================================================
                                          PATH ANALYZER CUMULATIVE DETAILED REPORT
FOR MODULE TRIANGLE

---------------------------------------------------------------------------------------------------------
CUMULATIVE RESULTS OF    35 TEST CASES

---------------------------------------------------------------------------------------------------------
DD-PATH |    NUMBER    |   NUMBER OF EXECUTIONS  (NORMALIZED TO MAXIMUM)    | DD-PATH | NUMBER OF      |
NUMBER  | NOT EXECUTED | .-------20.-------40.-------60.-------80.------100 | NUMBER  | TRAVERSALS     |
---------------------------------------------------------------------------------------------------------
        |              |                                                    |         |                |
    1   |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX |     1   |             35 |
    2   |              | XXXXXXXXXXXXXXXXXX                                 |     2   |             13 |
    3   |              | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                    |     3   |             22 |
    4   |              | XXXXXXXXXXXXXXXXXXXXXXXXX                          |     4   |             18 |
    5   |              | XXXXX                                              |     5   |              4 |
    6   |              | XXXXXXXXXXXXXXXXXXXXXXXX                           |     6   |             17 |
    7   |              | XXXXXXX                                            |     7   |              5 |
    8   |              | XXXXXXXXXXXXXXXXXXXX                               |     8   |             14 |
    9   |              | XXXXXXXXXXX                                        |     9   |              8 |
   10   |              | XXXXXXXXXXXXXXXXXX                                 |    10   |             13 |
   11   |              | XXXXXXXXXXXX                                       |    11   |              9 |
   12   |              | XX                                                 |    12   |              2 |
   13   |              | XXXXXXXXXX                                         |    13   |              7 |
   14   |              | X                                                  |    14   |              1 |
   15   |              | XXXXXXXX                                           |    15   |              6 |
   16   |              | X                                                  |    16   |              1 |
   17   |              | XXXXXXX                                            |    17   |              5 |
   18   |              | XXXXXXXXXXXXXXXXXX                                 |    18   |             13 |
        |    (  19)    |                                                    |         |                |
  ....  |     ....     |                                                    |   ....  |                |
        |    (  21)    |                                                    |         |                |
   22   |              | XXXXXXXXXXXXXX                                     |    22   |             10 |
   23   |              | XXXX                                               |    23   |              3 |
   24   |              | XX                                                 |    24   |              2 |
   25   |              | X                                                  |    25   |              1 |
   26   |              | XXXXX                                              |    26   |              4 |
   27   |              | XXXXXXXX                                           |    27   |              6 |
   28   |              | XX                                                 |    28   |              2 |
   29   |              | XXXXX                                              |    29   |              4 |
---------------------------------------------------------------------------------------------------------
                                                 TOTAL NUMBER OF DD-PATH TRAVERSALS =         225
TOTAL NUMBER OF DD-PATHS NOT EXECUTED =    3
TOTAL NUMBER OF DD-PATHS EXECUTED     =   26
     ** PERCENT EXECUTED  89.66 **
```

DD-PATH EXECUTION
(COMPUTATION ERROR)

FIGURE 18

all other forms of testing, for certain classes of errors (i.e., missing statement errors), the much easier to use static and dynamic style of testing provides comparable results.

Mutation Testing's different mutant operators, however, allow an extremely thorough testing strategy and is especially important for testing mission critical applications. It allows testers to match the degree of testing to the criticality of the application and the amount of resources available for testing. Its statement mutants allow it to provide statement level coverage, which overlaps RXVP80's capabilities.

Future work is planned to enhance the MOTHRA system and ensure a more usable testing system. As a result of this testing project, recommendations for enhancements include:

<u>Automated support to determine equivalent mutants.</u> If all classes of mutants are enabled, there are on the order of N^2 mutants created for an N line computer program! A significant number of these, as shown in the test sample, may be equivalent to the original program. Determining equivalent mutants is a time consuming process and requires detailed knowledge of the computer program. It could be very difficult for an independent tester to determine if a program is equivalent as intimate knowledge of the details of the program are necessary. This limits the variety of personnel that can easily use the tool. Automated support for determining equivalent programs would significantly increase the usability of the tool.

<u>State-of-the-art user interface.</u> MOTHRA's current menu driven user interface, while adequate, should be more helpful in guiding users through the proper sequence of steps necessary to execute the tool. Determining the proper sequence to create the mutants, execute the program with/without mutants, compare the expected output with the mutant program's output, equivalence programs, etc. should be part of the tool's user interface. It should display the menus with those options applicable only at that point in time in the testing process. This would ensure that a user would not be allowed to, for example, perform equivalence functions without first creating mutants, and/or execute the mutant program without saving the correct output results. In addition, on-line help at each menu option should also be available to explain each menu option.

Test case generation. Automated support to provide additional test cases to kill the remaining mutants would also enhance the tool's usability. Currently, test case generation, is a manual process which is assisted by the results of various MOTHRA reports which display the type of live mutants, number of equivalenced mutants and dead mutants. From this information, the user must deduce which test case(s) may kill those particular mutants. Automating the generation of test cases would relieve some of the burden from the user. While 100% automation may not be possible, some support would increase overall productivity of the tester and allow the testing process to be completed in a shorter time frame. This enhancement is applicable not only to MOTHRA but to most all testing tools (including RXVP80) and research is currently being performed in this area by academia, industry, and government.