A FORTRAN MUTATION TESTING SYSTEM *

Antonio C. Silvestri

Debra J. Richardson

COINS Technical Note TN-49

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

* As modified for the Computer and Information Science Scientific Testing Project at the University of Massachusetts from Dr. Timothy A. Budd's system.

This research was funded in part by the National Science Foundation under grant NSFDCR-84-04217.

ABSTRACT

This document describes the FORTRAN Mutation Testing System (FMTS), which evaluates a test data set for a program. The system was developed by Dr. Timothy Budd and implements the Mutation Analysis technique on programs written in a subset of FORTRAN IV. The procedure to use the FMTS at the University of Massachusetts is described and the system's output is discussed.

The Appendices contain documentation of the FMTS's major components. A description of the system's limitations and the bugs known to exist in the system are also provided.

TABLE OF CONTENTS

* Appendices A through E have been extracted from the documentation supplied with the copy of Budd's FMTS that was sent to the University of Massacusetts. Only minor revisions of this documentation were made to reflect any changes made to the original system.

1. INTRODUCTION

The FORTRAN Mutation Testing System (FMTS) is an analysis tool developed by Dr. Timothy Budd of the University of Arizona. The system measures the reliability of a given test data set for a program in terms of its ability to distinguish the program from other similar programs. In mutation analysis, a set of syntactically correct programs are generated by making small changes to the program being analyzed. These programs are called mutants and differ from the original program by exactly one of the modifications established by Budd. These mutants are executed with each element in the original program's test data set. If a mutant produces the same output as the orignal program for each test case, the mutant lives, otherwise it dies. Analysis of the live mutants for a given set of test data exposes parts of the program that have not been sufficiently tested and thereby assists the tester in the selection of additional test data.

A more extensive theoretical discussion on Mutation Testing can be found in Tim Budd's 1981 paper entitled "Mutation Analysis: Ideas, Examples, Problems, and Prospects". That paper defines the program modifications which constitute a mutation.

2. THE MUTATION SYSTEM

The FORTRAN Mutation Testing System, written by members of the Computer Science Department of the University of Arizona, implements mutation analysis. The system is currently operational in the Department of Computer and Information Science at the University of Massachusetts and can be accessed on VAX1 in directory [SDL.MUTATION].

The mutation system was originally developed under the UNIX operating system. Due to some incompatibilities between standard UNIX and EUNICE (the UNIX simulator run at UMASS), the system was modified to operate under the DEC VMS operating system.

Despite the conversion between UNIX and VMS, the UNIX conventions of standard input and output redirection and command line argument passage to a program have been retained. This was done by writing a pseudo-UNIX shell subroutine in VMS. All UNIX shell command line features have been written into the pseudo-UNIX shell with the exception of pipelining.

The system consists of five main subsystems. They are:

1. PARSE -- a program that translates your FORTRAN program into an internal representation;

2. MAKMUT -- a program that mutates the internal representation;

3. RDTEST -- a program that requests test cases and produces a file of all input data test cases;

4. MUTATE -- a program that interprets the different mutants of the user's program under all the desired test cases;

5. REPORT -- a program that displays those mutants that remain alive after interpretation.

3. TEST PROGRAM CONSTRUCTION

The system can MUTATE any standard FORTRAN IV program unit with the exceptions noted in Appendix F. These units include programs, subroutines, and functions. The system is not applicable to FORTRAN 77 program units.

Since the system determines whether a mutant lives or dies on the input-output characteristics of a program, you must inform the system which variables are input and which are output. You do this by using INPUT and OUTPUT pseudo-instructions. These instructions are not part of the FORTRAN language; they are needed by the mutation system to determine a variable's I/O mode. The list of variables that follow the pseudo-instruction INPUT indicate the variables that provide input values to the program unit; the variable list that follows the OUTPUT pseudo-instruction indicates the variables that contain the output values after exiting the program unit. In the case of a subroutine or a function, parameter variables that are not found in a pseudo-instruction list are assumed to be both input and output variables.

Another pseudo-instruction, READONLY, is used by the system to indicate that the variables in the list that follows the instruction cannot be assigned values within the program unit. As the instruction implies, the variables can only be read and are strictly INPUT variables. The READONLY attribute is used on subroutine or function parameters that are passed constants by a calling unit.

The INPUT, OUTPUT, and READONLY definitions of program variables have no effect on how the system generates a set of program mutations; the same set of mutants will be produced whether a variable is defined to be INPUT, OUTPUT or READONLY. However, these instructions do affect the mortality of a mutant. For example, a mutant that contains an assignment to a READONLY variable dies for any test case that forces execution of that assignment statement.

4. THE MUTATION PROCEDURE

To illustrate the mutation analysis procedure, consider the FORTRAN subroutine below. Note the use of INPUT and OUTPUT instructions. This routine takes the input variables 'a' and 'b' and outputs the smaller of the two through variable ic.

```
        subroutine sample(a,b,ic)
        integer a,b
        INPUT a,b
        OUTPUT ic
        if (a .gt. b) goto 10
          ic = a
          goto 20
10      continue
          ic = b
20      continue
        return
        end
```

Assuming this subroutine is in file SAMPLE.FOR, a sequence of commands are issued to ultimately obtain a report of the program mutants. These commands invoke FMTS subsystems. The order of subsystem presentation implies the usual order of execution. The discussions that follow provide the FMTS user with each subsystem's input requirements and an explanation of each subsystem's output.

Detailed subsystem information is provided in the Appendices. This information includes the general command line syntax as well as explanations of subsystem command line switches.

4.1 The PARSE Subsystem

Before any mutation analysis can be done, you must create an internal representation of a program. This internal representation is used by the FORTRAN interpreters in other subsystems. Type the following command to generate the internal form of program SAMPLE.FOR:

```
$ PARSE SAMPLE.FOR
```

PARSE outputs warnings when variables are not specifically type declared. The output for program SAMPLE.FOR is:

```
PARSE output data for program: SAMPLE.FOR

Warning, variable ic       given type integer
```

4.2 The MAKMUT Subsystem

This subsystem creates a file of the mutants of the internal representation. To invoke this subsystem, type:

```
$ MAKMUT SAMPLE.FOR
```

MAKMUT outputs a report of the number and type of mutants it generated. The MAKMUT output is:

```
Newly created mutation statistics for program: SAMPLE.FOR

Mutant maker statistics:
        38 new mutants were made.
        38 total mutants now exist.
         5 level     1 mutants made.
        11 level     2 mutants made.
        10 level     3 mutants made.
        12 level     4 mutants made.
New types made:
         3 continue replacements
         1 logical replacements
        20 absolute value and zpush insertions
         2 left, right, true and false replacements
        12 scalar replacements
Old types:
None.
```

4.3 The REPORT Subsystem

To obtain a report of the mutants that are currently in the internal representation mutant file, you must invoke the REPORT subsystem. To do this, you might type:

```
$ REPORT SAMPLE.FOR -1 100 -h SAMPLE.FOR: ALL GENERATED MUTANTS
```

This command illustrates the use of switches to modify REPORT's output. The -1 switch tells the REPORT Subsystem to output a maximum of 100 mutations for each statement. The -h switch tells REPORT to use the remaining command line as a title for the output.

Note that since no test data has yet been provided, all mutants are still live. REPORT applied at this point will list all mutants generated from the program.

Output for the above command is shown below:

```
SAMPLE.FOR: ALL GENERATED MUTANTS


     |      subroutine sample(a,b,ic)
     |      integer a,b,ic
     |      output ic
```

```
    |      input a,b
   1|      if (a.gt.b)

The live mutants of statement    1 are:

>   |      if (.true.)
>   |      if (.false.)
>   |      if (inc(a).gt.b)
>   |      if (dec(a).gt.b)
>   |      if (abs(a).gt.b)
>   |      if (-abs(a).gt.b)
>   |      if (zpush(a).gt.b)
>   |      if (a.gt.inc(b))
>   |      if (a.gt.dec(b))
>   |      if (a.gt.abs(b))
>   |      if (a.gt.-abs(b))
>   |      if (a.gt.zpush(b))
>   |      if (a.ge.b)
>   |      if (b.gt.b)
>   |      if (ic.gt.b)
>   |      if (a.gt.a)
>   |      if (a.gt.ic)

   2|     $goto 10
   3|      ic = a

The live mutants of statement    3 are:

>   |      continue
>   |      ic = inc(a)
>   |      ic = dec(a)
>   |      ic = abs(a)
>   |      ic = -abs(a)
>   |      ic = zpush(a)
>   |      a = a
>   |      b = a
>   |      ic = b
>   |      ic = ic

   4|      goto 20

The live mutants of statement    4 are:

>   |      continue

   5|   10 continue
   6|      ic = b

The live mutants of statement    6 are:

>   |      continue
>   |      ic = inc(b)
>   |      ic = dec(b)
>   |      ic = abs(b)
>   |      ic = -abs(b)
>    |       ic = zpush(b)
>    |       a = b
>    |       b = b
>    |       ic = a
>    |       ic = ic

    7|   20 continue
    8|      return
    9|      end
```

4.4 The RDTEST Subsystem

The RDTEST subsystem establishes a program's test data set. Type the following to interactively create a test data set:

```
$ RDTEST SAMPLE.FOR
```

RDTEST repeatedly prompts you for test data, one value for each input variable. Simply supply a value being requested to each prompt. Following the prompts for input values, RDTEST displays the output values produced by the original program for those input values, asks if these are the correct results, and asks if there are any more test cases. Both of these questions require yes or no responses indicated by typing a single letter 'y' or 'n'. Suppose that for SAMPLE.FOR only two test cases are input. The output from RDTEST is the echo of the responses:

```
Test Data Set construction for program: SAMPLE.FOR

Enter initial values for test case number     1.

Enter the value for a       :
4
Enter the value for b       :
2
ic       =          2
Are these the correct results?
y
Any more test cases?
y


Enter initial values for test case number     2.

Enter the value for a       :
5
Enter the value for b       :
2
ic       =          2
Are these the correct results?
y
Any more test cases?
n
```

Test data may be read from a file as an alternative to interactive input. The test data can be arranged with any number of data items per line. The data, however, must be placed in the exact order that it would be entered interactively. To read from a file, simply redirect the standard input device by using the UNIX "<" character.

A logical data entry approach is to arrange each test case's data on a single line in the datafile, along with responses to the questions that follow the prompts for input values. For example, you might create a file named SAMPLE.DAT that contains:

```
4 2 y y
5 2 y n
```

Data items in a file must be separated by one or more spaces and/or carriage returns. DO NOT use a tab to separate data; the system does not recognize it as a delimiter.

If you want RDTEST to read from this file, type the following:

```
$ RDTEST SAMPLE.FOR  <SAMPLE.DAT
```

It must be emphasized that when reading from a file, the data MUST be placed in the exact logical order that you would enter it interactively. This means when the system is expecting to read a number, a number must be the next item to read. If the system is expecting a single letter yes or no response, then a single letter 'y' or 'n' must be the next item. If there is an error in the data file, an 'END OF FILE' error will ultimately occur. The system will then delete the test case file. Your only option then is to edit the datafile and rerun RDTEST.

When reading from a file, you will still see RDTEST's prompts unless you redirect the standard output device. To redirect the standard output device, use the UNIX ">" character. You must redirect the output to the null device to eliminate the prompts. Type the following to read from the file SAMPLE.DAT and eliminate prompting:

```
$ RDTEST SAMPLE.FOR < SAMPLE.DAT > NL:
```

Note that NL: is the VMS name for the null device.

4.5 The MUTATE Subsystem

Once mutations are made and test data is initialized, the next step is to run each mutation with the test data. To do this type:

```
$ MUTATE SAMPLE.FOR -s -t 1
```

The -s switch tells MUTATE to output mutation mortality statistics to the standard output device. The -t switch instructs the MUTATE subsystem to output mutant survival information after each mutant execution. Output from MUTATE is:

```
Mutation Survival Output for program: SAMPLE.FOR

 Mutant number     1
       if (.true.)
 survived all test cases
 Mutant number     2
       if (.false.)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number     3
       if (inc(a).gt.b)
 survived all test cases
 Mutant number     4
       if (dec(a).gt.b)
 survived all test cases
 Mutant number     5
       if (abs(a).gt.b)
 survived all test cases
 Mutant number     6
       if (-abs(a).gt.b)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number     7
       if (zpush(a).gt.b)
 survived all test cases
 Mutant number     8
       if (a.gt.inc(b))
 survived all test cases
 Mutant number     9
       if (a.gt.dec(b))
 survived all test cases
 Mutant number    10
       if (a.gt.abs(b))
 survived all test cases
 Mutant number    11
       if (a.gt.-abs(b))
 survived all test cases
 Mutant number    12
       if (a.gt.zpush(b))
 survived all test cases
 Mutant number    13
       if (a.ge.b)
 survived all test cases
 Mutant number    14
        if (b.gt.b)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    15
        if (ic.gt.b)
 killed on test case     1 cause:
Error number  9 - Undefined variable used
 Mutant number    16
        if (a.gt.a)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    17
        if (a.gt.ic)
 killed on test case     1 cause:
Error number  9 - Undefined variable used
 Mutant number    18
        continue
 survived all test cases
 Mutant number    19
        ic = inc(a)
 survived all test cases
 Mutant number    20
        ic = dec(a)
 survived all test cases
 Mutant number    21
        ic = abs(a)
 survived all test cases
 Mutant number    22
        ic = -abs(a)
 survived all test cases
 Mutant number    23
        ic = zpush(a)
 survived all test cases
 Mutant number    24
        a = a
 survived all test cases
 Mutant number    25
        b = a
 survived all test cases
 Mutant number    26
        ic = b
 survived all test cases
 Mutant number    27
        ic = ic
 survived all test cases
 Mutant number    28
        continue
 survived all test cases
 Mutant number    29
        continue
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    30
```

```
       ic = inc(b)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    31
       ic = dec(b)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    32
       ic = abs(b)
 survived all test cases
 Mutant number    33
       ic = -abs(b)
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    34
       ic = zpush(b)
 survived all test cases
 Mutant number    35
       a = b
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    36
       b = b
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    37
       ic = a
 killed on test case     1 cause:
Error number -1 - results differ
 Mutant number    38
       ic = ic
 killed on test case     1 cause:
Error number  9 - Undefined variable used


Mutant Survival Statistics:


  14 died.   24 lived.  For a mortality rate of  36.8%

Individual statistics on mutants:
Results differed on         11
Undefined variable used         3
```

5. ANALYZING THE LIVE MUTATIONS

Assume that the commands discussed in the MUTATION PROCEDURE section were executed in the order presented. To obtain a report of the remaining live mutations after running the MUTATE subsystem, type:

```
$ REPORT SAMPLE.FOR -l 100 -h SAMPLE.FOR: THE SURVIVING MUTANTS
```

Output for this command would look like the following:

```
SAMPLE.FOR: THE SURVIVING MUTANTS


      |      subroutine sample(a,b,ic)
      |      integer a,b,ic
      |      output ic
      |      input a,b
     1|      if (a.gt.b)

The live mutants of statement    1 are:

>     |      if (.true.)
>     |      if (inc(a).gt.b)
>     |      if (dec(a).gt.b)
>     |      if (abs(a).gt.b)
>     |      if (zpush(a).gt.b)
>     |      if (a.gt.inc(b))
>     |      if (a.gt.dec(b))
>     |      if (a.gt.abs(b))
>     |      if (a.gt.-abs(b))
>     |      if (a.gt.zpush(b))
>     |      if (a.ge.b)

     2|     $goto 10
     3|      ic = a

The live mutants of statement    3 are:

>     |      continue
>     |      ic = inc(a)
>     |      ic = dec(a)
>     |      ic = abs(a)
>     |      ic = -abs(a)
>     |      ic = zpush(a)
>     |      a = a
>     |      b = a
>     |      ic = b
>     |      ic = ic

     4|      goto 20

The live mutants of statement    4 are:

>     |      continue
```

```
    5|   10 continue
    6|      ic = b

The live mutants of statement    6 are:

>     |     ic = abs(b)
>     |     ic = zpush(b)

    7|   20 continue
    8|      return
    9|      end
```

With the two test cases input to the system, it is clear that the statements,

```
                          ic = a
                          goto 20
```

never execute. This fact is alluded to in the report of the live mutants. Apparently, you can mutate the statement 'ic = a' to any of the mutants shown and still obtain the same output. This is a definite indication that we must add to the test data and kill these mutants.

To make additions to the test cases already established, simply invoke RDTEST again. Type:

```
$ RDTEST SAMPLE.FOR
```

Output from RDTEST follows. You will notice that the system is prompting for the third test case.

```
Enter initial values for test case number     3.

Enter the value for a        :
2
Enter the value for b        :
4
ic       =          2
Are these the correct results? y
Any more test cases? n
```

This third test case will test the statements that were neglected earlier.

We must run each mutation again with the new test data. To do this type:

```
$ MUTATE SAMPLE.FOR -s
```

Output from MUTATE is:

```
Mutation Survival Output for program: SAMPLE.FOR

Mutant Survival Statistics:


  11 died.   13 lived.  For a mortality rate of  45.8%

Individual statistics on mutants:
Results differed on         10
Undefined variable used         1
```

Now by typing:

```
$ REPORT SAMPLE.FOR
```

the following output is provided:

```
SAMPLE.FOR

     |      subroutine sample(a,b,ic)
     |      integer a,b,ic
     |      output ic
     |      input a,b
    1|      if (a.gt.b)

The live mutants of statement       1 are:

>    |      if (inc(a).gt.b)
>    |      if (dec(a).gt.b)
>    |      if (abs(a).gt.b)
>    |      if (zpush(a).gt.b)
>    |      if (a.gt.inc(b))
>    |      if (a.gt.dec(b))
>    |      if (a.gt.abs(b))
>    |      if (a.gt.zpush(b))
>    |      if (a.ge.b)

    2|     $goto 10
    3|      ic = a

The live mutants of statement       3 are:

>    |      ic = abs(a)
>    |      ic = zpush(a)

    4|      goto 20
    5|    10 continue
    6|      ic = b
```

The live mutants of statement 6 are:

```
>    |      ic = abs(b)
>    |      ic = zpush(b)

    7|   20 continue
    8|      return
    9|      end
```

Analysis of this report reveals that many of the mutants that survived the first test data set are dead following the augmentation of that test data set. The presence of the abs() mutant in the live mutant set implies that further testing requires test cases with negative numbers.

You must continue to analyze the remaining live mutants and derive additional test data to kill non-equivalent mutants. A mutant program that is equivalent to the original program will produce identical output on any input, and hence cannot be killed by additional test data. Augmenting the test data set should continue until you reach a point where additional test data does not eliminate remaining mutants. At this point, your test data set can be considered a reliable set in terms of the mutation analysis measure. There is also greater confidence that the final remaining mutants are equivalent to the original program.

USING THE FMTS

To use the system, type the following command:

```
$ @[SDL.MUTATION]DEFMUTE
```

This executes a command file, DEFMUTE.COM, that defines the following five logical symbols:

```
1. PARSE  :== "$_DRA1:[SDL.MUTATION]PARSE.EXE"
2. MAKMUT :== "$_DRA1:[SDL.MUTATION]MAKMUT.EXE"
3. RDTEST :== "$_DRA1:[SDL.MUTATION]RDTEST.EXE"
4. MUTATE :== "$_DRA1:[SDL.MUTATION]MUTATE.EXE"
5. REPORT :== "$_DRA1:[SDL.MUTATION]REPORT.EXE"
```

These logicals define the pathnames to the five main executable files that constitute the mutation system.

You can automate the mutation procedure by using batch processing. To create a full report of a typical FMTS session, a typical batch file might contain the following commands:

```
@ [SDL.MUTATION]DEFMUTE
$ COPY   SAMPLE.FOR    SAMPLE.RPT
$ PARSE  SAMPLE.FOR   >>SAMPLE.RPT
$ MAKMUT SAMPLE.FOR   >>SAMPLE.RPT
$ REPORT SAMPLE.FOR -l 1000 >>SAMPLE.RPT -h ALL MUTANTS
$ RDTEST SAMPLE.FOR < SAMPLE.DAT >>SAMPLE.RPT
$ MUTATE SAMPLE.FOR -s -t 1 >>SAMPLE.RPT
$ REPORT SAMPLE.FOR -c -l 1000 >>SAMPLE.RPT -h SURVIVING MUTANTS
```

The result of executing this batch procedure file is a report file called SAMPLE.RPT that contains the outputs of all the subsystems. To create one long report, the UNIX '>>' token was used. This token not only redirects the Standard Output File, but appends its output to a file that may already exist.

Assuming these commands are found in file MUTSAMPLE.BAT, you can initialize batch processing by typing the command:

```
$ SUBMIT MUTSAMPLE.BAT
```

APPENDIX A

PARSE - FORTRAN mutant parser

SYNOPSIS

```
PARSE [-n expname] file [-e entrypt] [-z] [-d|-di|-dc]
```

DESCRIPTION

PARSE is the parser for the FORTRAN Mutation Testing System. It produces a file containing the internal representation of the users program, including the code table, the symbol table and other information. This file is used by the other programs of the system. The parser accepts a subset of FORTRAN IV.

The FORTRAN program to be parsed is contained in file. The name of the experiment is called expname; if this option is missing, file defaults to be the experiment name. The subprogram or program that begins execution is specified by entrypt, the default is the module that is physically first in file.

The -z option lets a do-loop execute zero times (the new F77 standard). The default is the old FORTRAN convention where loops execute at least once, regardless of limit values.

The -d switches are for debugging. The -di switch prints out information about variables, such as input, output, parameter, etc. The -dc switch prints out the code as it is being generated. The -d switch does both -di and -dc.

FILES ACCESSED

file -- the file with the FORTRAN program
expname.ifm -- the internal form file

BUGS

Data statements do not work.
Equivalence statements are untested.
Do loops must have integer limits.
No double precision real or complex types.
Parameter type checking is done at run-time.

APPENDIX B

MAKMUT - FORTRAN mutant maker

SYNOPSIS

```
MAKMUT expname [procname level]*
```

DESCRIPTION

MAKMUT makes mutants as part of the FORTRAN Mutation Testing System using the internal form file produced by PARSE. MAKMUT in turn makes a file of mutant directives to be used by MUTATE and REPORT.

The name of the experiment is given by expname. Mutants may be created for specific modules within the FORTRAN program by using the options, where procname is the name of a module and level is a number from 1 through 4 which specifies the highest level of mutants to be created for the given module. The default is to create mutants through level 4 for the entry point module only. Note that the order of arguments is important.

FILES ACCESSED

expname.ifm -- the internal form file
expname.mut -- the file of mutants

APPENDIX C

RDTEST - FORTRAN mutant test case reader

SYNOPSIS

```
RDTEST expname [-c] [-f num]
```

DESCRIPTION

RDTEST makes a test case file as part of the FORTRAN Mutation Testing System, using the internal form file produced by PARSE. The -c option causes a count of the number of times each statement was executed to be printed following each sucessful test case.

The -f option sets the time limit factor. This is a mutiplying factor giving the amount of time a mutant will be allowed to execute as a factor of the amount of time the original program required on the test data. The default is 10.

RDTEST prompts the terminal for initial values for the input variables of the entry point procedure. After receiving these values from the terminal, it calls the interpreter to determine the output values, and asks the user to verify the correctness of these values. If they are correct they are written to a file along with the input values. The process is repeated if more test cases are desired. Note: repeated values for a single array may be entered as follows:

```
4*3.1
```

which would mean enter 3.1 four times. This notation can be used for part or all of the array.

The name of the experiment is given by expname.

FILES ACCESSED

expname.ifm -- the internal form file
expname.tst -- the file of test cases

BUGS

Arrays are read and printed in row major form (the right way).

Entrypoint subprograms with variable size arrays as parameters do not work.

When RDTEST prints out the results from running the program on the test data, if some of the output variables are undefined, they will print out as large numbers (i.e. 2147483647 for integers and 1e29 for reals)

APPENDIX D

MUTATE - FORTRAN mutant executer

SYNOPSIS

```
MUTATE expname [-t [nnn]] [-s]
```

DESCRIPTION

MUTATE takes files produced by MAKMUT, RDTEST, and PARSE in the FORTRAN Mutation Testing System and tries to kill off mutants using the test cases. MUTATE reads in a mutant from the mutant file, makes the alternation as described by the mutant, reads in a test case and runs the interpreter. If the mutant dies from either an internal error or incorrect results, it is discarded and a new mutant is read after which the process starts over. If the mutant succeeds and there is another test case, then the interpreter is run with the same mutant on the new test case. If a mutant succeeds all test cases, it is written to the file called expname.new. When all the mutants have finished, the file expname.mut is replaced by this file.

The name of the experiment is given by expname. The -t option controls when to be notified of which mutant is being tested. If absent, no notification is given, if present without a number following, notification is given for each mutant. If a number follows the -t, notification is given for every n'th mutant. After analysis, the status of the mutant is printed following it's number. If the -s option is given, statistics on the number and kind of mutants that died are printed.

FILES

expname.ifm -- the internal form file
expname.tst -- the file of test cases
expname.mut -- the old mutant file
expname.new -- the new mutant file with only live

BUGS

Entrypoint subprograms with variable size arrays as parameters do not work.

APPENDIX E

REPORT - FORTRAN mutant status reporter

SYNOPSIS

```
REPORT expname [-c] [-l listmax] [-d] [-h [heading]]
```

DESCRIPTION

REPORT takes files produced by MAKMUT and PARSE in the FORTRAN Mutation Testing System and produces a report of the living mutants.

The name of the experiment is given by expname. If -c is given as an option, the test cases will be printed following the REPORT. The -l option controls the maximum number of mutants to display for each statement. The default is 10. The -h option controls what to put in the heading of the REPORT. The default heading is expname. If heading is not given, it is left blank, otherwise, heading is printed in the heading. The -d option causes debugging information to be printed prior to each mutant.

FILES ACCESSED

expname.ifm -- the internal form file
expname.mut -- the old mutant file

BUGS

If lines are too long they are truncated on the left.

APPENDIX F

Known Limitations to the FMTS

1. The debugging information referred to in the Appendices has not been implemented on this system.

2. The lexical analyzer used in the PARSE subsystem is not as sophisticated as ones found in commercial FORTRAN compilers. Spaces, which the FORTRAN standard considers as having no meaning, act as token delimiters. The parser only accepts programs typed in lower-case letters. Words, such as IF, DATA, DO, etc., are given keyword status. This means you cannot use them as variable names.

3. Logical variables can be defined, however they cannot be used in logical expressions. For instance, the statement:

```
if (.not. x) goto 100
```

is not allowed. The only logical expressions that are allowed are those that contain relational operators.

4. Character constants are not allowed. Quoted strings or Hollerith constants are therefore not part of this FORTRAN implementation.

5. Subroutines or functions cannot declare arrays with variable length dimensions. Constant array lengths must be specified with each array declaration.

APPENDIX G

Bugs Known to Exist in the FMTS

1. The subroutine which takes an internal form of a statement and converts it to an ASCII string does not work on expressions that have nested parenthesis. For instance, the statement:

```
x = a+b/(2*a+3*(a+a))
```

would be output by REPORT as:

```
x = a+b/2*a+3*a+(a).
```