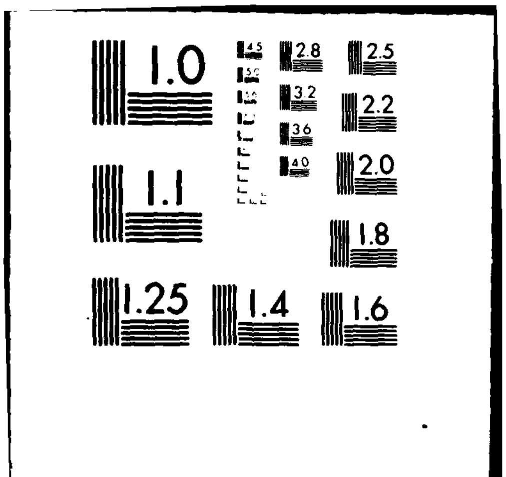
AD-A087 310 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
TESTING COBOL PROGRAMS BY MUTATION. VOLUME I. INTRODUCTION TO T--ETC(U)
FEB 80 J M HANKS N00014-79-C-0231

UNCLASSIFIED NL

1 OF 1
AD A
087310

END
DATE
FILMED
9-80
DTIC



MICROCOPY RESOLUTION TEST CHART

LEVEL II

ADA087310

School of

Information and Computer Science

GEORGIA INSTITUTE
OF TECHNOLOGY

80 6 19 017

DISCLAIMER NOTICE

(9) Master's thesis

(1)

(14) GIT-ICS-80/04-Vol-1

(6) TESTING COBOL PROGRAMS BY MUTATION*

Volume I. INTRODUCTION TO THE CMS.1 SYSTEM.

(10) JEANNE MARIE HANKS

(11) FEB[redacted]80

(12) 75

*This research was supported in part by The US Army Institute for Research in Management Information and Computer Science, ARO Grant No. DAAG29-78-G-0121 and The Office of Naval Research, Grant No. N00014-79-C-0231.

(15) DAAG29-78-G-0121

410044

JB

TESTING COBOL PROGRAMS BY MUTATION

A THESIS

Presented to

The Faculty of the Division of Graduate Studies

By

Jeanne Marie Hanks

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Information and Computer Science

Georgia Institute of Technology

February, 1980

ACKNOWLEDGEMENTS

I am grateful for the support of this thesis in part by The US Army Institute for Research in Management Information and Computer Science, ARO Grant No. DAAG29-78-G-0121 and The Office of Naval Research, Grant No. N00014-79-C-0231.

I would like to thank my thesis advisor, Dr. Richard A. DeMillo for providing continual support, thoughtful criticisms, and valuable suggestions for this thesis, and the members of my reading committee, Dr. Richard J. LeBlanc and Alton P. Jensen, for their helpful comments and suggestions.

I would also like to thank Allen T. Acree for his assistance in implementing the Cobol Mutation System.

I am also grateful to the graduate office for the waiver of certain format requirements so that this thesis could be generated on the PRIME-400 mini-computer.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figures

ABSTRACT

Testing Cobol Programs by Mutation

Jeanne M. Hanks

225 Pages

Directed by Dr. Richard. A. DeMillo

Program mutation is a testing technique which has been applied to Fortran programs[ABDLS]. This thesis will describe the application of mutation to the Cobol language in an automated program mutation system. The thesis will describe the development of a Cobol Mutation System (CMS.1), its testing using Fortran mutation analysis, and the subset of Cobol that is supported by CMS.1. The internal representation selected to represent the Cobol source statements and a description of the mutant operators that are implemented in CMS.1 will also be supplied.

CHAPTER I

INTRODUCTION

Program Testing

Methods of assuring program correctness can be divided into two different approaches: program proving techniques and program testing techniques. Program proving involves a formal proof that a program performs correctly [DLP]. This approach is currently ineffective because the proofs are generally hard to produce manually and are often incorrect or prove the wrong result [DLP].

The goals of program testing are to increase confidence that a program will perform as desired, to discover errors, and to provide some measure of performance. Various techniques have been proposed to reduce testing to a systematic methodology. These techniques include random generation, symbolic execution, and mutation analysis.

Random generation of test cases is easy to conceptualize and to implement but is rather inefficient [DLS1]. The number of test cases necessary to execute the 'normal' flow and the 'exception' flow in a program can become very large.

Symbolic execution of a program produces better test data than the random generation method. Variables are treated as algebraic unknowns and constraints are generated in terms of those unknowns to indicate those restrictions which data must satisfy if a certain path is to be executed. Symbolic execution generates data which executes every statement in the program and executes each branch.

Mutation analysis involves generating test data by any means that is available, then applying the technique to gain some measure of confidence of test "coverage". Through the mutation process a set of test data is generated that increases the confidence of a program's correctness.

During mutation a program is perturbed in simple ways which simulate typical programming errors. This process generates a variety of mutant programs. Given a set of test data that the programmer believes tests his program, the mutant programs are distinguished from the original program by their behaviour on the test data. Test data which is able to distinguish all non-equivalent mutants of a program must thoroughly exercise the program and, hence, provide strong evidence of the program's correctness [ABDLS].

Cobol Mutation

An automated system for Cobol Mutation Analysis (CMS.1) has been developed and implemented at Georgia Tech on a PRIME 400. CMS.1 has been derived from the Pilot Mutation System (PIMS or FMS.1) for Fortran program mutations which was designed at Yale University and has been implemented at

Yale, Georgia Tech and the University of California, Berkeley [BDLS]. CMS.1 has the added capability to handle I/O which is not currently available for Fortran.

CMS.1 is an interactive system that accepts as input a Cobol program and representative test data, which, when applied to the Cobol program, produces reference output that the programmer has verified to be correct. CMS.1 generates a large set of mutants of the Cobol program and executes these interpretively. The resultant outputs are compared to the reference output to identify (1) deficiencies in the test data, or (2) functionally equivalent versions of the program which are possibly more efficient. Through this interactive process, the user can become more confident of the program's correctness. For a detailed study of this aspect of mutation see [ABDLS, AA].

CMS.1 execution consists of five main phases: ENTRY, PRE-RUN, MUTATION, INTERPRETATION, and POST-RUN. Figure 1 shows interaction with CMS.1.

An input program, P, is parsed into an intermediate code. If any Cobol syntax errors exist in P, the errors are displayed at the user's console. When no syntax errors exist and the intermediate code has been created, mutant descriptors for the program are generated. Now the original program is executed interpretively on a set of test data supplied by the user. The results for the test data are shown to the user who verifies them as either acceptable or

MUTATION TEST METHODOLOGY

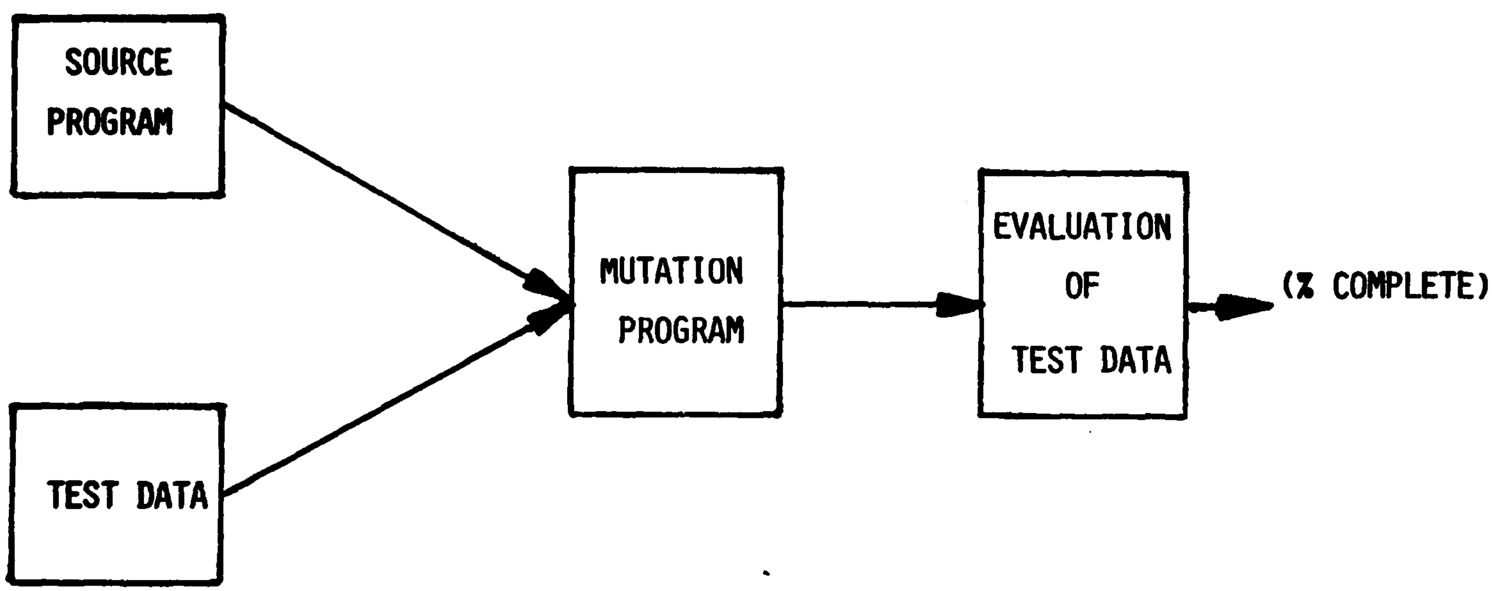


Figure 1 CMS Interaction

unacceptable. The user then has the opportunity to activate programs of some or all mutant types that have been generated for the original program. The results of the mutant runs are displayed for the user.

The ENTRY phase interacts with the user and initializes the system. The user gives the name of the program file to be tested. The internal files necessary for CMS.1 runs are created by using the program's file name and adding extensions to this name. For example, the files which are created are:

filename.MR	mutant record file
filename.MS	mutant status file
filename.LO	log file
filename.TD	test data file
filename.TS	test status file
filename.IF	internal form file

The ENTRY phase determines if the CMS.1 run is the initial run or a continuation of a previous run by checking whether these files exist or not. Even if this run is a continuation, the user is given an opportunity to restart the analysis. If the run is a continuation of a previous run, then the files are loaded. If this is a fresh run, then the program is parsed and its internal files are created (MEMORY, SYMBOL TABLE, CODE, and STATEMENT files). At this time, all the mutant records for the program are created.

The PRE-RUN phase interacts with the user to obtain test cases. These test cases may be contained in files so the user enters the name of the data file or the test data may be entered directly into CMS.1. When all the input test data has been given, the program is interpreted on the data. A copy of the test cases and the results are displayed for the user. The user is then asked to indicate if the test cases are acceptable or not. A test case is acceptable if it generates correct results. Any test cases which are marked as unacceptable are deleted from consideration.

The MUTATION phase gains control at this point. The user is given a list of the mutant types, for example scalar for scalar replacement, relational operator replacement, etc., that can be considered and asked which one he would like to activate. Once the mutant programs have been activated, they are executed interpretively on the test data that has been supplied. When mutants are being executed on a test case, only those mutants that affect a statement which is executed by the test case are interpreted. For an explanation of the mutants that are implemented in CMS.1, see the discussion on the MUTANT RECORD file.

The INTERPRETATION phase is invoked by the PRE-RUN phase to execute the original program on a set of test data with execution returning to the PRE-RUN phase. The INTERPRETATION phase is also invoked in a loop by the MUTATION phase for interpreting the mutants and obtaining results of the mutants.

The interpreter uses a program counter, PC, to control flow through the STATEMENT table. Some error checking is done by the interpreter; the errors that can be caught and their associated error codes are:

1 TRAP, execution beyond end-of-code, or SIZE ERROR without an exception handler.
2 TIMEOUT
more statements have been executed than is allowed.
3 DATA FAULT
incorrect mixing of numeric and alphanumeric data.
4 UNDEFINED
attempt to reference an undefined data item.
5 I/O FAULT IN OPEN/CLOSE
attempt to open a file that is already opened or attempt to close a file that is not opened.
6 ATTEMPT TO READ PAST EOF
7 OVERWRITE OR OVERREAD, COMPARED TO ORIGINAL PROGRAM
this error is detected when a mutant program tries to read or write more data than the original program did.
8 OUTPUT FILE TOO LARGE TO FIT IN BUFFER
the programs output exceeds the limits of the CMS.1 system.
9 ARRAY ELEMENT OUT OF BOUNDS
10 INCORRECT OUTPUT
output of mutant program differs from that of the original test program.
11 ILLEGAL CODE IN INTERNAL FORM
incorrect internal code has been generated by the system.

A variable is used to communicate errors to the PRE-RUN phase and the MUTATION phase.

When the interpreter executes a mutant, the results are compared with the original program output as it is generated (i.e., when a write statement is executed). If the two outputs are not the same, then interpretation is halted and an error code is reported to the MUTATION phase so that mutant will be marked as 'killed'. The main structure of the interpreter is based on a Fortran COMPUTED GO TO statement.

For a detailed discussion of the interpreter, see the documentation for SUBROUTINE INTERP.

After all the mutant programs have been executed, the results are displayed for the user during the POST-RUN phase. The user may see the live mutants, mark mutants as equivalent, turn previously marked equivalent mutants back on, stop the run, or loop back to the beginning of the run where more test cases may be entered.

If the mutants are to be seen, it is necessary to first 'decompile' the internal code for the statement into a recognizable Cobol statement. This 'decompiling' is accomplished by examining the internal form for a source statement and reconstructing its structure using the HASH TABLE for the printable names of variables.

Plan of Presentation

The purpose of this thesis is not to justify mutation analysis as a testing tool but to describe the implementation of a mutation system for Cobol. The Cobol system is written in Fortran and several major routines have been tested on the Fortran mutation system. A detailed discussion of the Cobol Mutation System (CMS.1) is given in Chapter II which includes a description of the subset of the Cobol language supported by CMS.1, a description of the file structures, and a description of the mutant operators implemented in CMS.1. Chapter III contains a sample run on the CMS.1 system and a discussion of testing CMS.1 routines on the Fortran mutation system, FMS.2. The conclusion is in Chapter IV which contains suggestions for improving the Cobol mutation system. Appendix A contains a Cobol tutorial of the Cobol subset suppported by CMS.1 and Appendix B contains detailed documentation on each routine in the CMS.1 system.

CHAPTER II

COBOL MUTATION SYSTEM (CMS.1)

Cobol Subset and Intermediate Code

The level of Cobol which can be accepted by CMS.1 is referred to as level 1 Cobol. The Cobol source program must be in the standard Cobol format with columns 1-6 containing the sequence number (which is ignored by CMS.1); column 7 is either blank or contains a hyphen for the continuation of a non-numeric literal or contains an asterisk for a comment line; information beyond column 72 is ignored [A]. A list of acceptable Cobol verbs follows. For each verb the format for the internal form generated by the parser for use by the interpreter is given. A detailed Cobol tutorial is given in Appendix A.

MOVE

MOVE {data name-1 | literal} TO data-name-2
[data-name-3] ...

The internal form:

<MOV><n><source><dest-1>...<dest-n>

ADD

ADD {data-1 | literal-1} [data-2 | literal-2] ... TO
data-m [ROUNDED] [ON SIZE ERROR imperative-statement]

The internal form:

<ADD><rnd><size><n><op-1>...<op-n>

The rnd field specifies whether to round the result or not. The size field indicates if a size error clause was given or not.

ADD GIVING

ADD {data-1 | literal-1} {data-2 | literal-2}
[data-3 | literal-3]... GIVING data-m [ROUNDED]
[ON SIZE ERROR imperative statement]

The internal form:

<ADG><rnd><size><n><op-1>...<op-n><dest>

SUBTRACT

SUBTRACT {data-1 | literal-1} [data-2 | literal-2]...
FROM data-m [ROUNDED] [ON SIZE ERROR imperative
statement]

The internal form:

<SU><rnd><size><n><op-1>...<op-n>

SUBTRACT GIVING

SUBTRACT {data-1 | literal-1} [data-2 | literal-2]...
FROM {data-m | literal-m} GIVING data-n [ROUNDED]
[ON SIZE ERROR imperative-statement]

The internal form:

<SUG><rnd><size><n><op-1>...<op-n><dest>

MULTIPLY

MULTIPLY {data-1 | literal-1} BY data-2 [ROUNDED]

[ON SIZE ERROR imperative statement]

The internal form:

<MUL><rnd><size><op-1><op-2>

MULTIPLY GIVING

MULTIPLY {data-1 | literal-1} BY {data-2 | literal-2}

GIVING data-3 [ROUNDED] [ON SIZE ERROR

imperative-statement]

The internal form:

<MUG><rnd><size><op-1><op-2><dest>

DIVIDE

DIVIDE {data-1 | literal-1} INTO data-2 [ROUNDED]

[ON SIZE ERROR imperative-statement]

The internal form:

<DIV><rnd><size><op-1><op-2>

DIVIDE GIVING

DIVIDE {data-1 | literal-1} { INTO | BY }

{data-2 | literal-2} GIVING data-3 [ROUNDED]

[ON SIZE ERROR imperative-statement]

The internal form:

<DIV><rnd><size><op-1><op-2><dest>

For the internal form, the parser codes both the BY and INTO options in the form of the INTO. CMS.1 will accept both forms of the DIVIDE GIVING statement.

COMPUTE

COMPUTE data-1 [ROUNDED] = {data-2 |

literal-1 | arithmetic-expression}

[ON SIZE ERROR imperative-statement]

The internal form:

<COM><rnd><size><ident><arithmetic expression>

Where ident refers to the data item that receives the result of the compute.

GO TO

GO TO procedure-name

The internal form:

<GO><procedure>

GO TO ... DEPENDING

GO TO procedure-name-1 [procedure-name-2]...

DEPENDING on data-name

The internal form:

<GOD><n><ident><proc-1>...<proc-n>

PERFORM

PERFORM procedure-name-1 [THRU procedure-name-2]

The internal form

<PEV><proc-1><proc-2>

PERFORM-VARYING

PERFORM procedure-name-1 [THRU procedure-name-2]

VARYING data-name-1 FROM {literal-2 | data-name-2}

BY {literal-3 | data-name-3} UNTIL condition-1

The internal form:

<PEV><proc-1><proc-2><id><low><high><inc>

<REP1><low><high><inc><start><stop>

The REP1 operation for the internal form of a PERFORM VARYING statement is an internal operation to aid in the repeating of the procedures. After the procedures are executed the control passes to this statement where the condition can be tested for its truth to determine if the procedures should be executed again or if the PERFORM VARYING is to be terminated.

PERFORM TIMES

PERFORM procedure-name-1 [THRU procedure-name-2]

{data-name-1 | integer-1} TIMES

The internal form:

<PET><proc-1><proc-2><ident>

<REP2><count><start><stop>

As in the PERFORM VARYING, it was necessary to implement another internal operation for the PERFORM TIMES to determine how many times the procedures have been executed. The count field is decremented each time the procedures are executed until it is zero and the PERFORM TIMES is completely executed. The start field is a pointer to the first statement in the procedures being PERFORMed and the stop field contains the statement number of the last statement being PERFORMed.

PERFORM UNTIL

PERFORM procedure-name-1 [THRU procedure-name-2]

UNTIL condition-1

The internal form:

<PEU><proc-1><proc-2><logical expression>

IF

IF condition {statement-1 | NEXT SENTENCE }

[{ ELSE } {statement-2 | NEXT SENTENCE }]

The internal form:

<IF><ELSE-statement pointer><logical expression>

OPEN

OPEN INPUT [file-name] ...

OPEN OUTPUT [file-name] ...

The internal form:

<OPEN><1 | 2 | ... | 20>

Where 1 thru 10 reference one of the ten input files and 11 thru 20 reference one of the ten output files.

CLOSE

CLOSE file-name-1 [filename-2] ...

The internal format:

<CLOSE><1 | 2 | ... | 20>

READ

READ file-name RECORD [INTO data-name]

AT END imperative-statement

The internal form:

<READ><1 | 2 | ... | 10><into-ident>

WRITE

WRITE record-name [FROM data-name]

The internal form of this statement is:

<WRITE><11 | 12 | ... | 20><from-ident><advance>

STOP

STOP RUN

The internal form:

<STOP>

There are two operations that are coded by the parser for use in the CMS.1 system. These two operations are not supported in the Cobol subset and will not be compiled by the parser. These two operations are the RETURN and the NO-OP. These operations are needed to implement the PERFORM verbs; when executing a PERFORM, it is necessary to return program control to the statement following the PERFORM statement after the last statement of the paragraph range has been executed. To make this feasible, the parser inserts a NO-OP at the end of each paragraph and the interpreter changes the NO-OP into a RETURN, if a PERFORM is being executed.

File Structures

There are several files the system produces in order to store information from one run to the next. These are shown in Figure 2, which also outlines the major functions of each phase. The major functions are:

The internal form file stores the parsed version of the program.

The test data file stores for each test case, the test data input and the results of execution of that test data.

The mutants information file keeps the mutant descriptor records plus various other counts on what types of mutants have been produced.

For a more detailed discussion see [BDLS].

INTERNAL REPRESENTATION

The 'INTERNAL FORM' of a program consists of the STATEMENT table, CODE array, SYMBOL TABLE, MEMORY array, and HASH TABLE. The INTERNAL FORM file contains, in addition to

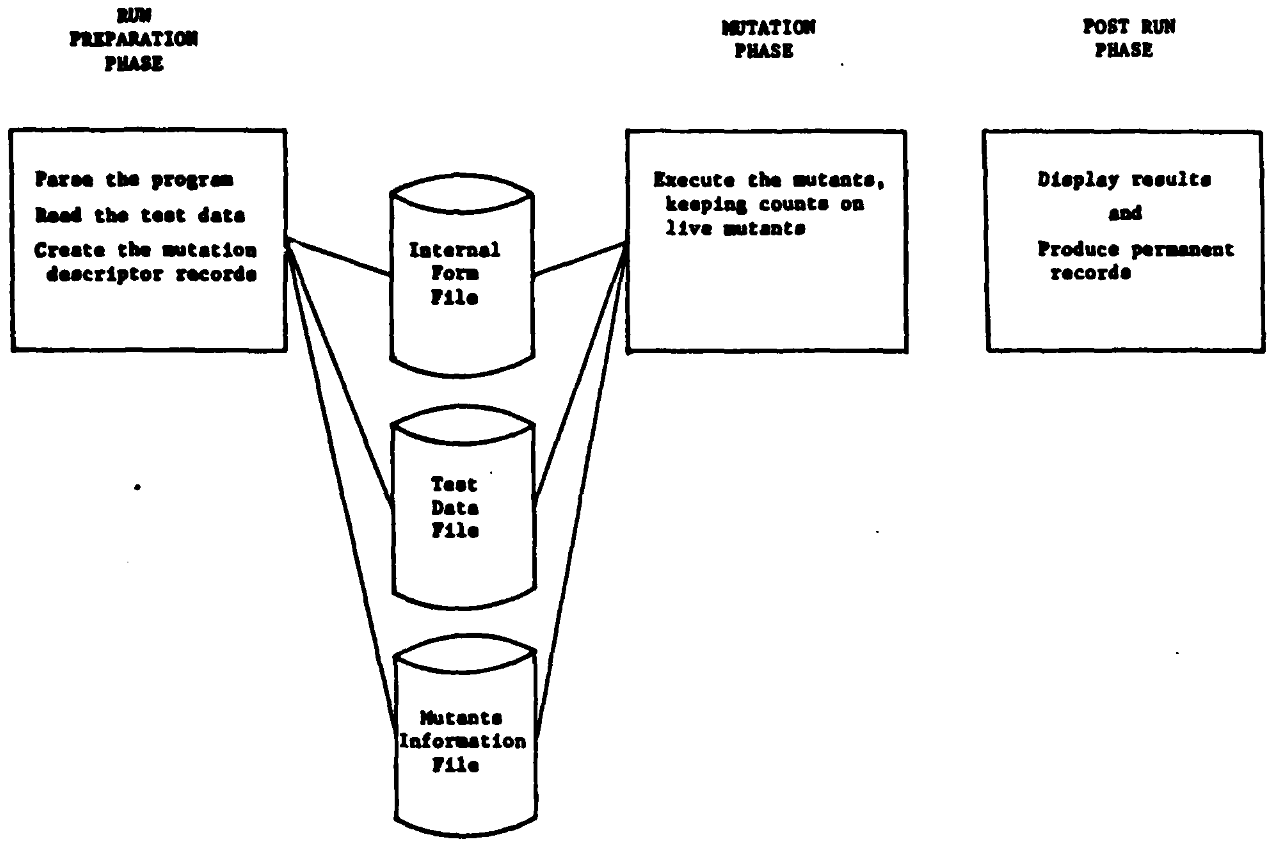


Figure 2 CMS File Layout

these files, the sizes for each of the files. These sizes are stored in the beginning of the INTERNAL FORM file; followed by the STATEMENT, CODE, SYMBOL, MEMORY, and HASH files. The INTERNAL FORM files are created when the program is parsed. Due to the nature of the CMS.1 system there are several levels of indirection which have been incorporated in order to maintain all the information that is necessary for mutation.

Every entry in the STATEMENT table references its code in the CODE array which contains references to the SYMBOL TABLE for variables, literals, paragraph-names, etc. and finally the SYMBOL TABLE contains references to memory locations for variables and literals in the literal pool contained in low memory. The SYMBOL TABLE also contains references to the HASH TABLE for a variable's name; this is usually used for 'decompiling' a statement.

STATEMENT TABLE FILE

The STATEMENT table contains an entry for each executable statement contained in the program. The file contains records of three elements each with the following format:

Position Use

1 - reference to the code array for the statement

2 - line number of the associated source listing

3 - statement level

0 - continuation of a statement

1 - beginning a new statement

2 - n - depth in a conditional statement

CODE ARRAY

The CODE file is a sequential array that contains the intermediate code for each Cobol statement; it also contains an element giving the length of the code for a statement. The length of each Cobol statement varies depending on the operation and the length can even vary for the same type of operation.

This organization of the CODE array allows for easy implementation of mutant descriptors. It is not necessary to alter the internal code for the original source statement because the internal code for the mutated statement can be appended to the end of the code array. The statement table entry for the code reference can be changed to reference the mutated statement. Cleaning up after a mutant program is executed is accomplished by changing the statement table reference back to the internal form of the source statement.

SYMBOL TABLE

The SYMBOL TABLE has been designed to contain important information that must be obtained at run time. It is thus necessary for the SYMBOL TABLE to be resident in core during execution.

The first record in the SYMBOL TABLE is for the name of the Cobol program. The next 20 records are reserved for the INPUT and OUTPUT files that can be used in the Cobol program (CMS.1 allows up to 10 input files and 10 output files). The reserved words ZERO and BLANK are used so widely in most Cobol programs that we have included these variables automatically in the SYMBOL TABLE and in MEMORY. The rest of the SYMBOL TABLE is created by the parser. Items are entered into the SYMBOL TABLE as they are encountered in the program. The HASH TABLE is used to determine if a data item has been entered previously or not. All the data items defined in a Cobol DATA DIVISION are entered in the same order as they are encountered. After the DATA DIVISION is parsed, the PROCEDURE DIVISION is parsed and the PARAGRAPH-NAMES and literals are entered into the SYMBOL TABLE. The SYMBOL TABLE file is an array containing 10 elements with the following information for each record:

Position Use

1 - pointer to the hash table for the printable name, this is used for 'decompiling' a statement.

2 - type
   1 - unsigned numeric
   2 - signed numeric
   3 - alphanumeric
   4 - edited
   5 - group item
   6 - continuation of a table item
   7 - numeric literal

8 - alphanumeric literal
9 - paragraph name

3 - level number, or
beginning statement number if this is a paragraph name entry

4 - number of digits for a numeric item, or,
memory location for a PICTURE item, or,
ending statement number if this is a paragraph name entry, or
multiplier for the first subscript if this is a table item entry

5 - memory location, or
multiplier for the second subscript if this is a table item entry

6 - length of the item in memory, or
maximum allowed subscript for the first subscript if this is a table item entry

7 - table level
0 - scalar
1 - one level table, row item
2 - two level table
or, the maximum allowed subscript for the second subscript if this is the second row of information table item entry

8 - pointer to the value string in the literal pool, if a VALUE clause was specified, or
the number of occurrences in the second row for a table item entry

9 - SYMBOL TABLE entry for a redefined item

10 - Number of the source statement for the data-item entry.

MEMORY

The MEMORY array contains the Cobol source program's memory and the storage areas necessary to execute a Cobol program. MEMORY is a sequential single-dimension array. The first thirty elements are reserved for the interpreter's working storage. The literal pool follows the working storage. This literal pool contains the PICTURE specifications for the edited data items, constants used in the Cobol VALUE clauses, and any literal constants used in the program. The Cobol variables ZERO and BLANK are the first two items in the literal pool. A variable is kept in COMMON to contain the location of the end of the literal pool. The working memory follows the literal pool and consists of character data. PICTURE items are the only items that do not have all their auxilliary information contained in the SYMBOL TABLE because they need more information than can be contained in one record; therefore, three extra words are stored in memory with a PICTURE's description. The structure of a picture item follows:

Position	Use
1 -	Picture length
2 -	Number of digits in the picture
3 -	Number of decimal digits in the picture
4 -	The actual picture description

HASH TABLE

The HASH TABLE is used to hold the printable names of Cobol reserved words, program file names, and variables. The HASH TABLE contains the names for all the RESERVED words, the Cobol program name, the Cobol input and output file names, the variables, and the paragraph names. Each record in this file contains 17 words. The first 15 words are used to store the name with 2 characters per word; Cobol allows a maximum of 30 characters per name. The 16th word contains the number of characters actually used in the name and the 17th word is the location in the SYMBOL TABLE for this item. The record layout is as follows:

Position	Use
1 thru 15	- Print name with two characters per word.
16	- Number of characters in the print name.
17	- SYMBOL TABLE location for the item.

TEST STATUS FILE

The TEST STATUS file contains status information for each test case that is accepted for CMS.1. Each record of the file contains 42 words of information. The first record of the file indicates which of the twenty allowable Cobol files are used, how many test cases have been defined, how many test cases have been defined during the current run, and the next available location in the file for appending information for the next test case to be defined. Note that INPUT0 is the first Cobol input file and that INPUT9 is the tenth input file and similarly for the output files. The format of the first record in the TEST STATUS file is as follows:

Position Use

1 - Indicates if INPUT0 has been used or not.
    0 - not used
    1 - used

2 - Indicates if INPUT1 has been used or not.
    0 - not used
    1 - used

3-20 - Similar to the above for INPUT2 thru INPUT9 and for OUTPUT0 thru OUTPUT9.

21 - Total number of test cases.

22 - The number of test cases previously defined.

23 - Location of the next available record in this file for appending information for a new test case.

After the first record for file information, there are two records for each test case. The first contains information about the number of records in each Cobol input and output file and the starting position for the file in the TEST DATA file. The format of the TEST STATUS file is:

Position	Use
1	The starting position in the TEST DATA file for INPUT0.
2	The number of records in INPUT0.
3	The starting position in the TEST DATA file for INPUT1.
4	The number of records in INPUT1.
5-40	Similar to the above for the other INPUT and OUTPUT files.
41	The number of statements in the original program that are executed by this test case.

The second record for a test case is a bit map which indicates which statements are executed by the test case. The first bit of each word is not used so there are 15 usable bits per word. There are 42 records which give a maximum of 42 x 15 = 630 bits per record. For Cobol programs with more than 630 procedure division statements, either the record size for the TEST STATUS file will have to be increased or more than 2 records per test case will have to be used.

TEST DATA file

The TEST DATA file is a sequential file that contains the input and output for each test case. The Cobol input for all the input files used by the original source program is stored in sequential order in the TEST DATA file followed by the output files generated by the original source program for a test case. The information for additional test cases is stored in the same manner following the existing data. The data is stored in a packed format by SUBROUTINE PACK. This packed format contains a character followed by a count of how many exist together; if a character is not repeated in the file, then it has no repeat count associated with it. An initial segment of the ASCII codes represent unprintable characters. Values in this initial segment are treated as repeat counts. The subroutine PACK breaks up long repeat strings in order to keep repeat counts within bounds. (Note: For portability, EBCDIC also has an initial segment of nonprintable characters). The reason this packing is done with repeat counts is to save storage space. The character and its count are stored in half-words of one byte each.

MUTANT STATUS FILE

The MUTANT STATUS file is created by appending .MS to the filename of the program being tested. This file contains status information about the mutants concerning the mutant types that have been turned on; the number of mutants created for each mutant type; a pointer for each mutant type to the first record in the MUTANT RECORD file; the number of 'live' and 'dead' mutants; and the number of mutants that are 'killed' by each of the eleven errors detected by the interpreter.

The first record on the MUTANT STATUS file contains a count of the total number of mutants created. This count is in the first word of the 16 word record. The next several records contain header information for each mutant type. The header uses four words to store its information; the header format is:

Psition Use

1 - Mutant type.
2 - On or off
    0 - off
    1 - on
3 - On or off this run
    0 - off
    1 - on
4 - Location in the MUTANT STATUS file for the status block.

The MUTANT STATUS file has 16 words per record. This means that four headers can be placed in one record; since there are currently 25 mutant types, 5 records are needed to store the header information. The information contained in the header blocks is resident in core during the CMS.1 run.

For each mutant type there is one record that contains the mutant status information. The information and the format for a mutant status record is:

Position	Use
1	Number of mutants for this type.
2	Number of words for the bit map.
3	MUTANT RECORD file location for the first mutant record of this type.
4	Number of live mutants.
5	Number of dead mutants.
6	Number killed by trap, attempt to execute beyond the end of code by the STOP statement being deleted or no SIZE ERROR clause given and a size error occurs.
7	Number killed by time-out.
8	Number killed by data fault.
9	Number killed by initialization fault.
10	Number killed by I/O fault in OPEN or CLOSE.
11	Number killed by attempt to read past end-of-file.
12	Number killed by writing more than original program.
13	Number killed by output too large for the buffer.
14	Number killed by array subscripts out-of-bounds.
15	Number killed by incorrect output.
16	Number killed by garbage in the CODE array.

Following these records are the bit maps for the live, dead, and equivalent mutants, where there is one bit for each mutant. In all of the bit maps the first bit (sign bit) of each word is not used. The bit maps are of varying length depending on the program and on the mutant operators. The number of records needed for a bit map is rounded up to the nearest whole-record size. There are four words per record with 15 usable bits per word, thus, there are 60 bits per record and the number of records necessary to store a bit map is the next largest integer greater than the number of mutant divided by 60 bits per record.

MUTATION RECORD FILE

The particular mutant types that have been implemented in CMS.1 are data, input/output, control structure, and procedural mutations. Data mutations alter the data descriptions contained in the SYMBOL TABLE. INPUT/OUTPUT mutations deal with changing a file reference in one read or write statement. For example, an input file may be exchanged for another input file in a read statement, or an output file exchanged with another output file in a write statement; but an input may not be exchanged with an output. Control structure mutations alter statements that deal with program flow. Procedural mutations are those mutations which are applied to procedure division statements.

The mutant record file consists of n records, where n is the number of mutants created for the program. Each record is 4 integers long. All the mutant descriptors for a mutant type are stored contiguously in the mutant record file. The first word is the mutant type and the other three contain information for that mutant type. A mutant record exists for each mutant that can be applied to a program. The following is a list of the mutants and their descriptor records (an x in any field means that that field is not used). All mutants that alter a statement are copied at the end of the code array and the code reference in the statement table is changed to refer to this mutant statement. To restore the internal form after implementing a statement mutant we only need to change the statement table code reference to refer back to the original statement. Data mutations alter the data descriptions to the original statement. The following is an explanation of the mutant types that are implemented in CMS.1.

1 Decimal alterations move implied decimal in numeric items one place to the left or right, if possible.

<DEC><SYMBOL TABLE location><+1 | -1><x>

Where +1 - add 1 digit to the fraction part,

-1 - subtract 1 digit from the fraction part.

2 Reverse occurs clauses reverses the row and column size in a two-level table.

<REVERSE-OCCURS><SYMBOL TABLE location><x>

<SYMBOL TABLE 2>

3 Alter occurs clause changes the dimension of a one- or two-level table by adding or subtracting 1 from the dimension.

<ALTER-OCCURS><SYMBOL TABLE location><code><x>

where code = 0 means "add 1 to occurs",

= 1 means "subtract 1 from occurs".

4 Insert a filler (PIC X) in a record. This mutation is aided by the fact that the parser inserts a dummy record between each data item in the symbol table; this was done so that the references in the code array to the SYMBOL TABLE will not be affected by implementing this mutant.

<INSERT><SYMBOL TABLE location><x><x>

5 Change a filler's size by adding or subtracting 1 to its size.

<CHANGE-FILLER><SYMBOL TABLE location><+1 | -1><x>

6 Reverse adjacent elementary items in a record. This is accomplished by reversing the memory pointer contained in the SYMBOL TABLE.

<REVERSE><SYMBOL TABLE location>

<next elementary location><x>

7 Input/Output reverses two file reference4s for input files or for output files.

<FILE><statement><x><new file-code>

8 DELETE mutant deletes a statement from the program by making it a NO-OP. This mutation checks for the necessity of a statement.

<DELETE><statement><x><x>

9 GO-TO changed to a PERFORM statement is implemented by

changing the opcode.

<GO-PERFORM><statement><x><x>

10 PERFORM changed to a GO TO.

<PERFORM-GO TO><statement><x><x>

11 THEN-ELSE clause reversal is implemented by negating the condition. A special opcode, NIFOP, was created for implementing this mutant.

<THEN-ELSE><statement><x><x>

12 STOP replacement mutation consists of changing a statement to a STOP statement to verify the necessity of a statement's existence.

<STOP><statement><x><x>

13 THRU clause adjustment extends the range of a PERFORM statement. <THRU><statement><new paragraph><x>

14 TRAP statement mutation consists of inserting a TRAP statement into the program after possible transfer points for path analysis.

<TRAP><statement><x><x>

15 ARITHMETIC OPERATION SUBSTITUTION changes one arithmetic verb for another. For example, change ADD to SUBTRACT.

<ARITHMETIC-1><statement><new operation><x>

16 COMPUTE OPERATION SUBSTITUTION exchanges an operand in a compute statement.

<ARITHMETIC-2><statement><field><new operation>

where 'field' is the relative location in the code description of the operator to be changed.

17 PARAMETER ALTERATION is used in COMPUTE statements to change the position of a parenthesis by moving one parenthesis one place to the left or right.

<PAREN><statement><from-field><to-field>

where the 'from-field' is the relative location in the code description of the parenthesis in the COMPUTE statement being altered and the 'to-field' is the location to which the parenthesis is to be moved.

18 ROUND mutation turns the 'rounded' condition on or off in an arithmetic statement; ROUNDED is changed to truncation and vice versa.

<ROUND><statement><x><x>

19 MOVE mutation reverses the direction of the MOVE operation when only two fields are used, if such a reverse would be a legal Cobol statement. For example,

MOVE DATA-1 TO DATA-2. changed to

MOVE DATA-2 TO DATA-1.

<MOVE><statement><x><x>

20 LOGICAL OPERATOR REPLACEMENT is implemented by changing a logical operator to a different logical operator.

<LOGIC><statement><field><new value>

where 'field' is a relative location in the code description for the logical operator being altered.

21 SCALAR for SCALAR replacement changes the reference from one scalar to another scalar in a statement.

<SCALAR-SCALAR><statement><field>

<new SYMBOL TABLE location>

where 'field' is the relative location in code description.

22 CONSTANT for CONSTANT replacement replaces one reference to a constant with another constant reference.

<CONSTANT-CONSTANT><statement><field><new location>

23 CONSTANT for SCALAR replacement.

<CONSTANT-SCALAR><statement><field><new location>

24 SCALAR for CONSTANT replacement.

<SCALAR-CONSTANT><statement><field><new location>

25 CHANGE CONSTANT mutant is used to change a numeric constant by +1%, -1%, +1, or -1 whichever is largest. To ease the implementation of this mutant, 'mutant' values for each numeric constant have been inserted in the SYMBOL TABLE right after the constant has been inserted during the parse.

<CHANGE-NUMERIC-CONSTANT><statement><field>
<new location>

LOG FILE

The LOG FILE is used to contain important information about a CMS.1 session. This file is a sequential file which can have some of its contents determined by the user (e.g. by issuing an OUTPUT command). The CMS.1 system automatically stores some information in the LOG file. During the PRE-RUN phase a copy of the Cobol source program is placed in the file. For each test case, the input file is stored with the result for that test case; the results are TEST CASE FAILED, TEST CASE REJECTED, and TEST CASE # ENTER AND ACCEPTED. During the MUTATION phase a list of the mutant types that are currently enabled is stored in the LOG file. The POST-RUN phase stores the status information for the pass. If the user marks any mutants equivalent, then the number marked is stored in the file. The user may have a list of the live mutants stored in the file or a list of the test cases stored by specifying the OUTPUT command. If the user aborts the run by issuing a KILL command, then he is asked to enter a message explaining the reason for aborting the run. This message is terminated by a control-C and is placed in the LOG file.

CHAPTER III

EXPERIENCE

Cobol Example

The following is a script of a CMS.1 run on a program originally from the Army SIDPERS personnel system. The program has been modified somewhat, mainly in the reduction of the record sizes to make a better CRT display. The program takes as input two files, representing an old backup tape and a new one. The output is a summary of the changes. The input files are assumed to be sorted on a key field. The program has 1195 mutants, of which 21 are easily seen to be equivalent to the original program. Initially ten test cases were generated to eliminate all of the nonequivalent mutants. Subsequently a subset of five test cases was found to be adequate for the task. The entire run took about 10 minutes of clock time, and 2 minutes and 13 seconds of CPU time on the PRIME 400.

The following is an example of the CMS.1 run. User input has been entered in lower case to distinguish it from the system instructions and prompts. This is the interaction at the pre run phase where the user has requested the program being tested be displayed at the user's console.

```
WELCOME TO THE COBOL PILOT MUTATION SYSTEM
PLEASE ENTER THE NAME OF THE COBOL PROGRAM FILE:>log-changes
DO YOU WANT TO PURGE WORKING FILES FOR A FRESH RUN ?>yes
PARSING PROGRAM
SAVING INTERNAL FORM
WHAT PERCENTAGE OF THE SUBSTITUTION MUTANTS DO YOU WANT TO CREATE?>100
CREATING MUTANT DESCRIPTOR RECORDS
PRE-RUN PHASE
DO YOU WANT TO SUBMIT A TEST CASE ? >program

 PROGRAM LAST COMPILED ON 1 11 80.

       IDENTIFICATION DIVISION.
       PROGRAM-ID. POBAACA.
       AUTHOR. CPT R W MOREHEAD.
       INSTALLATION. HQS USACSC.
       DATE-WRITTEN. OCT 1973.
       REMARKS.
           THIS PROGRAM PRINTS OUT A LIST OF CHANGES IN THE ETF.
           ALL ETF CHANGES WERE PROCESSED PRIOR TO THIS PROGRAM.  THE
           OLD ETF AND THE NEW ETF ARE THE INPUTS.  BUT THERE IS NO
           FURTHER PROCESSING OF THE ETF HERE.  THE ONLY OUTPUT IS A
           LISTING OF THE ADDS, CHANGES, AND DELETES.  THIS PROGRAM IS
           FOR HQ USE ONLY AND HAS NO APPLICATION IN THE FIELD.
            *****************
            MODIFIED FOR TESTING UNDER CPMS BY ALLEN ACREE
            JULY, 1979.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. PRIME.
       OBJECT-COMPUTER. PRIME.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT OLD-ETF ASSIGN INPUT1.
           SELECT NEW-ETF ASSIGN INPUT2.
           SELECT PRNTR ASSIGN TO OUTPUT1.
       DATA DIVISION.
       FILE SECTION.
       FD  OLD-ETF
           RECORD CONTAINS 80 CHARACTERS
           LABEL RECORDS ARE STANDARD
           DATA RECORD IS OLD-REC.
       01  OLD-REC.
           03  FILLER                            PIC X.
           03  OLD-KEY                           PIC X(12).
           03  FILLER                            PIC X(67).
       FD  NEW-ETF
           RECORD CONTAINS 80 CHARACTERS
           LABEL RECORDS ARE STANDARD
           DATA RECORD IS NEW-REC.
       01  NEW-REC.
           03  FILLER                            PIC X.
           03  NEW-KEY                           PIC X(12).
           03  FILLER                            PIC X(67).
       FD  PRNTR
           RECORD CONTAINS 43 CHARACTERS
           LABEL RECORDS ARE OMITTED
           DATA RECORD IS PRNT-LINE.
       01  PRNT-LINE                             PIC X(43).
       WORKING-STORAGE SECTION.
       01  PRNT-WORK-AREA.
           03  LINE1                             PIC X(30).
           03  LINE2                             PIC X(30).
           03  LINE3                             PIC X(20).
       01  PRNT-OUT-OLD.
           03  WS-LN-1.
               05  FILLER                        PIC X VALUE SPACE.
               05  FILLER                        PIC XXXX VALUE 'O   '.
               05  LN1                           PIC X(30).
               05  FILLER                        PIC XXX VALUE SPACES.
           03  WS-LN-2.
               05  FILLER                        PIC X VALUE SPACE.
               05  FILLER                        PIC XXXX VALUE 'L   '.
               05  LN2                           PIC X(30).
               05  FILLER                        PIC XXX VALUE SPACES.
           03  WS-LN-3.
               05  FILLER                        PIC X VALUE SPACE.
               05  FILLER                        PIC XXXX VALUE 'D   '.
               05  LN3                           PIC X(20).
               05  FILLER                        PIC XXX  VALUE SPACE.
```

```
 01  PRNT-NEW-OUT.
     03  NEW-LN-1.
         05  FILLER                       PIC XXXXX VALUE ' N    '.
         05  N-LN1                        PIC X(30).
         05  FILLER                       PIC XXX VALUE SPACE.
     03  NEW-LN-2.
         05  FILLER                       PIC XXXXX VALUE ' E    '.
         05  N-LN2                        PIC X(30).
         05  FILLER                       PIC XXX VALUE SPACES.
     03  NEW-LN-3.
         05  FILLER                       PIC XXXXX VALUE ' W    '.
         05  N-LN3                        PIC X(20).
         05  FILLER                       PIC XXX VALUE SPACES.
 PROCEDURE DIVISION.
 0100-OPENS.
     OPEN INPUT OLD-ETF NEW-ETF.
     OPEN OUTPUT PRNTR.
 0110-OLD-READ.
     READ OLD-ETF AT END GO TO 0160-OLD-EOF.
 0120-NEW-READ.
     READ NEW-ETF AT END GO TO 0170-NEW-EOF.
 0130-COMPARES.
     IF OLD-KEY = NEW-KEY
         NEXT SENTENCE
     ELSE GO TO 0140-CK-ADD-DEL.
     IF OLD-REC = NEW-REC
         GO TO 0110-OLD-READ.
     MOVE OLD-REC TO PRNT-WORK-AREA.
     PERFORM 0210-OLD-WRT THRU 0210-EXIT.
     MOVE NEW-REC TO PRNT-WORK-AREA.
     PERFORM 0200-NW-WRT THRU 0200-EXIT.
     GO TO 0110-OLD-READ.
 0140-CK-ADD-DEL.
     IF OLD-KEY > NEW-KEY
         MOVE NEW-REC TO PRNT-WORK-AREA
         PERFORM 0200-NW-WRT THRU 0200-EXIT
         GO TO 0120-NEW-READ
     ELSE GO TO 0150-CK-ADD-DEL.
 0150-CK-ADD-DEL.
     MOVE OLD-REC TO PRNT-WORK-AREA.
     PERFORM 0210-OLD-WRT THRU 0210-EXIT.
     READ OLD-ETF AT END
         MOVE NEW-REC TO PRNT-WORK-AREA
         PERFORM 0200-NW-WRT THRU 0200-EXIT
         GO TO 0160-OLD-EOF.
     GO TO 0130-COMPARES.
 0160-OLD-EOF.
     READ NEW-ETF AT END GO TO 0180-EOJ.
     MOVE NEW-REC TO PRNT-WORK-AREA.
     PERFORM 0200-NW-WRT THRU 0200-EXIT.
     GO TO 0160-OLD-EOF.
 0170-NEW-EOF.
     MOVE OLD-REC TO PRNT-WORK-AREA.
     PERFORM 0210-OLD-WRT THRU 0210-EXIT.
     READ OLD-ETF AT END GO TO 0180-EOJ.
     GO TO 0170-NEW-EOF.
 0180-EOJ.
     CLOSE OLD-ETF NEW-ETF PRNTR.
     STOP RUN.
 0200-NW-WRT.
     MOVE LINE1 TO N-LN1.
     MOVE LINE2 TO N-LN2.
     MOVE LINE3 TO N-LN3.
     WRITE PRNT-LINE FROM NEW-LN-1 AFTER ADVANCING 2.
     WRITE PRNT-LINE FROM NEW-LN-2 AFTER ADVANCING 1.
     WRITE PRNT-LINE FROM NEW-LN-3 AFTER ADVANCING 1.
 0200-EXIT.
     EXIT.
 0210-OLD-WRT.
     MOVE LINE1 TO LN1.
     MOVE LINE2 TO LN2.
     MOVE LINE3 TO LN3.
     WRITE PRNT-LINE FROM WS-LN-1 AFTER ADVANCING 2.
     WRITE PRNT-LINE FROM WS-LN-2 AFTER ADVANCING 1.
     WRITE PRNT-LINE FROM WS-LN-3 AFTER ADVANCING 1.
 0210-EXIT.
     EXIT.
>yes
```

Now, the user is asked to enter a test case. CMS.1 asks for each Cobol input file by name. The input files and the results are displayed for the user at his console. A test case for this program is a pair of input files. In CMS.1 these may be created outside the system and referenced by file name, or may be entered "on the fly" during the session.

```
 WHERE IS OLD-ETF?
>lc9
 WHERE IS NEW-ETF?
>lc5
 OLD-ETF PROVIDED TO THE PROGRAM

 I123456789012IIIIIIIIIIOJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNNBBB3339399GGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM

 I133456789012000000000000000000000000000000000000000000000000000000000000000
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE
  345678901234UUUUUUUUUUHHHHHHHHHHGGGGGGGGGGDDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAA

 PRNTR AS WRITTEN BY THE PROGRAM

  D    I123456789012IIIIIIIIIIOJJJJJ
  L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D    NNN9333993388BGGGGGG

  N    I13345678901200000000000000000
  E    000000000000000000000000000000
  W    00000000000000000000

  O    J234567890123YYYYYYYYYYGGGGGGG
  L    GGGFFFFFFFFFFDDDDDDDDDDSSSSSSS
  D    SSSXXXXXXXXXXEEEEEEE

  N    J234567890123YYYYYYYYYYGGGGGGG
  E    GGGFFFFFFFFFFDDDDDDDDDDSSSSSSS
  W    SSSXXXXXXXXXXEEEEEEE

  N     345678901234UUUUUUUUUUHHHHHHH
  E    HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
  W    SSSEEEEEEEEEEAAAAAAA

 THE PROGRAM TOOK  84 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >no
```

The following is the interaction necessary during the mutation phase. The user must indicate which mutant type programs are to be executed.

```
MUTATION PHASE
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ?  >select

ENTER THE NUMBERS OF THE MUTANT TYPES YOU WANT TO TURN ON AT THIS TIME.

       4      ****  INSERT FILLER TYPE  ****
       5      ****  FILLER SIZE ALTERATION TYPE  ****
       6      ****  ELEMENTARY ITEM REVERSAL TYPE  ****
       7      ****  FILE REFERENCE ALTERATION TYPE  ****
       8      ****  STATEMENT DELETION TYPE  ****
      10      ****  PERFORM --> GO TO TYPE  ****
      11      ****  THEN - ELSE REVERSAL TYPE  ****
      12      ****  STOP STATEMENT SUBSTITUTION TYPE  ****
      13      ****  THRU CLAUSE EXTENSION TYPE  ****
      14      ****  TRAP STATEMENT REPLACEMENT TYPE  ****
      19      ****  MOVE REVERSAL TYPE  ****
      20      ****  LOGICAL OPERATOR REPLACEMENT TYPE  ****
      21      ****  SCALAR FOR SCALAR REPLACEMENT  ****
      22      ****  CONSTANT FOR CONSTANT REPLACEMENT  ****
      23      ****  CONSTANT FOR SCALAR REPLACEMENT  ****
      25      ****  CONSTANT ADJUSTMENT  ****

TYPES ? >4 to 14 stop
```

The post run phase displays the mutant status as a result of the test cases currently defined. The user is given the opportunity to see the live mutants and the equivalent mutants during this phase and must indicate if the session is to be continued or not. For this example, the user specified the 'loop' option so that the CMS session will continue.

```
--- TESTCASE     1 ---
         250
         284 CONSIDERED        224 KILLED          60 REMAIN
MUTANT STATUS

TYPE        TOTAL     LIVE      PCT    EQUIV
INSERT         41        7    82.93       0
FILLSZ         38       14    63.16       0
ITEMRV         21        0   100.00       0
FILES           5        1    80.00       0
DELETE         54       13    75.93       0
PER GO          7        2    71.43       0
IF REV          3        1    66.67       0
STOP           53       10    81.13       0
THRU            8        2    75.00       0
TRAP           54       10    81.48       0

TOTALS
              284       60    78.87       0
DO YOU WANT TO SEE THE LIVE MUTANTS??>no
DO YOU WANT TO SEE THE EQUIVALENT MUTANTS??>no
WOULD YOU LIKE TO SEE THE TEST CASES??>no
LOOP OR HALT ? >loop
```

The user has indicated that the session is to continue.

```
PRE-RUN PHASE
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>lc15
 WHERE IS NEW-ETF?
>lc5
 OLD-ETF PROVIDED TO THE PROGRAM

 000000000012IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB3393B966666
 I1234567890I2IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB3B9B3BB66666
 J234567890123YYYYYYYYYY6666666666FFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM

 I1234567890I2IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB3B9939B6666S
 J234567890123YYYYYYYYYY6G66666G666FFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 PRNTR AS WRITTEN BY THE PROGRAM

  D    0000000000012IIIIIIIIIIJJJJJJJ
  L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D    NNNB9BBB999B96666666S

 THE PROGRAM TOOK  44 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>lc14
 WHERE IS NEW-ETF?
>lc5
 OLD-ETF PROVIDED TO THE PROGRAM

 I1234567890I2IIIIIIIIIIKJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB333B3B36G6G6
 J234567890123YYYYYYYYYY6666666666FFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 NEW-ETF PROVIDED TO THE PROGRAM

 I1234567890I2IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB3393333G666S
 J234567890123YYYYYYYYYY6666666666FFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE

 PRNTR AS WRITTEN BY THE PROGRAM

  O    I1234567890I2IIIIIIIIIIKJJJJJJ
  L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D    NNNBB39BBBBBBB6666666

  N    I1234567890I2IIIIIIIIIIJJJJJJJ
  E    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  W    NNNBB9BBBBBBBB6666666

 THE PROGRAM TOOK  48 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>lc11
 WHERE IS NEW-ETF?
>lc1
 OLD-ETF PROVIDED TO THE PROGRAM

 0000000000000000000000000000000000000000000

 NEW-ETF PROVIDED TO THE PROGRAM

 I1234567890I2IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBB9399B966666
 J234567890123YYYYYYYYYY6666666666FFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE
  345678901234UUUUUUUUUUHHHHHHHHHH6666666666DDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAA

 PRNTR AS WRITTEN BY THE PROGRAM

  O    000000000000000000000000000000
  L    000000000000000
  D

  N    I1234567890I2IIIIIIIIIIJJJJJJJ
  E    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  W    NNN9B33B33BB6666666
```

```
  N    J234567890123YYYYYYYYYYYGGGGGGG
  E    GGGFFFFFFFFFFFDDDDDDDDDDSSSSSSS
  W    SSSXXXXXXXXXXEEEEEEE

  N     345678901234UUUUUUUUUUHHHHHHH
  E    HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
  W    SSSEEEEEEEEEEEAAAAAAA

 THE PROGRAM TOOK  64 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >yes
 WHERE IS OLD-ETF?
>lc1
 WHERE IS NEW-ETF?
>lc11
 OLD-ETF PROVIDED TO THE PROGRAM

 I1234567890I2IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLNNNNNNNNNNBBBBBBBBGGGGG
 J234567890123YYYYYYYYYYGGGGGGGGGGFFFFFFFFFFDDDDDDDDDDSSSSSSSSSSXXXXXXXXXXEEEEE
  345678901234UUUUUUUUUUHHHHHHHHHHGGGGGGGGGGDDDDDDDDDDSSSSSSSSSSEEEEEEEEEEAAAAA

 NEW-ETF PROVIDED TO THE PROGRAM

 000000000000000000000000000000000000000000000000

 PRNTR AS WRITTEN BY THE PROGRAM

  N    000000000000000000000000000000
  E    000000000000000
  W

  O    I1234567890I2IIIIIIIIIIJJJJJJJ
  L    JJJKKKKKKKKKKLLLLLLLLLLNNNNNNN
  D    NNNBBBBBBBBBBGGGGGGG

  O    J234567890123YYYYYYYYYYYGGGGGGG
  L    GGGFFFFFFFFFFFDDDDDDDDDDSSSSSSS
  D    SSSXXXXXXXXXXEEEEEEE

  O     345678901234UUUUUUUUUUHHHHHHH
  L    HHHGGGGGGGGGGDDDDDDDDDDSSSSSSS
  D    SSSEEEEEEEEEEEAAAAAAA

 THE PROGRAM TOOK  64 STEPS
IS THIS TEST CASE ACCEPTABLE ? >yes
DO YOU WANT TO SUBMIT A TEST CASE ? >no
MUTATION PHASE
WHAT NEW MUTANT TYPES ARE TO BE CONSIDERED ?  >all
--- TESTCASE      1 ---
       250
       500
       750
       816 CONSIDERED        640 KILLED        174 REMAIN
--- TESTCASE      2 ---
       234 CONSIDERED         82 KILLED        152 REMAIN
--- TESTCASE      3 ---
       152 CONSIDERED          1 KILLED        151 REMAIN
--- TESTCASE      4 ---
       151 CONSIDERED         61 KILLED         90 REMAIN
--- TESTCASE      5 ---
        90 CONSIDERED         69 KILLED         21 REMAIN
MUTANT STATUS

TYPE      TOTAL     LIVE       PCT    EQUIV
INSERT       41        3     92.68     0
FILLSZ       38       12     68.42     0
ITEMRV       21        0    100.00     0
FILES         5        0    100.00     0
DELETE       54        1     98.15     0
PER GO        7        0    100.00     0
IF REV        3        0    100.00     0
STOP         53        0    100.00     0
THRU          8        0    100.00     0
TRAP         54        0    100.00     0
MOVE R       13        0    100.00     0
LOGIC        15        1     93.33     0
SUBSFS      704        4     99.43     0
SUBCFC       12        0    100.00     0
SUBCFS       58        0    100.00     0
C ADJ        12        0    100.00     0

TOTALS
           1098       21     98.09     0
```

```
DO YOU WANT TO SEE THE LIVE MUTANTS?>yes
   THE LIVE MUTANTS

FOR EACH MUTANT : HIT RETURN TO CONTINUE.  TYPE 'STOP' TO STOP.
TYPE 'EQUIV' TO JUDGE THE MUTANT EQUIVALENT.

 ****  INSERT FILLER TYPE  ****

THERE ARE          3  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 52
 ITS LEVEL NUMBER IS 3

>
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 53
 ITS LEVEL NUMBER IS 3

>
 A FILLER OF LENGTH ONE HAS BEEN INSERTED AFTER
 THE ITEM WHICH STARTS ON LINE 69
 ITS LEVEL NUMBER IS 3

>
 ****  FILLER SIZE ALTERATION TYPE  ****

THERE ARE         12  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 THE FILLER ON LINE 58  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 58  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 63  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 63  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 68  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 68  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 73  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 73  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 77  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 77  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 THE FILLER ON LINE 81  HAS HAD ITS SIZE  DECREMENTED  BY ONE.

>
 THE FILLER ON LINE 81  HAS HAD ITS SIZE  INCREMENTED  BY ONE.

>
 ****  STATEMENT DELETION TYPE  ****

THERE ARE          1  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 106  THE STATEMENT:
              GO TO 3150-CK-ADD-DEL
 HAS BEEN DELETED.

>
 ****  LOGICAL OPERATOR REPLACEMENT TYPE  ****
```

```
THERE ARE          1  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 102  THE STATEMENT:
             IF OLD-KEY  > NEW-KEY
 HAS BEEN CHANGED TO:
             IF OLD-KEY NOT < NEW-KEY

>

 ****  SCALAR FOR SCALAR REPLACEMENT  ****

THERE ARE          4  MUTANTS OF THIS TYPE LEFT.
DO YOU WANT TO SEE THEM?>yes
 ON LINE 129  THE STATEMENT:
             MOVE LINE1 TO N-LN1
 HAS BEEN CHANGED TO:
             MOVE NEW-REC TO N-LN1

>
 ON LINE 129  THE STATEMENT:
             MOVE LINE1 TO N-LN1
 HAS BEEN CHANGED TO:
             MOVE PRNT-WORK-AREA TO N-LN1

>
 ON LINE 138  THE STATEMENT:
             MOVE LINE1 TO LN1
 HAS BEEN CHANGED TO:
             MOVE OLD-REC TO LN1

>
 ON LINE 138  THE STATEMENT:
             MOVE LINE1 TO LN1
 HAS BEEN CHANGED TO:
             MOVE PRNT-WORK-AREA TO LN1

>
DO YOU WANT TO SEE THE EQUIVALENT MUTANTS?>no
WOULD YOU LIKE TO SEE THE TEST CASES?>no
LOOP OR HALT ? >halt

**** STOP
```

Testing CMS.1

Several routines from the CMS.1 system have been tested on the Fortran Mutation System (FMS.2). The subroutines which were chosen comply very closely to ANSI Standard Fortran.

FMS.2 will accept any ANSI Fortran program which does not use complex arithmetic or input/output statements [ABDLS]. FMS.2 will accept several subroutines for a testing run and will also accept character data as input which makes it possible to test CMS.1 routines which store the Cobol program and data in character format.

Some of the machine dependent features that had to be rewritten were the PRIME Fortran functions 'AND', 'INTL' (interger long), 'OR', and 'RS' (right shift). The 'RS' function can be implemented by simple division; to shift right n bits divide by 2 to the nth power. The PRIME function INTL, which converts a 16-bit integer into a 'long' 32-bit integer, can be deleted because the FMS.2 test was conducted on a 36-bit machine. The Fortran 'AND' function can be implemented by subtraction and the 'OR' function is implemented by addition, in the context in which they are used in the tested routines.

In CMS.1 a negative number is coded with a negative sign placed in the low order byte of the word containing the last character of the least significant digit, all the low order bytes of the words for the other digits contain a blank. Improvising for the negative sign on the FMS.2 system is accomplished by setting a bit in the second byte of the last word of a number. In FMS.2 a character is stored in the most significant byte with the remaining 4 bytes containing a blank. FMS.2 has an UNPACK and PACK routine that may be used by the user. The UNPACK routine takes an A5 word format and repacks it into a five word A1 format. The PACK routine reformats 5 words in A1 formats to a single word with an A5 format. The UNPACK and PACK routines were used in rewritting two of the routines that use the 'negative' sign. Some of the routines tested on FMS.2 use the subroutines MAKNEG, which turns the negative sign mask on; MAKPOS, which turns the negative sign mask off; or the logical function ISNEG which determines if the negative sign mask is on. These three routines have been expanded in-line to facilitate implementation on FMS.2. To code the MAKPOS subroutine it is necessary to turn the 'negative' bit off, this is accomplished by storing a blank in the second byte of the word containing the negative sign (call PACK with a blank in the second word which gets placed in the second byte of the word). MAKNEG is expanded in-line by calling PACK with the low order bit of the second word turned on; this word gets packed into the second byte. When MAKNEG is invoked, the negative sign mask is off. ISNEG is expanded by calling UNPACK and checking to see if the second word is blank, not negative, or non-blank, negative.

Another dependent feature, the $INSERT command, has been changed in all the routines to contain COMMON statements where needed or to insert constants where parameters were used.

The MOVENM and MOVENW routines are believed to be correct and the testing was done to increase confidence in the program's correctness. The two programs are shown in Figure 3. Mutation analysis on each subroutine indicates that no errors exist and that the two subroutines are correct. A listing of each subroutine with its equivalent mutants and the MUTANT STATE information is given in Figure 4. It can be seen that most of the equivalent mutants a e the absolute value or the never been zero mutant of a variable; these variables are always positive and never zero because they are referring to the memory location and length for either the sending field or destination field in the Cobol MOVE statement and this cannot be negative or zero. One important note to be made concerns the statement:

```
IF (K .EQ.  '#') IER=4
```

This conditional checks for undefined data. If the data is undefined, the data is moved entirely to the receiving field before the interpreter is halted and an error returned to the calling subroutine. The conditional statement:

```
IF (IER .NE.  0) GO TO 9999 as in MOVENW
IF (IER .NE.  0) GO TO 50 as in MOVENM
```

is located after the Fortran DO loop that moves the data; if this statement were moved inside the DO loop, then the error could cause the error return before all the data is moved. After further consideration, it was decided that evaluating the error condition on every iteration is larger than moving the remaining data to the receiving field. It should be noted that moving the undefined data to the receiving field has no effect because interpretation of the program is halted.

The MOVEED, numeric edited move, subroutine was submitted for mutation analysis because it has not been fully tested by conventional means. The program as modified for FMS.2 is in Figure 5.

The data for this subroutine consists of the following input and input/output data.

INPUT DATA

SOURCE - INTEGER data that contains the starting location in memory for the sending field.

SLEN - INTEGER data that specifies the length of the item in memory.

SDEC - INTEGER specifing the number of digits in the fraction part of a number.

DEST - INTEGER data that contains the starting location in memory for the receiving field.

DLEN - INTEGER data that specifies the length of the receiving data item in memory.

PLEN - INTEGER that specifies the length of the PICTURE specification.

PDIG - INTEGER that gives the number of digits in the PICTURE description.

PDEC - INTEGER specifying the number of digits in the fraction part of the PICTURE.

```
LISTING THE PROGRAM UNIT "MOVENW    "
       SUBROUTINE MOVENW(SOURCE,SLEN,DEST,DLEN)
       INTEGER MLEN, K, SUB2, SUB1, LOOPHI, I, IHI, IER
       INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
       CHAR MEMORY(625)
       INTEGER DLEN, DEST, SLEN, SOURCE
       INPUT OUTPUT IER, MEMORY
       INPUT DLEN, DEST, SLEN, SOURCE
       MLEN = DLEN
       IF(SLEN .LT. MLEN) MLEN = SLEN
       LOOPHI = (DEST + MLEN) - 1
       SUB2 = SOURCE - 1
       DO 20 SUB1=DEST, LOOPHI
       SUB2 = SUB2 + 1
       K = MEMORY(SUB2)
       IF(K .EQ. '#') IER = 4
 20    MEMORY(SUB1) = K
       IF(IER .NE. 0) GOTO 9999
       IF(DLEN .LE. MLEN) GOTO 9999
       I = LOOPHI + 1
       LOOPHI = (DEST + DLEN) - 1
       DO 30 SUB1=I, LOOPHI
 30    MEMORY(SUB1) = ' '
 9999  CONTINUE
       RETURN
       END

LISTING THE PROGRAM UNIT "MOVENM    "
       SUBROUTINE MOVENM(SOURCE,SLEN,SDEC,DEST,DLEN,DDEC,TYPPE)
       LOGICAL NEGNO
       INTEGER X(5), PTNEGD, PTNEGS, K, SUB2, SUB1, LOOPHI, LEND
       INTEGER LENS, I, IHI, DDECPT, SDECPT, IER, STMT(3,10)
       INTEGER CODE(30), SYMTAB(10,9)
       CHAR MEMORY(625)
       INTEGER TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       INPUT OUTPUT IER, MEMORY
       INPUT TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       PTNEGS = (SOURCE + SLEN) - 1
       PTNEGD = (DEST + DLEN) - 1
       CALL UNPACK(MEMORY(PTNEGS),X,5)
       NEGNO = X(2) .EQ. '-'

       X(2) = ' '
       IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)
       LENS = SLEN - SDEC
       LEND = DLEN - DDEC
       SDECPT = SOURCE + LENS
       DDECPT = DEST + LEND
       SUB1 = DDECPT - 1
       IF(SDEC .EQ. 0 .OR. DDEC .EQ. 0) GOTO 22
       IHI = (SDEC + SDECPT) - 1
       IF(DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
       DO 20 SUB2=SDECPT, IHI
       SUB1 = SUB1 + 1
       K = MEMORY(SUB2)
       IF(K .EQ. '#') IER = 4
 20    MEMORY(SUB1) = K
       IF(IER .NE. 0) GOTO 50
 22    IF(DDEC .LE. SDEC) GOTO 30
       I = SUB1 + 1
       IHI = (DEST + DLEN) - 1
       DO 25 SUB1=I, IHI
 25    MEMORY(SUB1) = '0'
 30    LOOPHI = LEND
       IF(LENS .LE. LEND) LOOPHI = LENS
       SUB1 = DDECPT
       SUB2 = SDECPT
       IF(LEND .EQ. 0) GOTO 50
       IF(LENS .EQ. 0) GOTO 41
       DO 40 I=1, LOOPHI
       SUB1 = SUB1 - 1
       SUB2 = SUB2 - 1
       K = MEMORY(SUB2)
       IF(K .EQ. '#') IER = 4
 40    MEMORY(SUB1) = K
       IF(IER .NE. 0) GOTO 50
       IF(LEND .LE. LENS) GOTO 50
 41    IHI = SUB1 - 1
       DO 45 I=DEST, IHI
 45    MEMORY(I) = '0'
 50    X(2) = '-'
       IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)
       IF(.NOT. (NEGNO .AND. TYPPE .EQ. 2)) RETURN
       CALL UNPACK(MEMORY(PTNEGD),X,5)
       X(2) = '-'
       CALL PACK(X,MEMORY(PTNEGD),5)
       RETURN
       END
```

Figure 3 MOVENW and MOVENM Original Program Listings

```
LISTING THE PROGRAM UNIT "MOVENW    " WITH SPECIFIED EQUIV MUTANTS

        SUBROUTINE MOVENW(SOURCE,SLEN,DEST,DLEN)
        INTEGER MLEN, K, SUB2, SUB1, LOOPHI, I, IHI, IER
        INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
        CHAR MEMORY(425)
        INTEGER DLEN, DEST, SLEN, SOURCE
        INPUT OUTPUT IER, MEMORY
        INPUT DLEN, DEST, SLEN, SOURCE
        MLEN = DLEN                                                  1

$755$   MLEN = ABS DLEN
$757$   MLEN = ZPUSH DLEN

        IF(SLEN .LT. MLEN) MLEN = SLEN                            2   3

$43$    IF(SLEN .LT. DLEN) MLEN = SLEN
$530$   IF(-- SLEN .LT. MLEN) MLEN = SLEN
$532$   IF(SLEN .LT. ++ MLEN) MLEN = SLEN
$727$   IF(SLEN .LE. MLEN) MLEN = SLEN
$759$   IF(ABS SLEN .LT. MLEN) MLEN = SLEN
$760$   IF(ZPUSH SLEN .LT. MLEN) MLEN = SLEN
$761$   IF(SLEN .LT. ABS MLEN) MLEN = SLEN
$763$   IF(SLEN .LT. ZPUSH MLEN) MLEN = SLEN
$764$   IF(SLEN .LT. MLEN) MLEN = ABS SLEN
$765$   IF(SLEN .LT. MLEN) MLEN = ZPUSH SLEN

        LOOPHI = (DEST + MLEN) - 1                                   4

$767$   LOOPHI = (ABS DEST + MLEN) - 1
$769$   LOOPHI = (ZPUSH DEST + MLEN) - 1
$770$   LOOPHI = (DEST + ABS MLEN) - 1
$772$   LOOPHI = (DEST + ZPUSH MLEN) - 1
$773$   LOOPHI = ABS (DEST + MLEN) - 1
$775$   LOOPHI = ZPUSH (DEST + MLEN) - 1
$776$   LOOPHI = ABS ((DEST + MLEN) - 1)
$778$   LOOPHI = ZPUSH ((DEST + MLEN) - 1)

        SUB2 = SOURCE - 1                                            5

$779$   SUB2 = ABS SOURCE - 1
$781$   SUB2 = ZPUSH SOURCE - 1
$782$   SUB2 = ABS (SOURCE - 1)
$784$   SUB2 = ZPUSH (SOURCE - 1)
        DO 20 SUB1=DEST, LOOPHI                                      6

$785$   DO 20 SUB1=ABS DEST, LOOPHI
$787$   DO 20 SUB1=ZPUSH DEST, LOOPHI
$788$   DO 20 SUB1=DEST, ABS LOOPHI
$790$   DO 20 SUB1=DEST, ZPUSH LOOPHI
$892$   FOR 20 SUB1=DEST, LOOPHI

        SUB2 = SUB2 + 1                                              7

$791$   SUB2 = ABS SUB2 + 1
$793$   SUB2 = ZPUSH SUB2 + 1
$794$   SUB2 = ABS (SUB2 + 1)
$796$   SUB2 = ZPUSH (SUB2 + 1)

        K = MEMORY(SUB2)                                             8

$797$   K = MEMORY(ABS SUB2)
$799$   K = MEMORY(ZPUSH SUB2)

        IF(K .EQ. '#') IER = 4                                    9  10

$554$   IF(MEMORY(SUB2) .EQ. '#') IER = 4
$800$   IF(ABS K .EQ. '#') IER = 4
$832$   IF(ZPUSH K .EQ. '#') IER = 4

 20     MEMORY(SUB1) = K                                            11

$559$   MEMORY(SUB1) = MEMORY(SUB2)
$833$   MEMORY(ABS SUB1) = K
$835$   MEMORY(ZPUSH SUB1) = K
$808$   MEMORY(SUB1) = ZPUSH K

        IF(IER .NE. 0) GOTO 9999                                 12  13
$745$   IF(IER .GT. 0) GOTO 9999
$874$   IF(IER .NE. 0) RETURN
```

Figure 4 MOVENW and MOVENM Listings With Equivalent Mutants and Mutant State Information

```
      IF(DLEN .LE. MLEN) GOTO 9999                                  14  15

$254$ IF(DLEN .LE. SLEN) GOTO 9999
$749$ IF(DLEN .EQ. MLEN) GOTO 9999
$809$ IF(ABS DLEN .LE. MLEN) GOTO 9999
$811$ IF(ZPUSH DLEN .LE. MLEN) GOTO 9999
$812$ IF(DLEN .LE. ABS MLEN) GOTO 9999
$814$ IF(DLEN .LE. ZPUSH MLEN) GOTO 9999
$879$ IF(DLEN .LE. MLEN) RETURN

      I = LOOPHI + 1                                                    16

$815$ I = ABS LOOPHI + 1
$817$ I = ZPUSH LOOPHI + 1
$819$ I = ABS (LOOPHI + 1)
$820$ I = ZPUSH (LOOPHI + 1)

      LOOPHI = (DEST + DLEN) - 1                                        17
$821$ LOOPHI = (ABS DEST + DLEN) - 1
$823$ LOOPHI = (ZPUSH DEST + DLEN) - 1
$824$ LOOPHI = (DEST + ABS DLEN) - 1
$825$ LOOPHI = (DEST + ZPUSH DLEN) - 1
$827$ LOOPHI = ABS (DEST + DLEN) - 1
$829$ LOOPHI = ZPUSH (DEST + DLEN) - 1
$830$ LOOPHI = ABS ((DEST + DLEN) - 1)
$832$ LOOPHI = ZPUSH ((DEST + DLEN) - 1)

      DO 30 SUB1=I, LOOPHI                                              18

$833$ DO 30 SUB1=ABS I, LOOPHI
$835$ DO 30 SUB1=ZPUSH I, LOOPHI
$836$ DO 30 SUB1=I, ABS LOOPHI
$838$ DO 30 SUB1=I, ZPUSH LOOPHI
$891$ DO 9999 SUB1=I, LOOPHI
$893$ FOR 30 SUB1=I, LOOPHI

 30   MEMORY(SUB1) = ' '                                                19

$839$ MEMORY(ABS SUB1) = ' '
$841$ MEMORY(ZPUSH SUB1) = ' '

 9999 CONTINUE                                                          20

$883$ RETURN

      RETURN                                                            21
      END
MUTANT STATE FOR MOVENW

    FOR EXPERIMENT "MOVENW    " THIS IS RUN    7

    NUMBER OF TEST CASES =  11


    NUMBER OF MUTANTS =     893
    NUMBER OF DEAD MUTANTS =     821 ( 91.9%)
    NUMBER OF LIVE MUTANTS =       0 (  0.0%)
    NUMBER OF EQUIV MUTANTS =     72 (  8.1%)


    NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS   393
    NORMALIZED MUTANT RATIO 821.0%
    NUMBER OF MUTATABLE STATEMENTS =      21
    GIVING A MUTANTS/STATEMENT RATIO OF       42.52

    NUMBER OF DATA REFERENCES =      48
    NUMBER OF UNIQUE DATA REFERENCES =      16


   ALL MUTANT TYPES HAVE BEEN ENABLED
```

Figure 4 cont.

```
LISTING THE PROGRAM UNIT "MOVENM    " WITH SPECIFIED EQUIV MUTANTS

       SUBROUTINE MOVENM(SOURCE,SLEN,SDEC,DEST,DLEN,DDEC,TYPPE)
       LOGICAL NEGNO
       INTEGER X(5), PTNEGD, PTNEGS, K, SUB2, SUB1, LOOPHI, LEND
       INTEGER LENS, I, IHI, DDECPT, SDECPT, IER, STMT(3,10)
       INTEGER CODE(30), SYMTAB(10,9)
       CHAR MEMORY(425)
       INTEGER TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       INPUT OUTPUT IER, MEMORY
       INPUT TYPPE, DDEC, DLEN, DEST, SDEC, SLEN, SOURCE
       PTNEGS = (SOURCE + SLEN) - 1                                     23

$46650$  PTNEGS = (ABS SOURCE + SLEN) - 1
$46652$  PTNEGS = (ZPUSH SOURCE + SLEN) - 1
$46653$  PTNEGS = (SOURCE + ABS SLEN) - 1
$46655$  PTNEGS = (SOURCE + ZPUSH SLEN) - 1
$46656$  PTNEGS = ABS (SOURCE + SLEN) - 1
$46658$  PTNEGS = ZPUSH (SOURCE + SLEN) - 1
$46659$  PTNEGS = ABS ((SOURCE + SLEN) - 1)
$46661$  PTNEGS = ZPUSH ((SOURCE + SLEN) - 1)

       PTNEGD = (DEST + DLEN) - 1                                       24

$46652$  PTNEGD = (ABS DEST + DLEN) - 1
$46664$  PTNEGD = (ZPUSH DEST + DLEN) - 1
$46665$  PTNEGD = (DEST + ABS DLEN) - 1
$46657$  PTNEGD = (DEST + ZPUSH DLEN) - 1
$46658$  PTNEGD = ABS (DEST + DLEN) - 1
$46670$  PTNEGD = ZPUSH (DEST + DLEN) - 1
$46671$  PTNEGD = ABS ((DEST + DLEN) - 1)
$46673$  PTNEGD = ZPUSH ((DEST + DLEN) - 1)

       CALL UNPACK(MEMORY(PTNEGS),X,5)                                  25

$46674$  CALL UNPACK(MEMORY(ABS PTNEGS),X,5)
$46676$  CALL UNPACK(MEMORY(ZPUSH PTNEGS),X,5)

       NEGNO = X(2) .EQ. '-'                                            26

$45545$  NEGNO = X(2) .GE. '-'
$46577$  NEGNO = ABS X(2) .EQ. '-'
$46579$  NEGNO = ZPUSH X(2) .EQ. '-'

       X(2) = ' '                                                       27
       IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)                       28 29

$46580$  IF(NEGNO) CALL PACK(X,MEMORY(ABS PTNEGS),5)
$46582$  IF(NEGNO) CALL PACK(X,MEMORY(ZPUSH PTNEGS),5)

       LENS = SLEN - SDEC                                               30

$46683$  LENS = ABS SLEN - SDEC
$46685$  LENS = ZPUSH SLEN - SDEC
$46686$  LENS = SLEN - ABS SDEC
$46689$  LENS = ABS (SLEN - SDEC)

       LEND = DLEN - DDEC                                               31

$46692$  LEND = ABS DLEN - DDEC
$46694$  LEND = ZPUSH DLEN - DDEC
$46695$  LEND = DLEN - ABS DDEC
$46698$  LEND = ABS (DLEN - DDEC)

       SDECPT = SOURCE + LENS                                           32

$47031$  SDECPT = ABS SOURCE + LENS
$47033$  SDECPT = ZPUSH SOURCE + LENS
$47034$  SDECPT = SOURCE + ABS LENS
$47007$  SDECPT = ABS (SOURCE + LENS)
$47009$  SDECPT = ZPUSH (SOURCE + LENS)

       DDECPT = DEST + LEND                                             33

$47010$  DDECPT = ABS DEST + LEND
$47012$  DDECPT = ZPUSH DEST + LEND
$47013$  DDECPT = DEST + ABS LEND
$47016$  DDECPT = ABS (DEST + LEND)
$47018$  DDECPT = ZPUSH (DEST + LEND)
```

Figure 4 cont.

```
       SUB1 = DDECPT - 1                                             34

$47198  SUB1 = ABS DDECPT - 1
$47218  SUB1 = ZPUSH DDECPT - 1
$47228  SUB1 = ABS (DDECPT - 1)
$47248  SUB1 = ZPUSH (DDECPT - 1)

       IF(SDEC .EQ. 0 .OR. DDEC .EQ. 0) GOTO 22                   35 36

$45503$  IF(SDEC .LE. 0 .OR. DDEC .EQ. 0) GOTO 22
$45557$  IF(SDEC .EQ. 0 .OR. DDEC .LE. 0) GOTO 22

       IHI = (SDEC + SDECPT) - 1                                     37

$47255$  IHI = (ABS SDEC + SDECPT) - 1
$47275$  IHI = (ZPUSH SDEC + SDECPT) - 1
$47295$  IHI = (SDEC + ABS SDECPT) - 1
$47305$  IHI = (SDEC + ZPUSH SDECPT) - 1
$47315$  IHI = ABS (SDEC + SDECPT) - 1
$47335$  IHI = ZPUSH (SDEC + SDECPT) - 1
$47345$  IHI = ABS ((SDEC + SDECPT) - 1)
$47365$  IHI = ZPUSH ((SDEC + SDECPT) - 1)

       IF(DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1               38 39

$43005$  IF(++ DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$45635$  IF(DDEC .LT. SDEC) IHI = (DDEC + SDECPT) - 1
$47375$  IF(ABS DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$47395$  IF(ZPUSH DDEC .LE. SDEC) IHI = (DDEC + SDECPT) - 1
$47405$  IF(DDEC .LE. ABS SDEC) IHI = (DDEC + SDECPT) - 1
$47425$  IF(DDEC .LE. ZPUSH SDEC) IHI = (DDEC + SDECPT) - 1
$47435$  IF(DDEC .LE. SDEC) IHI = (ABS DDEC + SDECPT) - 1
$47455$  IF(DDEC .LE. SDEC) IHI = (ZPUSH DDEC + SDECPT) - 1
$47465$  IF(DDEC .LE. SDEC) IHI = (DDEC + ABS SDECPT) - 1
$47485$  IF(DDEC .LE. SDEC) IHI = (DDEC + ZPUSH SDECPT) - 1
       *6 MORE*

       DO 20 SUB2=SDECPT, IHI                                        40

$47555$  DO 20 SUB2=ABS SDECPT, IHI
$47575$  DO 20 SUB2=ZPUSH SDECPT, IHI
$47585$  DO 20 SUB2=SDECPT, ABS IHI
$47605$  DO 20 SUB2=SDECPT, ZPUSH IHI
$50925$  FOR 20 SUB2=SDECPT, IHI

       SUB1 = SUB1 + 1                                               41

$47615$  SUB1 = ABS SUB1 + 1
$47635$  SUB1 = ZPUSH SUB1 + 1
$47645$  SUB1 = ABS (SUB1 + 1)
$47655$  SUB1 = ZPUSH (SUB1 + 1)

       K = MEMORY(SUB2)                                              42

$47675$  K = MEMORY(ABS SUB2)
$47695$  K = MEMORY(ZPUSH SUB2)

       IF(K .EQ. '#') IER = 4                                     43 44

$22425$  IF(K .EQ. '#') IER = DLEN
$22445$  IF(K .EQ. '#') IER = LENS
$22455$  IF(K .EQ. '#') IER = SDEC
$22475$  IF(K .EQ. '#') IER = DDEC
$34675$  IF(MEMORY(SUB2) .EQ. '#') IER = 4
$47705$  IF(ABS K .EQ. '#') IER = 4
$47725$  IF(ZPUSH K .EQ. '#') IER = 4

 20    MEMORY(SUB1) = K                                              45

$34845$  MEMORY(SUB1) = MEMORY(SUB2)
$47735$  MEMORY(ABS SUB1) = K
$47755$  MEMORY(ZPUSH SUB1) = K
$47765$  MEMORY(SUB1) = ABS K
$47785$  MEMORY(SUB1) = ZPUSH K

       IF(IER .NE. 0) GOTO 50                                     46 47

$45815$  IF(IER .GT. 0) GOTO 50
$50265$  IF(IER .NE. 0) GOTO 40
```

Figure 4 cont.

```
 22     IF(DDEC .LE. SDEC) GOTO 30                                  49  49

$47798  IF(ABS DDEC .LE. SDEC) GOTO 30
$47828  IF(DDEC .LE. ABS SDEC) GOTO 30

        I = SUB1 + 1                                                    50

$47858  I = ABS SUB1 + 1
$47878  I = ZPUSH SUB1 + 1
$47898  I = ABS (SUB1 + 1)
$47908  I = ZPUSH (SUB1 + 1)

        IHI = (DEST + DLEN) - 1                                         51

$47918  IHI = (ABS DEST + DLEN) - 1
$47938  IHI = (ZPUSH DEST + DLEN) - 1
$47948  IHI = (DEST + ABS DLEN) - 1
$47968  IHI = (DEST + ZPUSH DLEN) - 1
$47978  IHI = ABS (DEST + DLEN) - 1
$47998  IHI = ZPUSH (DEST + DLEN) - 1
$48008  IHI = ABS ((DEST + DLEN) - 1)
$48028  IHI = ZPUSH ((DEST + DLEN) - 1)

        DO 25 SUB1=I, IHI                                               52

$11688  DO 25 SUB1=I, PTNEGD
$48038  DO 25 SUB1=ABS I, IHI
$48058  DO 25 SUB1=ZPUSH I, IHI
$48068  DO 25 SUB1=I, ABS IHI
$48088  DO 25 SUB1=I, ZPUSH IHI
$50738  DO 30 SUB1=I, IHI
$50938  FOR 25 SUB1=I, IHI

 25     MEMORY(SUB1) = '0'                                              53

$48098  MEMORY(ABS SUB1) = '0'
$48118  MEMORY(ZPUSH SUB1) = '0'

 30     LOOPHI = LEND                                                   54

$48128  LOOPHI = ABS LEND

        IF(LENS .LE. LEND) LOOPHI = LENS                            55  56

$12938  IF(LENS .LE. LOOPHI) LOOPHI = LENS
$43598  IF(** LENS .LE. LEND) LOOPHI = LENS
$45918  IF(LENS .LT. LEND) LOOPHI = LENS
$48158  IF(ABS LENS .LE. LEND) LOOPHI = LENS
$48198  IF(LENS .LE. ABS LEND) LOOPHI = LENS
$48218  IF(LENS .LE. LEND) LOOPHI = ABS LENS

        SUB1 = DDECPT                                                   57

$48248  SUB1 = ABS DDECPT
$48268  SUB1 = ZPUSH DDECPT

        SUB2 = SDECPT                                                   58

$48278  SUB2 = ABS SDECPT
$48298  SUB2 = ZPUSH SDECPT

        IF(LEND .EQ. 0) GOTO 50                                     59  60

$23388  IF(LEND .EQ. IER) GOTO 50
$45998  IF(LEND .LE. 0) GOTO 50

        IF(LENS .EQ. 0) GOTO 41                                     61  62

$14438  IF(LOOPHI .EQ. 0) GOTO 41
$46068  IF(LENS .LE. 0) GOTO 41

        DO 40 I=1, LOOPHI                                               63

$14468  DO 40 SOURCE=1, LOOPHI
$14478  DO 40 SLEN=1, LOOPHI
$14508  DO 40 DLEN=1, LOOPHI
$14538  DO 40 SDEC=1, LOOPHI
$14558  DO 40 DDEC=1, LOOPHI
$14568  DO 40 SDECPT=1, LOOPHI
$14578  DO 40 DDECPT=1, LOOPHI
$14598  DO 40 IHI=1, LOOPHI
$14618  DO 40 K=1, LOOPHI
$14638  DO 40 LOOPHI=1, LOOPHI
        *6 MORE*
```

Figure 4 cont.

```
       SUB1 = SUB1 - 1                                                        64

$4833$  SUB1 = ABS SUB1 - 1
$4835$  SUB1 = ZPUSH SUB1 - 1
$4836$  SUB1 = ABS (SUB1 - 1)
$4838$  SUB1 = ZPUSH (SUB1 - 1)

       SUB2 = SUB2 - 1                                                        65

$4839$  SUB2 = ABS SUB2 - 1
$4841$  SUB2 = ZPUSH SUB2 - 1
$4842$  SUB2 = ABS (SUB2 - 1)
$4844$  SUB2 = ZPUSH (SUB2 - 1)

       K = MEMORY(SUB2)                                                       66

$4845$  K = MEMORY(ABS SUB2)
$4847$  K = MEMORY(ZPUSH SUB2)

       IF(K .EQ. '#') IER = 4                                              67 68

$3573$  IF(MEMORY(SUB2) .EQ. '#') IER = 4
$4848$  IF(ABS K .EQ. '#') IER = 4
$4850$  IF(ZPUSH K .EQ. '#') IER = 4

 40    MEMORY(SUB1) = K                                                       69

$3658$  MEMORY(SUB1) = MEMORY(SUB2)
$4851$  MEMORY(ABS SUB1) = K
$4853$  MEMORY(ZPUSH SUB1) = K
$4856$  MEMORY(SUB1) = ZPUSH K

       IF(IER .NE. 0) GOTO 50                                              70 71

$4623$  IF(IER .GT. 0) GOTO 50
$5053$  IF(IER .NE. 0) GOTO 20

       IF(LEND .LE. LENS) GOTO 50                                          72 73

$1743$  IF(LEND .LE. LOOPHI) GOTO 50
$4857$  IF(ABS LEND .LE. LENS) GOTO 50
$4859$  IF(ZPUSH LEND .LE. LENS) GOTO 50
$4860$  IF(LEND .LE. ABS LENS) GOTO 50
$4862$  IF(LEND .LE. ZPUSH LENS) GOTO 50

 41    IHI = SUB1 - 1                                                         74

$4863$  IHI = ABS SUB1 - 1
$4865$  IHI = ZPUSH SUB1 - 1
$4866$  IHI = ABS (SUB1 - 1)
$4868$  IHI = ZPUSH (SUB1 - 1)

       DO 45 I=DEST, IHI                                                      75

$4869$  DO 45 I=ABS DEST, IHI
$4871$  DO 45 I=ZPUSH DEST, IHI
$4872$  DO 45 I=DEST, ABS IHI
$4874$  DO 45 I=DEST, ZPUSH IHI
$5091$  DO 50 I=DEST, IHI
$5095$  FOR 45 I=DEST, IHI

 45    MEMORY(I) = '0'                                                        76

$4875$  MEMORY(ABS I) = '0'
$4877$  MEMORY(ZPUSH I) = '0'

 50    X(2) = '-'                                                             77
       IF(NEGNO) CALL PACK(X,MEMORY(PTNEGS),5)                             78 79

$4878$  IF(NEGNO) CALL PACK(X,MEMORY(ABS PTNEGS),5)
$4880$  IF(NEGNO) CALL PACK(X,MEMORY(ZPUSH PTNEGS),5)

       IF(.NOT. (NEGNO .AND. TYPPE .EQ. 2)) RETURN                         80 81

$4881$  IF(.NOT. (NEGNO .AND. ABS TYPPE .EQ. 2)) RETURN
$4883$  IF(.NOT. (NEGNO .AND. ZPUSH TYPPE .EQ. 2)) RETURN

       CALL UNPACK(MEMORY(PTNEGD),X,5)                                        82

$57$   CALL UNPACK(MEMORY(PTNEGD),X,4)
$2560$  CALL UNPACK(MEMORY(PTNEGD),X,SDEC)
$2572$  CALL UNPACK(MEMORY(PTNEGD),X,TYPPE)
$3015$  CALL UNPACK(MEMORY(PTNEGD),X,1)
$3016$  CALL UNPACK(MEMORY(PTNEGD),X,2)
$4884$  CALL UNPACK(MEMORY(ABS PTNEGD),X,5)
$4886$  CALL UNPACK(MEMORY(ZPUSH PTNEGD),X,5)
```

Figure 4 cont.

```
                                                                          83
      X(2) = '-'

$2593$  X(TYPPE) = '-'
      CALL PACK(X,MEMORY(PTNEGD),5)                                       84

$4887$  CALL PACK(X,MEMORY(ABS PTNEGD),5)
$4889$  CALL PACK(X,MEMORY(ZPUSH PTNEGD),5)

      RETURN                                                              85
      END
```

MUTANT ELIMINATION PROFILE FOR MOVENM

MUTANT TYPE	TOTAL	DEAD		LIVE		EQUIV	
CONSTANT REPLACEMENT	64	63	98.4%	0	0.0%	1	1.6%
SCALAR VARIABLE REPLACEME	1920	1906	99.3%	0	0.0%	14	0.7%
SCALAR FOR CONSTANT REP.	630	622	98.7%	0	0.0%	8	1.3%
CONSTANT FOR SCALAR REP.	331	331	100.0%	0	0.0%	0	0.0%
SOURCE CONSTANT REPLACEME	102	100	98.0%	0	0.0%	2	2.0%
ARRAY REF. FOR CONSTANT R	179	179	100.0%	0	0.0%	0	0.0%
ARRAY REF. FOR SCALAR REP	547	543	99.3%	0	0.0%	4	0.7%
COMPARABLE ARRAY NAME RE	40	40	100.0%	0	0.0%	0	0.0%
CONSTANT FOR ARRAY REF RE	40	40	100.0%	0	0.0%	0	0.0%
SCALAR FOR ARRAY REF REP.	315	315	100.0%	0	0.0%	0	0.0%
ARRAY REF. FOR ARRAY REF.	75	75	100.0%	0	0.0%	0	0.0%
UNARY OPERATOR INSERTION	191	189	99.0%	0	0.0%	2	1.0%
ARITHMETIC OPERATOR REPLA	107	107	100.0%	0	0.0%	0	0.0%
RELATIONAL OPERATOR REPLA	98	89	90.8%	0	0.0%	9	9.2%
LOGICAL CONNECTOR REPLACE	10	10	100.0%	0	0.0%	0	0.0%
ABSOLUTE VALUE INSERTION	240	93	38.8%	0	0.0%	147	61.3%
STATEMENT ANALYSIS	29	29	100.0%	0	0.0%	0	0.0%
STATEMENT DELETION	35	35	100.0%	0	0.0%	0	0.0%
RETURN STATEMENT REPLACEM	61	61	100.0%	0	0.0%	0	0.0%
GOTO STATEMENT REPLACEMEN	49	47	95.9%	0	0.0%	2	4.1%
DO STATEMENT END REPLACEM	32	25	78.1%	0	0.0%	7	21.9%

MUTANT STATE FOR MOVENM

FOR EXPERIMENT "MOVENM " THIS IS RUN 22

NUMBER OF TEST CASES = 41

NUMBER OF MUTANTS = 5095
NUMBER OF DEAD MUTANTS = 4899 (96.2%)
NUMBER OF LIVE MUTANTS = 0 (0.0%)
NUMBER OF EQUIV MUTANTS = 196 (3.8%)

NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS 2206
NORMALIZED MUTANT RATIO *****%
NUMBER OF MUTATABLE STATEMENTS = 63
GIVING A MUTANTS/STATEMENT RATIO OF 80.87

NUMBER OF DATA REFERENCES = 158
NUMBER OF UNIQUE DATA REFERENCES = 32

ALL MUTANT TYPES HAVE BEEN ENABLED

Figure 4 cont.

```
LISTING THE PROGRAM UNIT "MOVEED    "

      SUBROUTINE MOVEED(SOURCE,SLEN,SDEC,DEST,DLEN,PLEN,PDIG,PDEC,
     * PIC,IER)
      LOGICAL SUPRES, NEGNO
      INTEGER X(5), SUB2, SUB1, IHI, PLDIG, IVAR, I, SCOUNT, DESTHI
      INTEGER CHAR, PDIGLN, SDIG, SARRAY(50), PICST, DDEC
      INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
      CHAR MEMORY(310)
      INTEGER IER
      CHAR PIC(10)
      INTEGER PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      INPUT OUTPUT MEMORY, IER
      INPUT PIC, PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      SUPRES = .TRUE.
      DO 5 I=1, PLEN
5     SARRAY(I) = '0'
      PLDIG = PDIG - PDEC
      SDIG = SLEN - SDEC
      IF(SDEC .EQ. 0) GOTO 11
      SUB1 = PLDIG
      SUB2 = (SOURCE + SDIG) - 1
      DO 10 I=1, SDEC
      SUB1 = SUB1 + 1
      SUB2 = SUB2 + 1
      IF(MEMORY(SUB2) .EQ. '#') IER = 4
10    SARRAY(SUB1) = MEMORY(SUB2)
      IF(IER .NE. 0) GOTO 101
11    IF (SDIG .GE. PLDIG) IHI = PLDIG
      IF(SDIG .LT. PLDIG) IHI = SDIG
      SUB1 = PLDIG + 1
      SUB2 = SOURCE + SDIG
      DO 15 I=1, IHI
      SUB1 = SUB1 - 1
      SUB2 = SUB2 - 1
      IF(MEMORY(SUB2) .EQ. '#') IER = 4
15    SARRAY(SUB1) = MEMORY(SUB2)
      IF(IER .NE. 0) GOTO 101
16    SUB1 = (SOURCE + SLEN) - 1
      CALL UNPACK(MEMORY(SUB1),X,2)
      NEGNO = X(2) .EQ. '-'
      SUB1 = DEST
      SCOUNT = 0
      DO 100 I=1, PLEN
      SUB1 = DEST + I
      IF((DEST + I) - 1 .GT. (DLEN + DEST) - 1)) GOTO
      CHAR = PIC(I)
      IF(PIC(I) .EQ. '9') SUPRES = .FALSE.
      IF(SARRAY(SCOUNT + 1) .NE. '0') SUPRES = .FALSE.
      IF(CHAR .NE. '-') GOTO 20
      MEMORY(SUB1 - 1) = ' '
      IF(I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      IF (NEGNO) MEMORY(SUB1 - 1) = '-'
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '
      GOTO 100
20    IF(CHAR .NE. '+') GOTO 30
      IF(I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'
      IF(I .EQ. 1 .AND. .NOT. NEGNO) MEMORY(SUB1 - 1) = '+'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      IF (NEGNO) MEMORY(SUB1 - 1) = '-'
      IF (.NOT. NEGNO) MEMORY(SUB1 - 1) = '+'
      IF(MEMORY(SUB1 - 2) .EQ. '+') MEMORY(SUB1 - 2) = ' '
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '
      GOTO 100
30    IF(CHAR .NE. '$') GOTO 40
      IF (I .EQ. 1) MEMORY(SUB1 - 1) = '$'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      MEMORY(SUB1 - 1) = '$'
      IF(MEMORY(SUB1 - 2) .EQ. '$') MEMORY(SUB1 - 2) = ' '
      GOTO 100
```

Figure 5 MOVEED Original Program Listing

```
40      IF(CHAR .NE. '*') GOTO 50
        SCOUNT = SCOUNT + 1
        IF(.NOT. SUPRES) GOTO 99
        MEMORY(SUB1 - 1) = '*'
        GOTO 100
50      IF(CHAR .NE. 'Z') GOTO 55
        SCOUNT = SCOUNT + 1
        IF(.NOT. SUPRES) GOTO 99
        MEMORY(SUB1 - 1) = ' '
        GOTO 100
55      IF(CHAR .NE. '9') GOTO 60
        SCOUNT = SCOUNT + 1
        MEMORY(SUB1 - 1) = SARRAY(SCOUNT)
        GOTO 100
60      IF(CHAR .NE. 'B') GOTO 70
        MEMORY(SUB1 - 1) = ' '
        GOTO 100
70      IF(CHAR .NE. '/') GOTO 80
        MEMORY(SUB1 - 1) = '/'
        GOTO 100
80      IF(CHAR .NE. 'V') GOTO 81
        GOTO 100
81      IF(CHAR .NE. '.') GOTO 82
        MEMORY(SUB1 - 1) = '.'
        GOTO 100
82      IF(CHAR .NE. ',') GOTO 83
        IF(.NOT. SUPRES) MEMORY(SUB1 - 1) = ','
        IF(SUPRES) MEMORY(SUB1 - 1) = ' '
        GOTO 100
83      IER = 3
        GOTO 101
99      MEMORY(SUB1 - 1) = SARRAY(SCOUNT)
100     CONTINUE
101     CONTINUE
        RETURN
        END
```

Figure 5 cont.

```
TEST CASE NUMBER      9
PARAMETERS ON INPUT
SOURCE = 296
SLEN = 7
SDEC = 7
DEST = 5
DLEN = 8
PLEN = 8
PDIG = 7
PDEC = 2
PIC = "ZZZZ9.99##"
IER = 0
MEMORY = "################################                       00101-      UJUUU
*A      ZZZZZZZZZZ             05            10-          235787             ZZZ9
*.99                ****.9                 $$$$$V                $*****9.99
*            9,999.9             99/99/99               99999999      XXXXXXX
*XXXXXXXXXXXX      YYYYYYYYY304321020O*BCDEELSEZIFZELSE1203010DONE##############
*###########################UUUUUAZZZZZZZZZZ            000500001000-0123456*7**
*########"
PARAMETERS ON OUTPUT
MEMORY = "#### 1234.56######################                   00101-      UJUJJ
*A      ZZZZZZZZZZ             05            10-          235787             ZZZ9
*.99                ****.9                 $$$$$V                $*****9.99
*            9,999.9             99/99/99               99999999      XXXXXXX
*XXXXXXXXXXXX      YYYYYYYYY3C4321020C4BCDEELSEZIFZELSE1203010DONE##############
*###########################UUUUUAZZZZZZZZZZ            000500001000-0123456*7**
*########"
IER = 0
```

Figure 6 MOVEED Test Case that Uncovered an Error

PIC - CHARACTER array which contains the Cobol PICTURE for the edited move.

INPUT/OUTPUT DATA

MEMORY - CHARACTER data that contains the programs memory.

IER - INTEGER used as error indicator.

The numeric edited move takes data from a source field and places it in a receiving field according to what may be called a template or instructions specified in the Cobol PICTURE.

Through the course of mutation analysis two errors and redundant conditional statements were found in MOVEED. The first error detected involved a Fortran DO loop where the DO loop was being executed once when it should not be executed at all. The specific statement is:

```
DO 15 I=1,IHI
```

at line 111 in Figure 5 where IHI has been assigned the value of SDIG (number of digits in the whole part of a number) or PLDIG (number of allowable digits in the whole part of the PICTURE description). The test data that uncovered this error is shown in Figure 6.

The program was corrected and the affected lines for the new program are shown in Figure 7. The new line is the line with the Fortran statement label 11.

The second error that was uncovered by mutation analysis involved the handling of the PICTURE item 'V' which means that a decimal point is not placed in the receiving

```
11     IF(SDIG .EQ. 0 .OR. PLDIG .EQ. 0) GOTO 16          104 105
       IHI = PLDIG                                            106
       IF(SDIG .LT. PLDIG) IHI = SDIG                     107 108
       SUB1 = PLDIG + 1                                       109
       SUB2 = SOURCE + SDIG                                   110
       DO 15 I=1, IHI                                         111
       SUB1 = SUB1 - 1                                        112
       SUB2 = SUB2 - 1                                        113
       IF(MEMORY(SUB2) .EQ. '#') IER = 4                  114 115
15     SARRAY(SUB1) = MEMORY(SUB2)                            116
```

Figure 7 Corrected Program Section of MOVEED

```
TEST CASE NUMBER      1
PARAMETERS ON INPUT
SOURCE = 296
SLEN = 8
SDEC = 4
DEST = 5
DLEN = 7
PLEN = 8
PDIG = 7
PDEC = 3
PIC = "9999V999  "
IER = 0
MEMORY = "################################                 00101-      UUUJU
*A      ZZZZZZZZZZ           05             10-          235787              ZZZ9
*.99                ++++.9                 $$$$$V                $*****9.99
*          9,999.9              99/99/99                 99B99999      XXXAXXXX
*XXXXXXXXXXXX      YYYYYYYYY3040210200ABCDEELSE2IF2ELSE1200001DONE############
*##########################UUJUUAZZZZZZZZZZ            00050000100#12345678###
*#######"
PARAMETERS ON OUTPUT
MEMORY = "####1234567#####################                 00101-      UJJUU
*A      ZZZZZZZZZZ           05             10-          235787              ZZZ9
*.99                ++++.9                 $$$$$V                $*****9.99
*          9,999.9              99/99/99                 99B99999      XXXXXXXX
*XXXXXXXXXXXX      YYYYYYYYY3040210200ABCDEELSE2IF2ELSE1200001DONE############
*##########################UUJUUAZZZZZZZZZZ            00050000100#12345678###
*#######"
IER = 0
THE PROGRAM TOOK        1593 STEPS TO EXECUTE
```

Figure 8 MOVEED Test Case that Uncovered Second Error field. This error was detected from the data shown in Figure 8. As can be seen from the program in Figure 5, at statement label 80, if a V is the item in the picture, then nothing is done and control goes back to the top of the loop where the next item in the PICTURE description is retrieved. The error occurs because the pointer (variable SUB1) for the next available location in the receiving field is automatically incremented at the beginning of the loop; to correct this error subtract 1 from SUB1 when a V instruction is detected. The original method for calculating the next available location used the DO loop index and the absolute location of the destination field. This disregards the statement SUB1=SUB-1 executed when a 'V' is encountered, making it mandatory to rewrite the handling of the destination pointer. The new code is given in Figure 9. It has been indicated that some conditional statements were redundant in the original program. These have been rewritten as can be seen in Figure 9 also. Figure 5 contains the program with the 'V' error and with the redundant statements. It can be seen from this listing that several redundant conditional statements have no effect on the result of the program. These redundant statements have been taken out or rewritten as can be seen by looking at Figure 9. Specifically, a redundant conditional statement exists for statement 106 where IHI is assigned the value of PLDIG if SDIG is greater than or equal to PLDIG;

```
LISTING THE PROGRAM UNIT "MOVEED    "

      SUBROUTINE MOVEED(SOURCE,SLEN,SDEC,DEST,DLEN,PLEN,PDIG,PDEC,
     * PIC,IER)
      LOGICAL SUPRES, NEGNO
      INTEGER X(5), SUB2, SUB1, IHI, PLDIG, IVAR, I, SCOUNT, DESTHI
      INTEGER CHAR, PDISLN, SDIG, SARRAY(50), PICST, DDEC
      INTEGER STMT(3,10), CODE(30), SYMTAB(10,9)
      CHAR MEMORY(310)
      INTEGER IER
      CHAR PIC(10)
      INTEGER PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      INPUT OUTPUT MEMORY, IER
      INPUT PIC, PDEC, PDIG, PLEN, DLEN, DEST, SDEC, SLEN, SOURCE
      SUPRES = .TRUE.
      DO 5 I=1, PLEN
    5 SARRAY(I) = '0'
      PLDIG = PDIG - PDEC
      SDIG = SLEN - SDEC
      IF(SDEC .EQ. 0) GOTO 11
      SUB1 = PLDIG
      SUB2 = (SOURCE + SDIG) - 1
      DO 10 I=1, SDEC
      SUB1 = SUB1 + 1
      SUB2 = SUB2 + 1
      IF(MEMORY(SUB2) .EQ. '#') IER = 4
   10 SARRAY(SUB1) = MEMORY(SUB2)
      IF(IER .NE. 0) GOTO 101
   11 IF(SDIG .EQ. 0 .OR. PLDIG .EQ. 0) GOTO 16
      IHI = PLDIG
      IF(SDIG .LT. PLDIG) IHI = SDIG
      SUB1 = PLDIG + 1
      SUB2 = SOURCE + SDIG
      DO 15 I=1, IHI
      SUB1 = SUB1 - 1
      SUB2 = SUB2 - 1
      IF(MEMORY(SUB2) .EQ. '#') IER = 4
   15 SARRAY(SUB1) = MEMORY(SUB2)
      IF(IER .NE. 0) GOTO 101
   16 SUB1 = (SOURCE + SLEN) - 1
      CALL UNPACK(MEMORY(SUB1),X,2)
      NEGNO = X(2) .EQ. '-'
      SUB1 = DEST
      SCOUNT = 0
      DO 100 I=1, PLEN
      SUB1 = SUB1 + 1
      IF(SUB1 .GT. DLEN + DEST) GOTO 101
      CHAR = PIC(I)
      IF(PIC(I) .EQ. '9') SUPRES = .FALSE.
      IF(SARRAY(SCOUNT + 1) .NE. '0') SUPRES = .FALSE.
      IF(CHAR .NE. '-') GOTO 20
      MEMORY(SUB1 - 1) = ' '
      IF(NEGNO) MEMORY(SUB1 - 1) = '-'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '
      GOTO 100
   20 IF(CHAR .NE. '+') GOTO 30
      IF(NEGNO) MEMORY(SUB1 - 1) = '-'
      IF(.NOT. NEGNO) MEMORY(SUB1 - 1) = '+'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      IF(MEMORY(SUB1 - 2) .EQ. '+') MEMORY(SUB1 - 2) = ' '
      IF(MEMORY(SUB1 - 2) .EQ. '-') MEMORY(SUB1 - 2) = ' '
      GOTO 100
   30 IF(CHAR .NE. '$') GOTO 40
      MEMORY(SUB1 - 1) = '$'
      IF(I .EQ. 1) GOTO 100
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      IF(MEMORY(SUB1 - 2) .EQ. '$') MEMORY(SUB1 - 2) = ' '
      GOTO 100
   40 IF(CHAR .NE. '*') GOTO 50
      SCOUNT = SCOUNT + 1
      IF(.NOT. SUPRES) GOTO 99
      MEMORY(SUB1 - 1) = '*'
      GOTO 100
   50 IF(CHAR .NE. 'Z') GOTO 55
      SCOUNT = SCOUNT + 1
```

Figure 9 MOVEED Final Corrected Program Listing

```
      IF(.NOT. SUPRES) GOTO 99
      MEMORY(SUB1 - 1) = ' '
      GOTO 100
55    IF(CHAR .NE. '9') GOTO 60
      SCOUNT = SCOUNT + 1
      MEMORY(SUB1 - 1) = SARRAY(SCOUNT)
      GOTO 100
60    IF(CHAR .NE. 'B') GOTO 70
      MEMORY(SUB1 - 1) = ' '
      GOTO 100
70    IF(CHAR .NE. '/') GOTO 80
      MEMORY(SUB1 - 1) = '/'
      GOTO 100
80    IF(CHAR .NE. 'V') GOTO 91
      SUB1 = SUB1 - 1
      GOTO 100
91    IF(CHAR .NE. '.') GOTO 92
      MEMORY(SUB1 - 1) = '.'
      GOTO 100
92    IF(CHAR .NE. ',') GOTO 93
      IF(.NOT. SUPRES) MEMORY(SUB1 - 1) = ','
      IF(SUPRES) MEMORY(SUB1 - 1) = ' '
      GOTO 100
93    IER = 3
      GOTO 101
99    MEMORY(SUB1 - 1) = SARRAY(SCOUNT)
100   CONTINUE
101   RETURN
      END
```

Figure 9 cont.

MUTANT ELIMINATION PROFILE FOR MOVEED

MUTANT TYPE	TOTAL	DEAD		LIVE		EQUIV	
CONSTANT REPLACEMENT	151	146	96.7%	0	0.0%	5	3.3%
SCALAR VARIABLE REPLACEME	2430	2413	99.3%	0	0.0%	17	0.7%
SCALAR FOR CONSTANT REP.	1121	1119	99.8%	0	0.0%	2	0.2%
CONSTANT FOR SCALAR REP.	694	692	99.7%	0	0.0%	2	0.3%
SOURCE CONSTANT REPLACEME	601	599	99.7%	0	0.0%	2	0.3
ARRAY REF. FOR CONSTANT R	470	470	100.0%	0	0.0%	0	0.0%
ARRAY REF. FOR SCALAR REP	1041	1030	98.9%	0	0.0%	11	1.1%
COMPARABLE ARRAY NAME RE	148	148	100.0%	0	0.0%	0	0.0%
CONSTANT FOR ARRAY REF RE	105	105	100.0%	0	0.0%	0	0.0%
SCALAR FOR ARRAY REF REP.	694	690	99.4%	0	0.0%	4	0.6%
ARRAY REF. FOR ARRAY REF.	251	246	98.0%	0	0.0%	5	2.0%
UNARY OPERATOR INSERTION	325	318	97.8%	0	0.0%	7	2.2%
ARITHMETIC OPERATOR REPLA	218	218	100.0%	0	0.0%	0	0.0%
RELATIONAL OPERATOR REPLA	210	191	91.0%	0	0.0%	19	9.0%
LOGICAL CONNECTOR REPLACE	5	5	100.0%	0	0.0%	0	0.0%
ABSOLUTE VALUE INSERTION	399	151	37.8%	0	0.0%	248	62.2%
STATEMENT ANALYSIS	80	80	100.0%	0	0.0%	0	0.0%
STATEMENT DELETION	56	56	100.0%	0	0.0%	0	0.0%
RETURN STATEMENT REPLACEM	128	128	100.0%	0	0.0%	0	0.0%
GOTO STATEMENT REPLACEMEN	648	636	98.1%	0	0.0%	12	1.9%
DO STATEMENT END REPLACEM	76	72	94.7%	0	0.0%	4	5.3%

MUTANT STATE FOR MOVEED
FOR EXPERIMENT "MOVEED " THIS IS RUN 18

NUMBER OF TEST CASES = 65

NUMBER OF MUTANTS = 9841
NUMBER OF DEAD MUTANTS = 9503 (96.5%)
NUMBER OF LIVE MUTANTS = 0 (0.0%)
NUMBER OF EQUIV MUTANTS = 338 (3.4%)

NUMBER OF MUTANTS WHICH DIED BY NON STANDARD MEANS 4530
NORMALIZED MUTANT RATIO *****%
NUMBER OF MUTATABLE STATEMENTS = 133
GIVING A MUTANTS/STATEMENT RATIO OF 73.99

NUMBER OF DATA REFERENCES = 272
NUMBER OF UNIQUE DATA REFERENCES = 34

ALL MUTANT TYPES HAVE BEEN ENABLED

Figure 10 MOVEED Status Information after Testing

IF (SDIG .GE. PLDIG) IHI=PLDIG

but, the next statement

IF (SDIG .LT. PLDIG) IHI=SDIG

will reassign the value of IHI to SDIG if SDIG is less than PLDIG; it can be seen that the first conditional statement can be changed to the assignment statement IHI=PLDIG because it will be reassigned if the following conditional statement is true.

Another redundant conditional statement is the statement containing mutants 136 137 where the statement:

IF (I .EQ. 1 .AND. NEGNO) MEMORY(SUB1 - 1) = '-'

does not need the compound conditional portion I .EQ. 1 because the next statement takes care of that portion of the conditional. This is rewritten:

IF (NEGNO) MEMORY(SUB1 - 1) = '-'

which allows the deletion of this statement later at location 143 144.

As in the previous conditional statement, envolved with the execution of a negative picture item, the same redundant conditionals exist for the positive picture item.

The code for dealing with the Cobol floating dollar sign can be compacted for the same reason the conditionals can be rewritten in the code for the floating negative and positive signs.

The rewritten MOVEED subroutine is shown in Figure 9 and the results of the mutation testing indicate that the routine is now correct. Figure 10 contains the status information for the testing of subroutine MOVEED.

After becoming familiar with the FMS.2 system the testing sessions were easier to conduct. During the testing, an error was detected in the FMS.2 system which involved COMMON blocks where the data items had to be defined after the COMMON block statement which is oppositite of the way it should be with the declarations before the COMMON block definition. As an inexperienced user of the FMS.2 system, I had a few suggestions for the format of some user instructions which were mainly personal preferences that would not affect the systems performance. I also gained some insight for user interface for the CMS.1 system.

I found with testing that my programming style could be changed in order to avoid redundant code and unnecessary variables.

The results of the routines which were tested revealed what was believed to be true. The routines MOVENM and MOVENW proved to be correct and fully tested. The testing of subroutine MOVEED was done because it was known that it had not been fully tested and might contain some errors. The testing revealed two errors and allowed for the complete testing and generation of sufficient test data. The three routines are now tested and presumably correct; as a result of the testing, I have confidence that the routines perform as they should.

CHAPTER IV

CONCLUSION

Mutation systems have been implemented for Fortran, Lisp, and now for Cobol. Mutation analysis allows a programmer to improve his test data through an interactive process with a mutation system. Performing this iterative process allows a user to become confident that his program is correct.

CMS.1 has been implemented and operational since the Fall of 1979 with no reported problems. Several of the major routines of CMS.1 have been tested on the Fortran mutation system, FMS.2, at Yale University which increases the confidence in the Cobol system as a useful operational system for program testing of Cobol.

CMS.1 has been limited to a certain Cobol subset which should be expanded to support a wider range of typical Cobol programs. These expansions should include Cobol subroutine calls, search capabilities, and report generation. The system was designed with portability as a major consideration and also with expandability of the system in mind. A discussion of system routines and machine dependencies is given in Appendix B.

A current limitation in CMS.1 is the I/O because the input and output must be in buffered core arrays; this problem could be eliminated by redesigning the I/O handling routines to use disk direct access. This will cause some consideration of 'What is meant by correctness of output with direct access files'. Will it be required that records must be read and written in the same order as they were in the original program or should the final results be the same as the original programs final results without caring in what order the data was generated. If the requirement for correctness is the final results must be identical, then the mutant programs will have to run to completion before a comparison of the output can be made; this will slow down processing.

There are some Cobol data types which are not currently implemented in CMS.1. These data types include condition names, alphabetic type, edited alphanumeric, computational type, and index type.

It has been suggested that mutation systems might be more efficient if they could mutate compiled code instead of interpreting code. This enhancement would require the capability to decipher compiled code to determine a statements operation and the capability to alter this code. Mutating compiled code would allow for easier implementation of subroutine calls and the necessity for a SYMBOL TABLE would not be necessary. The mutation of compiled code would increase the efficiency and testing time of programs.

BIBLIOGRAPHY

[A] Allen T. Acree, CMS.1 Users Guide, July 1, 1979.

[AA] Allen T. Acree, Phd. thesis to be published Spring quarter 1980.

[ABDLS] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, Frederick G. Sayword, "Mutation Analysis".

[BDLS] T. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayword, "The Design of a Prototype Mutation System for Program Testing," Proc. 1978 NCC, AFIPS Conference Record, pp. 623-627.

[Bur] J. Burns, "The Stability of Test Data from Program Mutation Digest for the Workshop on Software Testing and Test Documentaion, Fort Lauderdale, Fla., 1978, pp. 324-334.

[Budd] Tim Budd, The Generation of Test Cases for Mutation Analysis Internal Mutation Group Memo.

[DLP] Richard A. Demillo, Richard J. Lipton, and Alan J. Perlis, Social Processes and Proofs of Theorems and Programs, Communicataions of the ACM, May 1979. Volume 22, Number 5.

[DLS1] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, April, 1978, pp. 34-41.

[DLS2] R. A. DeMillo, R. J. Lipton and F. G. Sayward, Program Mutation: A New Approach to Program Testing," INFOTECH State of the Art Report on Software Testing, Vol. 2, INFOTECH/SRA, 1979, pp. 107-127.

[S] Donald A. Sordillo, The Programmer's Ansi Cobol Reference Manual, Prentice Hall Inc., 1978.