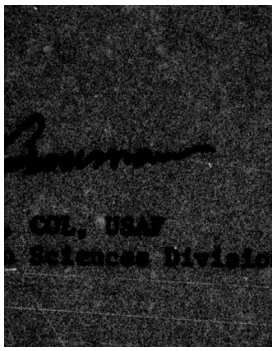
AD-A065 405 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
FORTRAN AUTOMATED VERIFICATION SYSTEM (FAVS). VOLUME I.(U)
JAN 79 R A MELTON, D M ANDREWS F30602-76-C-0436
UNCLASSIFIED RADC-TR-78-268-VOL-1 NL

| OF | ADA 065405

| OF | ADA 065405

END
DATE FILMED
4 79
DDC


COL, USAF
Sciences Divisio

(19) TR-78-268-Vol-1

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: RADC-TR-78-268, Vol I (of three)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): FORTRAN AUTOMATED VERIFICATION SYSTEM (FAVS).
Volume I.

(9) 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. Oct 76 — Feb 78

6. PERFORMING ORG. REPORT NUMBER: N/A

(10) 7. AUTHOR(s): A. Melton, M. Andrews
Rich
Dorothy

(15) 8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0436

9. PERFORMING ORGANIZATION NAME AND ADDRESS
General Research Corporation
P.O. Box 6770
Santa Barbara CA 93111

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
63701B
(16) 32010320
(17) 03

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (ISIE)
Griffiss AFB NY 13441

(11) 12. REPORT DATE: January 1979

13. NUMBER OF PAGES: 49

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Same
(12) 50 p.

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

DDC
RECEIVED
MAR 8 1979
B

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Frank S. Lamonica (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Computer Software
Software Testing
Software Verification
Software Documentation
FAVS
Automated Verification System

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The FORTRAN Automated Verification System (FAVS) is intended to reduce the cost of assuring that software systems written in FORTRAN are comprehensively tested. It consists of automated algorithms and techniques for verifying the testing of FORTRAN software. FAVS supports testable programming in FORTRAN, augments the static error detection performed by FORTRAN compilers, automates the measurement of testing effectiveness, assists the manual design and selection of test cases, and increases the mechanization of certain aspects → next page
(Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

402 754

79 03 05 040

Item 20 (Cont'd)

of software system maintenance. FAVS is a series of tools which provide (1) translation from DMATRAN (a structured extension of FORTRAN) to FORTRAN and from FORTRAN to DMATRAN, (2) static detection of unreachable statements, set/use errors, mode-conversion errors, and external reference errors, (3) a means of measuring the effectiveness of test cases, (4) assistance in the construction of test data that will thoroughly exercise the software, and (5) automated documentation. FAVS has been implemented for the analysis of computer software written in the FORTRAN V or DMATRAN language and is operational on the HIS-6180 GCOS and MULTICS computer systems at Rome Air Development Center at Griffiss AFB, New York, the UNIVAC 1100/42 computers at DMATC in Washington, DC, and DMAAC in St. Louis, Missouri, and the CDC 6400 computer at General Research Corporation in Santa Barbara, California, where it was developed.

ACKNOWLEDGEMENTS

Many individuals contributed to the design and implementation of FAVS. E. F. Miller, Jr., originated the work at General Research Corporation which resulted in the methodology for FAVS. He, together with Michael Paige, Jeoff Benson, Randy Urban, Rich Melton, Carolyn Gannon, Dick Wisehart, and others, initially built RXVP, an automated verification system for FORTRAN, a product of the Program Validation Project sponsored by General Research Corporation.

JAVS (JOVIAL automated verification system) was the immediate successor to RXVP. In its development, the RXVP software testing methods were examined, the algorithms were extended to JOVIAL constructs, and the JAVS software itself was written in the JOVIAL language. JAVS was installed at RADC, and user training was conducted. The major contributors to the JAVS project were Jeoff Benson, Nancy Brooks, Carolyn Gannon, E. F. Miller, Jr., Ray Stone, Randy Urban, and Dick Wisehart, all employees (at that time) of General Research Corporation. The RADC Project Engineer for JAVS was Dick Robinson.

STRUCTRAN-1 and STRUCTRAN-2 were developed concurrently with the JAVS project. STRUCTRAN-1 translates DMATRAN (a structured extension of FORTRAN) to FORTRAN, and STRUCTRAN-2 translates FORTRAN to DMATRAN. The STRUCTRAN software was installed at DMAAC in St. Louis, Missouri. The major contributors to STRUCTRAN were Dorothy Andrews, Rich Melton, and Randy Urban. The RADC Project Engineer for STRUCTRAN was Don Mark.

The development of FAVS integrated concepts from RXVP with STRUCTRAN-2, incorporated certain capabilities of JAVS, extended the STRUCTRAN-2 capabilities, and improved STRUCTRAN-1. The FAVS software is written in DMATRAN. FAVS has been installed at DMAAC, DMATC, and RADC, and user training and maintenance training have been conducted. The major contributors to FAVS were Dorothy Andrews, Carolyn Gannon, Rich Melton, and Randy Urban. The RADC Project Engineer was Frank LaMonica.

CONTENTS

ILLUSTRATIONS

EVALUATION

The purpose of this effort was to provide the Aerospace and Topographic Centers of the Defense Mapping Agency (DMA) with a capability to test and verify FORTRAN software. A software tool, designated the FORTRAN Automated verification System (FAVS), meeting DMA's requirements was developed. It was installed and acceptance tested on the UNIVAC 1100/42 Computer System at each of the two DMA centers. FAVS will be utilized internally by DMA for testing, verifying and documenting vender supplied software.

Frank S LaMonica

FRANK S. LAMONICA
Project Engineer

1 INTRODUCTION

As part of its program for applying advanced technology to improve the quality and reliability of software, and to provide testing tools for the Defense Mapping Agency, Rome Air Development Center contracted with General Research Corporation for the design, development, installation, and documentation of a FORTRAN Automated Verification System (FAVS). This system is intended to reduce the cost of assuring that software systems written in FORTRAN are comprehensively tested. The work involved the application of practical and automatable algorithms and techniques to the verification of FORTRAN software testing. The specific tasks were to engineer workable and efficient ways to support testable programming in FORTRAN, augment the static error detection performed by FORTRAN compilers, automate the measurement of testing effectiveness, assist in the manual design and selection of test cases, and increase the mechanization of certain aspects of software system maintenance.

This report (the final report for the project) describes the fundamentals and application of a method for systematically and comprehensively testing computer software. FAVS is a series of tools which provide:

- Translation from DMATRAN (a structured extension of FORTRAN) to FORTRAN and from FORTRAN to DMATRAN
- Static detection of unreachable statements, set/use errors, mode-conversion errors, and external reference errors
- A means of measuring the effectiveness of software test cases, both individually and cumulatively
- Assistance in the construction of test data that will thoroughly exercise the software
- Automated documentation

FAVS has been implemented for the analysis of computer software written in the FORTRAN V or DMATRAN language and is operational on the HIS-6180 GCOS and MULTICS computer systems at the Rome Air Development Center (RADC), Griffiss AFB, New York, the UNIVAC 1100/42 computers at DMATC in Washington, D.C., and DMAAC in St. Louis, Missouri, and the CDC 6400 computer at General Research Corporation in Santa Barbara, California, where is was developed.

Section 2 of this report summarizes the FAVS tools. Section 3 describes methods for the effective utilization of FAVS in single-module testing, system testing, and software documentation.

In addition to this report, a number of other reports have been prepared as part of this effort:

- FAVS (FORTRAN Automated Verification System) User's Manual (CR-1-754, May 1978)
  This report is an introduction to using FAVS in the testing process. Its purpose is to acquaint the user with the application of FAVS to program testing, so that an efficient approach to program verification can be taken. The basic commands by which FAVS provides this assistance are discussed in detail. FAVS processing is described in the order normally followed by the beginning FAVS user. The Appendixes include a summary of all FAVS commands and a description of FAVS operation at RADC, DMATC, and DMAAC with both sample command sets and sample job control statements.

- DMATRAN User's Guide (CR-1-673/1, January 1978)
  This report describes the structured constructs and syntax of DMATRAN, a structured extension to FORTRAN. It also details the use of the DMATRAN preprocessor, which translates DMATRAN into FORTRAN. Procedures for using the UNIVAC 1110 or the Honeywell-6180 version of DMATRAN are included.

- FAVS (FORTRAN Automated Verification System) Computer Program Documentation: Vols. 1, 2, 3 (CR-2-754, January 1978)
  These reports describe the FAVS software design, the organization and contents of the FAVS data base, and for each FAVS component its function, each of its invokable modules, and the global data structures it uses. The report is intended for use in FAVS software maintenance, together with the Software Analysis reports described below.

- FAVS Computer Program Documentation: Vol. 4, Software Analysis
  This volume is a collection of computer output produced by FAVS, not reproduced but on file at RADC. The source code for each component of the FAVS software has been analyzed by FAVS itself to produce enhanced source code listings of FAVS with indentation and control structure identification, inter-module dependence, all module invocations, module control structure, and a cross reference of symbol usage. This volume is intended to be used with Vols. 1-3 for FAVS software maintenance. It is itself also an excellent example of the use of FAVS for computer software documentation.

2 OVERVIEW OF DMATRAN AND FAVS

This section provides a summary of the DMATRAN language extensions to support testable programming in FORTRAN, and describes the commands processed by FAVS. DMATRAN and FAVS are implemented in three software tools.

- DMATRAN precompiler. DMATRAN constructs are automatically indented on the source listing produced by the DMATRAN compiler. All DMATRAN statements are translated into standard FORTRAN, and standard FORTRAN statements are passed unmodified to the "intermediate source" file, which is then compiled by the FORTRAN compiler.

- FAVS Processing. Favs provides static analysis, code restructuring, instrumentation, testcase data generation assistance, retesting and documentation assistance for FORTRAN and DMATRAN programs. User-supplied commands control the FAVS processing.

- FAVS Analyzer. After the execution of software that has been instrumented by FAVS, the FAVS Analyzer provides reports describing the extent of testing coverage obtained. User-supplied commands, similar to FAVS commands, control the Analyzer's processing.

The DMATRAN constructs are described in Sec. 2.1; the use of the DMATRAN precompiler is fully described in the DMATRAN User's Guide. The FAVS processing and Analyzer commands are summarized in Secs. 2.2 and 2.3; they are described in detail in the FAVS User's Manual.

2.1 DMATRAN CONTROL CONSTRUCTS

DMATRAN extends the FORTRAN programming language with five control constructs that replace FORTRAN controls. These statement forms can be intermixed with ordinary FORTRAN non-control statements in the text which is processed by the DMATRAN precompiler. DMATRAN statements are converted by the precompiler to equivalent FORTRAN statements, and the resulting file can be compiled by the FORTRAN compiler in the normal manner.

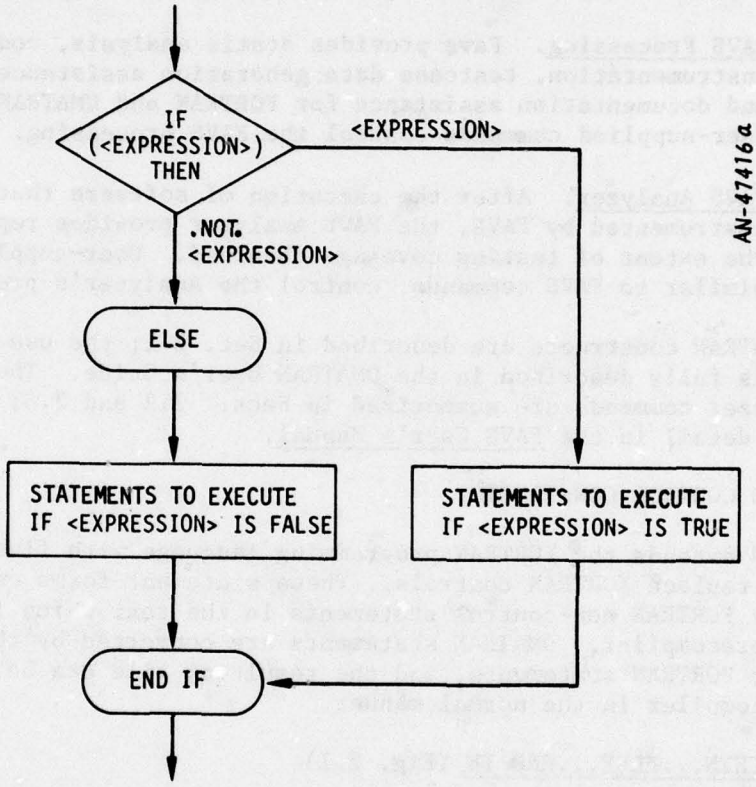
2.1.1 IF...THEN...ELSE...END IF (Fig. 2.1)

This construct provides block structuring of conditionally executable statements. If <expression> is true, control transfers to the first statement within the block; if false, to the next statement after the END IF. The ELSE statement is optional. If it is present and <expression> is false, the statements following the ELSE are executed.

2.1.2 DO WHILE...END WHILE (Fig. 2.2)

This construct defines a repetitive operation which is to be performed zero or more times. If <expression> is true, the statements within the block are executed; if false, the next statement after the END WHILE. After the statements within the block have been executed, the value of <expression> is

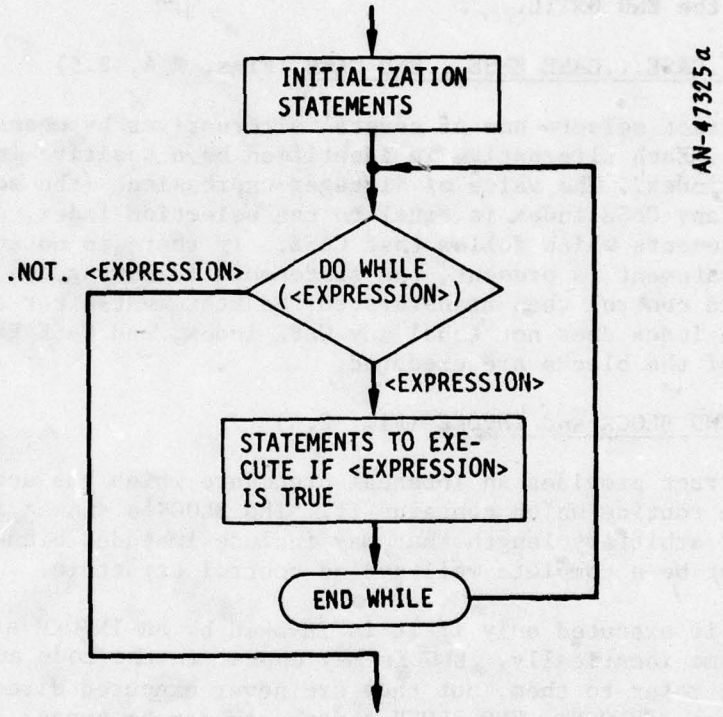
```
IF  (<EXPRESSION>) THEN
.
.   STATEMENTS TO EXECUTE IF <EXPRESSION> IS TRUE
.
ELSE
.
.   STATEMENTS TO EXECUTE IF <EXPRESSION> IS FALSE
.
END IF
```




```
FUNCTION SINC( X )
IF ( X .EQ. 0 ) THEN
.  SINC = 1.
ELSE
.  SINC = SIN(X)/X
END IF
RETURN
END
```

Figure 2.1. IF...THEN...ELSE...END IF Construct

```
INITIALIZATION STATEMENTS
DO WHILE (<EXPRESSION>)
.
.       STATEMENTS TO EXECUTE IF <EXPRESSION> IS TRUE
.
END WHILE
```




```
FUNCTION SQRT( A )
X = A
DO WHILE( ABS(X-A/X) .GT. 1.E-6 )
      X = (X+A/X)/2
END WHILE
SQRT = X
RETURN
END
```

Figure 2.2. DO WHILE...END WHILE Construct

checked again. Note that the iteration variable that controls the value of <expression> must be explicitly initialized before the loop is entered, and must be explicitly modified on each pass through the loop.

2.1.3 DO UNTIL...END UNTIL (Fig. 2.3)

This construct is like a FORTRAN DO loop in that it is performed at least once and has its exit at the bottom of the loop; otherwise it is like the DO WHILE. After the first execution of the statements in the block, <expression> is evaluated; if it is false, the block is executed again, and so on until <expression> is true. At that time control transfers to the next statement after the END UNTIL.

2.1.4 CASE OF...CASE...CASE ELSE...END CASE (Figs. 2.4, 2.5)

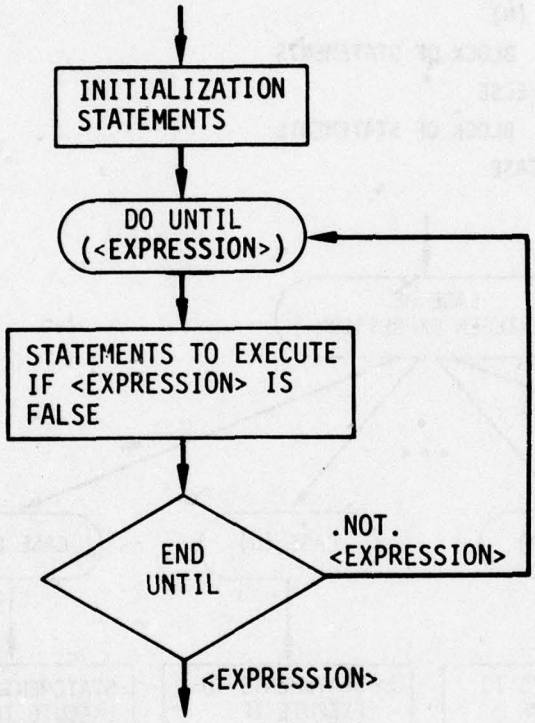
This construct selects one of several alternatives by means of a computed selection index. Each alternative is identified by a positive integer constant called the CASE index. The value of <integer expression> (the selection index) is computed; if any CASE index is equal to the selection index, control transfers to the statements which follow that CASE. If there is no such CASE, and the CASE ELSE statement is present, the statements following the CASE ELSE are executed, and control then transfers to the statement after the END CASE. If the selection index does not equal any CASE index, and CASE ELSE is not provided, none of the blocks are executed.

2.1.5 BLOCK...END BLOCK and INVOKE (Fig. 2.6)

This construct provides an internal procedure which has access to all variables in the routine which contains it. The BLOCK's <name> is a string of characters of arbitrary length that may include imbedded blanks. The body of the BLOCK must be a complete well-nested control structure.

The BLOCK is executed only if it is invoked by an INVOKE statement that refers to its name identically. BLOCKs may appear in the code anywhere after all INVOKEs that refer to them, but they are never executed directly (by falling into them). BLOCK...END BLOCK constructs can be nested, but not recursive (i.e., a BLOCK may not directly or indirectly invoke itself). A BLOCK cannot be invoked from an external routine, nor can it be passed as a parameter to another routine.

```
INITIALIZATION STATEMENTS

DO UNTIL (<EXPRESSION>)
.
.     STATEMENTS TO EXECUTE IF <EXPRESSION> IS TRUE
.
END UNTIL
```




```
FUNCTION CONVRG(XINIT, EPS, F )
EXTERNAL F
X = XINIT
DO UNTIL (ABS(X-XOLD).LE.EPS)
.     XOLD = X
.     X = F(X)
END UNTIL
CONVRG = X
RETURN
END
```

Figure 2.3. DO UNTIL...END UNTIL CONSTRUCT

```
CASE OF (<INTEGER EXPRESSION>)
CASE (I)
.     BLOCK OF STATEMENTS
CASE (J)
.     BLOCK OF STATEMENTS
                ⋮
CASE (N)
.     BLOCK OF STATEMENTS
CASE ELSE
.     BLOCK OF STATEMENTS
END CASE
```

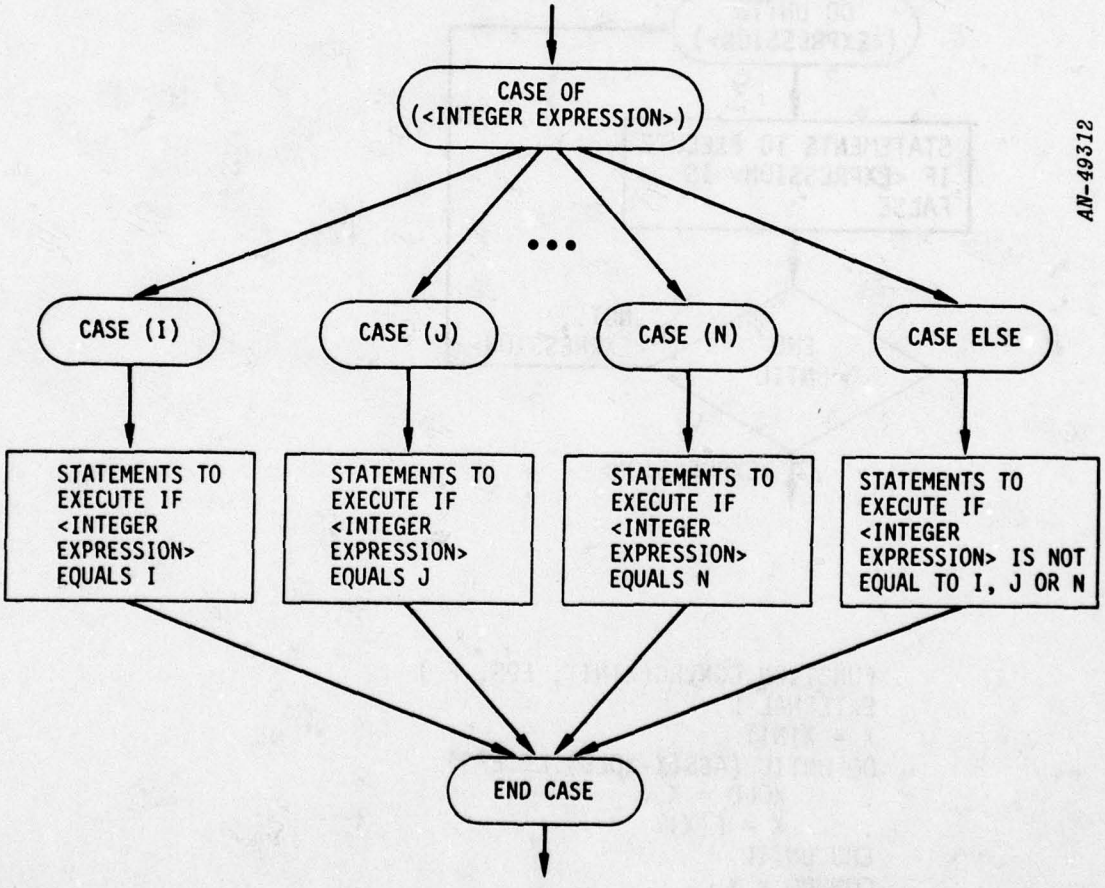
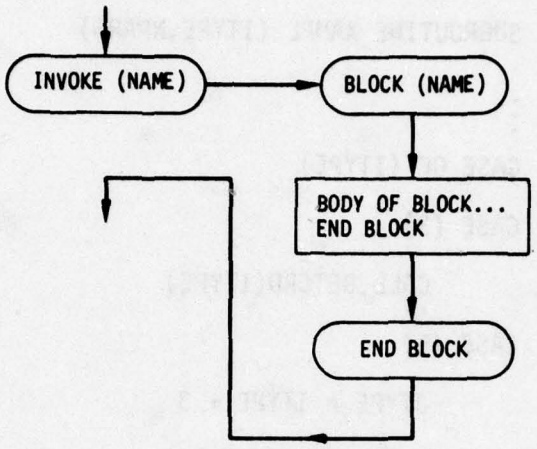


Figure 2.4. CASE OF..CASE..CASE ELSE...END CASE CONSTRUCT

```
SUBROUTINE XAMPL (ITYPE,NPARS)
:
CASE OF (ITYPE)
CASE (3)
      CALL GETCRD(ITYPE)
CASE (5)
      JTYPE = ITYPE + 3
      CALL STRUCT(JTYPE)
CASE (9)
      CALL IBALPR(ITYPE,NPARS)
CASE ELSE
      CALL ERROR
END CASE
RETURN
END
```

Figure 2.5. DMATRAN Example of CASE Construct

```
INVOKE (<NAME>)
BLOCK (<NAME>)
.
.     BLOCK STATEMENTS
.
END BLOCK
```




```
SUBROUTINE MLTPLY(A,B,C,N)
DIMENSION A(10,10),B(10,10),C(10,10)
I = 1
DO WHILE ( I .LE. N )
.     J = 1
.     DO WHILE ( J .LE. N )
.     .     INVOKE ( COMPUTE NEW ARRAY ELEMENT )
            J = J + 1
.     END WHILE
      I = I + 1
END WHILE
BLOCK ( COMPUTE NEW ARRAY ELEMENT )
.     S = 0.0
.     K = 1
.     DO WHILE ( K .LE. N )
.     .     S = S + A(I,K) * B(K,J)
            K = K + 1
.     END WHILE
.     C(I,J) = S
END BLOCK
RETURN
END
```

Figure 2.6. BLOCK...END BLOCK and INVOKE Construct

2.2 FAVS PROCESSING

FAVS is a software system which reads as data the user's FORTRAN or DMATRAN source code. The type of processing to be performed on the source code is specified through commands that are input to FAVS. During an initial run, a restart file is constructed which contains information about each module submitted for analysis. FAVS has several components which extract information from this file and produce reports. Figure 2.7 illustrates the basic elements of a FAVS analysis.

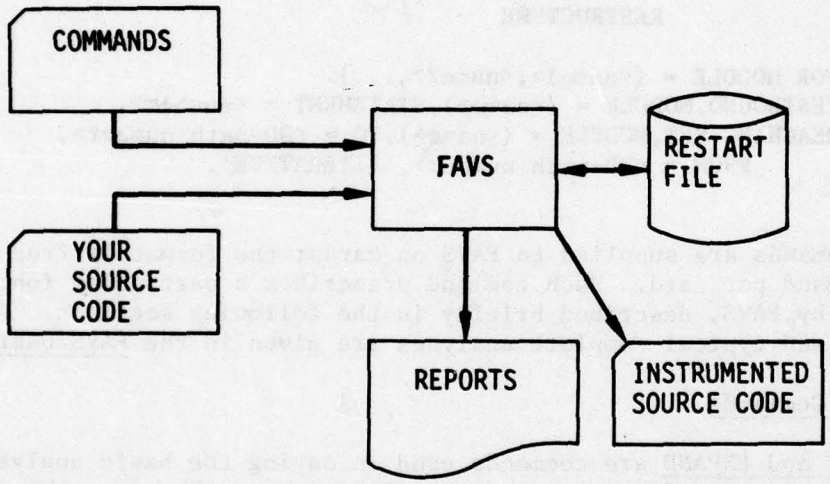


Figure 2.7. FAVS Analysis

There are eight basic FAVS commands:

```
RESTART.
EXPAND.
LANGUAGE=DMATRAN.
FILE,PUNCH=<file number>.
OPTIONS=<list>.
```

<list> may contain one or more of the following options, separated by commas:

```
LIST
DOCUMENT
SUMMARY
STATIC
INSTRUMENT
INPUT/OUTPUT
REACHING SET
RESTRUCTURE
```

```
FOR MODULE = (<name1>,<name2>,...).
TESTBOUND,MODULE = (<name>),STATEMENT = <number>.
REACHING SET,MODULE = (<name>),TO = <DD-path number>,
      FROM = <DD-path number>, {ITERATIVE}.
```

The commands are supplied to FAVS on cards; the format is free form, with one command per card. Each command prescribes a particular function to be performed by FAVS, described briefly in the following sections. Detailed descriptions and typical complete analyses are given in the FAVS User's Manual.

2.2.1 Setup Commands

RESTART and EXPAND are commands used in saving the basic analysis information for a set of modules from one FAVS run to the next, thus saving execution time when a large number of modules is being analyzed more than once. As a set of modules is processed (using any of the OPTIONS), a restart file is created. If this file is saved, it can be used in subsequent FAVS runs which further analyze the same modules (using other OPTIONS) by supplying either RESTART or EXPAND as the first FAVS command. EXPAND performs the same functions as RESTART and also permits additional modules to be added to the restart file.

LANGUAGE = DMATRAN sets up FAVS to accept source code that includes DMATRAN constructs. The command is not necessary for FORTRAN source code, since the default setup is to process FORTRAN.

FILE,PUNCH = <file number> is used to change the file to which FAVS outputs an enhanced source code when either INSTRUMENT, INPUT/OUTPUT, or RESTRUCTURE options are selected.

FOR MODULE = (<name1>, <name2>,...) causes single-module reports to be produced only for a specified set of modules, rather than for every module.

2.2.2 Processing Commands

The command which controls the type of processing to be done by FAVS is:

OPTION(S) = <list>

where <list> contains one or more of the option names, separated by commas. The following sections briefly describe each option except for RESTRUCTURE. Details and additional examples of the reports and all options are given in the FAVS User's Manual.

LIST. This option causes an enhanced source listing of every module to be produced. The listing has indentation of control constructs, defines the logical structure in terms of DD-Paths, and is the reference for line numbers used by the other reports.

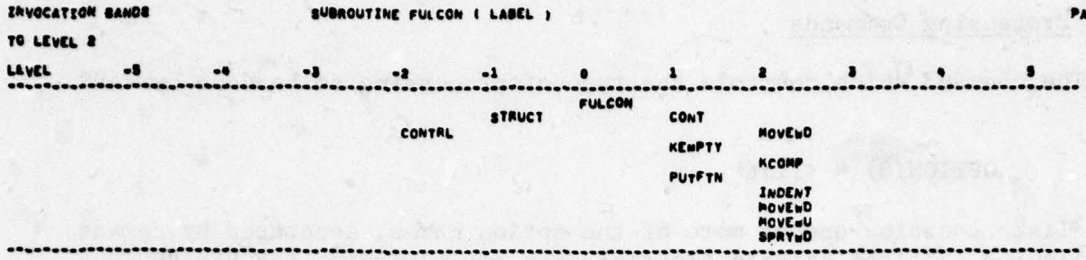
DOCUMENT. This option causes six reports to be produced which describe the system of modules ("library") being analyzed. For each module separately, three reports are produced:

- READS Report: lists each READ statement, with its associated format.
- INVOKES Report: lists all calls to externals (in the library or not), and all calls from other modules in the library.
- BANDS Report: Locates the module in the invocation hierarchy by showing an upward and downward calling tree (Fig. 2.8).

For the system as a whole, three reports are produced:

- COMMONS Report: displays two matrixes that show the location of all references to common blocks (Fig. 2.9).
- CROSS REF Report: cross-references all variable names in the library, giving a list of all modules that reference each variable, and a list of statement numbers where referenced in each module with a marker on statements where the variable is set.
- LIBRARY DEPENDENCE Report: shows the invocation structure of the library in a matrix format (Fig. 2.10).

SUMMARY. This option produces an abbreviated library documentation consisting of three reports: a statement profile (Fig. 2.11), and the COMMONS Report and LIBRARY DEPENDENCE Report just described.



This report shows the selected module within the invocation hierarchy. At the center is the selected module. Each successive band of modules from the center to the left shows the calling modules; each successive band to the right shows the called modules. The left (calling) modules reside on the library; the right (called) modules can include modules external to the library.

Figure 2.8. BANDS Report

STATIC. This option causes "static analysis" of the software being analyzed: that is, analysis of the source code, without compiling and running it.

- Mode checking which identifies possible misuse of constants and variables in expressions, assignments, and invocations.
- Invocational checking which validates actual invocations against formal declarations, checking for consistency in number of parameters and type.
- Set and use checking which uncovers possible use-before-set conditions and similar program abnormalities.
- Graph checking which identifies possible errors in program control structure such as unreachable code.

```
LIBRARY COMMON BLOCK MATRIX

------------------------.---------
C **         *          .        *
O * * MODULE * C C E F K.M M P S *
M *  *       * O O X U E.A C U T *
M *   *      * N N A L M.I V T R *
O *    *     * T T M C P.N E F U *
N *     *    * R   P O T. * T C *
  *      *   * L   L N Y. C N T *
N *       *  *          .        *
O * COMMON * *          .        *
  *         **          .        *
------------------------.---------
  1 * ACCTNG * O        .        *
  2 * CARDS  * X       O.        *
  3 * CONSTN * O X     X.    X X *
  4 * FORTRN * O X      .    X O *
  5 * INTERN * X     O  .    X O *
  6 * INVOKE * C        .      X *
  7 * RECNIZ * O        .        *
  8 * SESE   * C        .        *
  9 * STACK  * O        .      X *
 10 * STATE  * X     X  .    O X *
 11 * STYPE  * O        .      O *
 12 * TRACE  * X        .    X   *
 13 * USEOPT * X O     O.    X X *
 14 * WARNIN * O        .        *
------------------------ ---------
```

```
LIBRARY COMMON SYMBOL MATRIX

------------------------.---------
C **         *          .        *
O * * MODULE * C C E F K.M M P S *
M *  *       * O O X U E.A C U T *
M *   *      * N N A L M.I V T R *
O *    *     * T T M C P.N E F U *
N *     *    * R   P O T. W T C *
  *      *   * L   L N Y. C N T *
N *       *  *          .        *
O * SYMBOL * *          .        *
  *         **          .        *
------------------------.---------
  2 * IECF   * U       O.        *
 13 * INDON  * O O     O.    U O *
  9 * INSTAK * C        .      X *
 10 * ITYPE  * U     O  .    O U *
 A4 * KABEL  * C X      .    U O *
  4 * KENGTH * O S      .    X O *
 A4 * KFTN   * O U      .    U U *
 13 * KOMFTN * O O     O.    U U *
  5 * KSTMT  * U     O  .    U O *
A10 * LABEL  * O     X  .    O U *
  3 * LBK    * C U     U.    U U *
 10 * LENGTH * S     O  .    C O *
 10 * LINBEG * U     O  .    O O *
 10 * LINEND * U     U  .    O O *
A10 * LIST   * O     U  .    O U *
 10 * LPOINT * O     U  .    O U *
 A5 * LSTACK * O        .      X *
 10 * LTYPE  * X     O  .    O U *
 13 * LUNFOR * O O     O.    U O *
 13 * LUNOUT * U O     O.    O U *
 10 * MENGTH * O     O  .    O U *
 12 * NALTER * C        .    U   *
 A6 * NAMEL1 * O        .      U *
  5 * NFATER * U     O  .    O O *
 13 * NINDNT * O O     O.    X O *
  5 * NLINES * S     O  .    O O *
  6 * NOBE   * O        .      X *
 A6 * NOBLOK * O        .      S *
 A6 * NOINV  * O        .      X *
 10 * NSTATE * X     O  .    O O *
------------------------ ---------
```

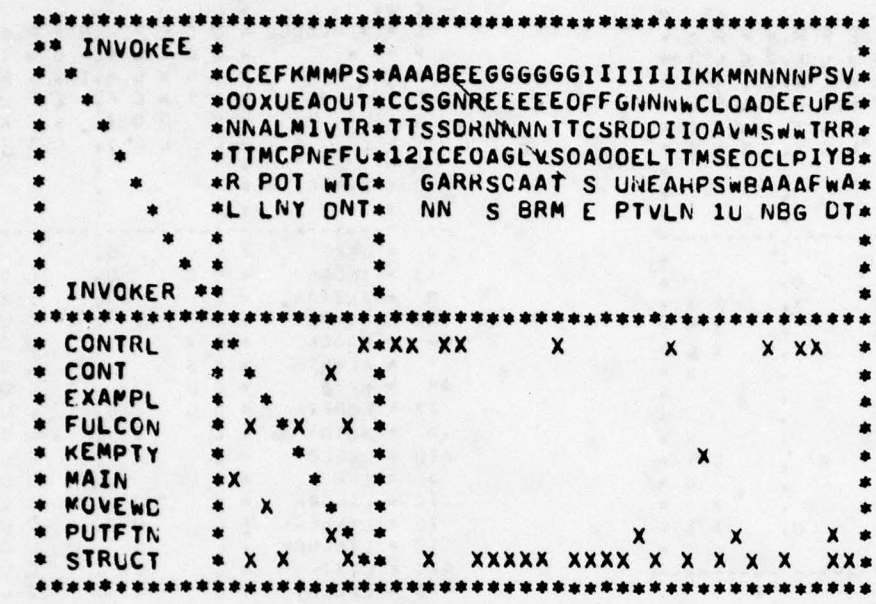
```
LEGEND
--------------------------------------

COMMONS VS. MODULES
-------------------
X => AT LEAST ONE SYMBOL REFERENCED
O => NO SYMBOL EVER REFERENCED

SYMBOLS VS. MODULES
-------------------
X => SYMBOL SET AND USED
O => SYMBOL NEVER SET OR USED
S => SYMBOL SET ONLY
U => SYMBOL USED ONLY
E => SYMBOL EQUIVALENCED (OVERLAID) ONLY
A => SYMBOL IS AN ARRAY
```

Two matrices are produced by this report. The first one lists all common blocks encountered in any module in the library, and indicates which blocks do and do not contain any symbols used by each module. Routines from which a common block may safely be removed are easily found. This matrix assigns a number to each common block.

The second matrix lists only the symbols which are used by some module; the number of the common block in which each is found is printed to the left. This report is an excellent aid when changes are being made in a software system.

Figure 2.9. COMMONS Report

LIBRARY DEPENDENCE



THE FOLLOWING MODULES ARE NOT INVOKED BY ANY MODULE ON THE LIBRARY

MAIN

THE FOLLOWING MODULES DO NOT INVOKE ANY MODULE ON THE LIBRARY

EXAMPL KEMPTY

The interaction of all modules in the library is shown in the first matrix. If the library contains all modules in the user's program, this report provides a concise, complete picture of the module dependencies. If the library contains only a part of the program, this report aids in determining what modules do not interact with the part and might be better suited for another part. The modules are listed in alphabetical order.

The second matrix shows external modules, not in the library. If the library contains all modules in the program, the external modules will consist only of system routines. If the library contains only a part of the program, this report shows the part's interfaces with other parts.

This report also identifies the "top" and "bottom" modules in the system--those that are not invoked by, and those that do not invoke, any other module.

Figure 2.10. LIBRARY DEPENDENCE Report

STATEMENT PROFILE SUBROUTINE EXAMPL (INFO, LENGTH)

INTERFACE CHARACTERISTICS

ARGUMENTS	2
ENTRY	1
EXIT	1
INTERNAL PROCEDURES	2
INVOKES	4
WRITE	1

STATEMENT CLASSIFICATION	STATEMENT TYPE	NUMBER	PERCENT
DECLARATION...			
	FORMAT	1	2.8
	TOTAL	1	2.8
EXECUTABLE...			
	ASSIGNMENT	4	11.1
	CALL	1	2.8
	CASE	2	5.6
	CASEELSE	1	2.8
	DOUNTIL	1	2.8
	ELSE	1	2.8
	ENDBLOCK	2	5.6
	ENDCASE	1	2.8
	ENDIF	2	5.6
	ENDWHILE	2	5.6
	END	1	2.8
	INVOKE	3	8.3
	RETURN	1	2.8
	WRITE	1	2.8
	TOTAL	23	63.9
DECISION...			
	BLOCK	2	5.6
	CASEOF	1	2.8
	DOWHILE	2	5.6
	ENDUNTIL	1	2.8
	IFTRAN-IF	2	5.6
	SUBROUTINE	1	2.8
	TOTAL	9	25.0
DOCUMENTATION...			
	COMMENT	3	8.3
	TOTAL	3	8.3

* TOTAL PERCENTAGE MAY BE MORE THAN 100 BECAUSE OF OVERLAPPING CLASSIFICATIONS

This report classifies each statement of a module as either a declaration, executable, decision, or documentation statement. Under these classifications, a tabulation of the subtypes is listed.

Figure 2.11. Statement Profile

A rigorous analysis of program variables, including inter-module checking, uncovers subtle inconsistencies which lead to errors, such as:

- The number of parameters in a subroutine or function call does not agree with those of the routine called.
- The mode of an actual parameter does not match that of the corresponding formal parameter.
- A parameter is listed in the calling argument list as a non-subscripted variable but is used in the routine as an array.

Another consistency check is performed on the structure of the program. The graph for each module is checked to see that all statements are reachable from the module's entry and that the module's exit is reachable from each statement. Unreachable statements represent extra overhead in terms of memory space required for a module, while statements from which the exit cannot be reached represent potentially catastrophic system failures.

The output produced by this option is a Static Analysis report for each module (Fig. 2.12).

INSTRUMENT. This option "instruments" the source code by inserting "probe statements" at the entry and exit of each module and at each statement which begins a DD-path. Each probe includes a call to a data collection routine which records information concerning the flow of control when the software is executed. A special probe is inserted at the end of the main program to signal the end of test execution. The user can also have this special probe inserted at other points in the code, which has the effect of breaking one test execution into multiple test cases. The command TESTBOUND, MODULE=(<name>),STATEMENT=<number>. causes the special probe to be inserted after the statement specified. The instrumented source-code file can be input to the FORTRAN compiler (after first being processed by the DMATRAN preprocessor if DMATRAN is being used). The instrumented object code is then ready for loading and test execution.

During execution of the instrumented program, the probes record (on the LTEST file) a summary of execution data which result from processing the set of test cases input for this run.

INPUT/OUTPUT. Additional information may be gathered during test execution by inserting INPUT and OUTPUT statements into each module. The INPUT statements list the global variables (either parameters or in common) that are required to have a value whenever the module is invoked; the OUTPUT statements list variables that will be assigned values in the module. An INPUT variable may also be an OUTPUT variable. The INPUT/OUTPUT option provides for tracing the values of the variables during execution, by translating the INPUT and OUTPUT statements into comments followed by data-collection code that is inserted in the FORTRAN or DMATRAN (along with the DD-Path probes if INSTRUMENT is also selected). When the program is executed, the entry and exit values of the variables will be reported.

```
STATIC ANALYSIS                SUBROUTINE CIRCLE ( AREA )

 SEQ NEST SOURCE                                                    UNKNOWN EXTERNALS
-------------------------------------------------------------------------------------
   1              SUBROUTINE CIRCLE ( AREA )
   2              COMMON / VALUES / DIAMTR
   3              INTEGER AREA
   4              RADIUS = DIAMTR / 2
   5              AREA = PI * RADIUS ** 2
                  ---------------------------------------------------------------
                  -                        MODE WARNING                         -
                  - LEFT HAND SIDE HAS MODE INTEGERRIGHT HAND SIDE HAS MODE REAL -
                  ---------------------------------------------------------------
   6              IF ( AREA .GT. 50 ) THEN
   7 ( 1)         .  CALL PRNT ( AREA )
                  ---------------------------------------------------------------
                  -                        CALL ERROR                           -
                  -       PRNT      CALLED WITH  1 ACTUALLY HAS  2 ARGUMENTS    -
                  ---------------------------------------------------------------
                  ---------------------------------------------------------------
                  -                        CALL ERROR                           -
                  -PARAMETER   1 OF PRNT      .ACTUAL PARAMETER HAS MODE INTEGER -
                  -                           .FORMAL PARAMETER HAS MODE REAL    -
                  ---------------------------------------------------------------
   8              ENDIF
   9              RETURN
  10              CALL STACK ( RADIUS, AREA )
                  ---------------------------------------------------------------
                  -                        GRAPH WARNING                        -
                  -       STATEMENT  10  IS UNREACHABLE OR IS IN AN INFINITE LOOP -
                  ---------------------------------------------------------------
                                                                          STACK
  11              END
-------------------------------------------------------------------------------------
                  STATEMENT ANALYSIS SUMMARY          ERRORS  WARNINGS
                  ---------------------------         ------  --------
                  GRAPH CHECKING                         0        1
                  CALL CHECKING                          2        0
                  MODE CHECKING                          0        1

                                1ST  TOTAL  LAST   IN/OUT ACTUAL PHYSICAL
 NAME       SCOPE      MODE     STMT  USES  STMT     USE   USE    UNITS
-------------------------------------------------------------------------------------
 AREA       PARAMETER  INTEGER   1     6    10             BOTH
 DIAMTR     VALUES     REAL      2     2     4             INPUT
 RADIUS     LOCAL      REAL      4     3    10
 PI         LOCAL      REAL      3     1     5
                  ---------------------------------------------------------------
                  -                        SET/USE WARNING                      -
                  -  VARIABLE PI          MAY BE USED BEFORE BEING ASSIGNED A VALUE -
                  ---------------------------------------------------------------
-------------------------------------------------------------------------------------
                  SYMBOL ANALYSIS SUMMARY             ERRORS  WARNINGS
                  ---------------------------         ------  --------
                  SET/USE CHECKING                       0        1
```

The Statement Analysis Summary contains the warning and error messages interspersed appropriately in the code. Unknown externals--routines called which are not in the library--are listed on the right side of the printout. The numbers of errors and warnings are listed at the bottom.

The Symbol Analysis Summary shows the name, scope, and mode of each symbol in any executable statement in the module. The actual use of global variables is defined as INPUT, OUTPUT, or BOTH. For any variable that is used before being set to a value, or set and not used, a warning indicates the condition, which could lead to errors.

Figure 2.12. Static Analysis Report

REACHING SET. This option executes the "module retesting assistance" process of FAVS. The user identifies a section of code he desires to exercise by specifying the number of the DD-path to be "reached", and the number of a DD-path from which it is to be reached. The user may specify either iterative or non-iterative reaching sets to be generated. FAVS prints a list of DD-paths that connect the specified points -- the "reaching set" -- and the actual program statements on the paths. With this list, the user can identify which parts of the program need to be executed (and therefore which program values need to be modified) for the selected statement to be executed. Test cases can then be constructed, and the user may rerun Test Execution to ascertain the additional program coverage provided by the new set of test cases.

The basic command

OPTION = REACHING SET

must be followed by a command specifying a reaching set:

REACHING SET,MODULE=(<name>),TO= <DD-path number>,

FROM= <DD-path number>{,ITERATIVE}.

This command generates a non-iterative reaching set unless ITERATIVE (preceded by a comma) is included; in that case, the reaching set which includes all possible iterative paths is generated.

2.3 FAVS ANALYZER COMMANDS

The FAVS Analyzer produces coverage analysis reports, generated from the data collected during execution by the probes inserted when the INSTRUMENT option is selected. Figure 2.13 shows the execution coverage sequence beginning with FAVS instrumentation of a program (compare with Fig. 2.7), through the usual compilation and execution (shown inside dashes), to the input of execution coverage commmands which then generate coverage reports. The entire sequence can be performed in the same run.

During test execution the program operates normally, reading its own data and writing its own outputs. The instrumented modules also accumulate data on DD-path traversals. Each test execution may consist of a number of test cases.

The coverage reports are controlled by two coverage commands, an option selection and a module selection command. The type of report is specified by the command:

OPTION(S) = <list>.

where <list> may be one or more of the three options: SUMMARY, NOTHIT, or DETAILED.

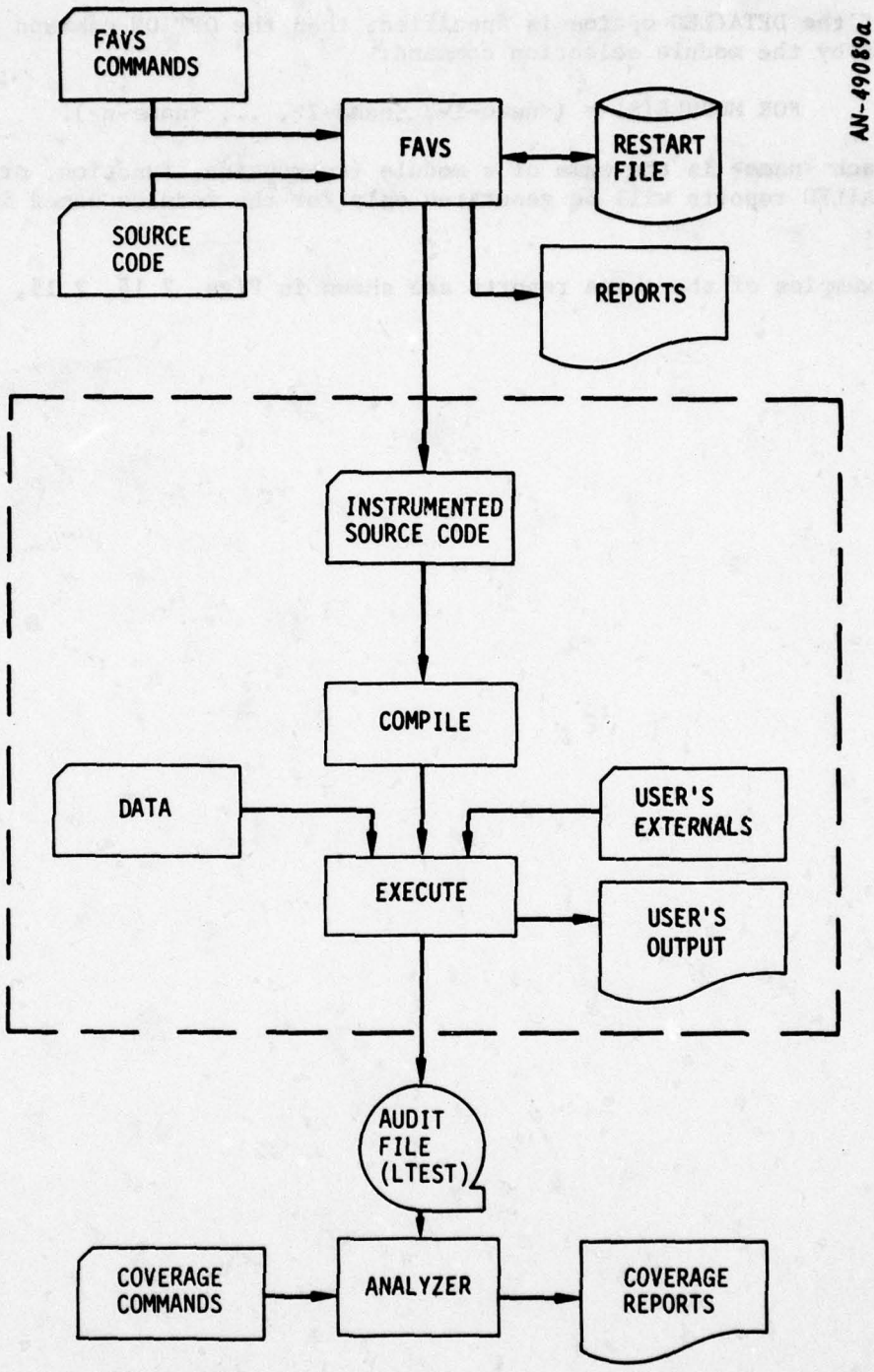


Figure 2.13. Execution Coverage Sequence

If the DETAILED option is specified, then the OPTION command must be preceded by the module selection command:

FOR MODULE(S) = (<name-1>, <name-2>, ... <name-n>).

where each <name> is the name of a module (subroutine, function, or program). The DETAILED reports will be generated only for the modules named in this command.

Examples of the three reports are shown in Figs. 2.14, 2.15, and 2.16.

			SUMMARY -- THIS TEST			CUMULATIVE SUMMARY			
TEST CASE	MODULE NAME	NUMBER OF DD PATHS	NUMBER OF INVOCATIONS	DD PATHS TRAVERSED	PER CENT COVERAGE	NUMBER OF TESTS	INVOCATIONS	TRAVERSED	COVERAGE
1	MAIN	3	1	2	66.67	1	1	2	66.67
	CLASS	98	0	0	0.00	1	0	0	0.00
	SSALLSS	101		2	1.98	1		2	1.98
2	MAIN	3	0	1	33.33	2	1	2	66.67
	CLASS	98	1	34	34.69	2	1	34	34.69
	SSALLSS	101		35	34.65	2		36	35.64
3	MAIN	3	0	1	33.33	3	1	2	66.67
	CLASS	98	1	2	2.04	3	2	35	35.71
	SSALLSS	101		3	2.97	3		37	36.63
			•			•			
			•			•			
			•			•			
9	MAIN	3	0	1	33.33	9	1	2	66.67
	CLASS	98	1	30	30.61	9	8	57	58.16
	SSALLSS	101		31	30.69	9		59	58.42
10	MAIN	3	0	1	33.33	10	1	2	66.67
	CLASS	98	1	27	27.55	10	9	57	58.16
	SSALLSS	101		28	27.72	10		59	58.42

Figure 2.14. SUMMARY Report: Multiple-Test DD-Path Execution, All Modules

MODULE NAME	TEST NUMBER	PATHS NOT HIT	LIST OF DECISION TO DECISION PATHS NOT EXECUTED
<MAIN >	10	2	1 2
	CUMUL	1	2
<CLASS >	10	71	3 5 6 7 10 12 13 14 15 16 17 18 19 20 21 22 24 26 28 30 32 34 35 36 37 38 39 40 41 42 43 45 48 50 51 52 53 54 58 59 62 65 66 67 68 70 71 73 74 76 77 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98
	CUMUL	41	5 6 7 10 17 28 32 34 36 37 38 40 41 42 43 45 48 50 51 52 53 59 62 67 68 70 71 77 79 80 81 82 83 84 85 86 92 95 96 97 98

Figure 2.15. NOTHIT Report: DD-Paths Not Executed

RECORD OF DECISION TO DECISION (DD PATH) EXECUTION

MODULE sCLASS s TEST CASE NO. 8

DD PATH NUMBER	NO.	NOT EXECUTED	NUMBER OF EXECUTIONS -- NORMALIZED TO MAXIMUM 1.-------20.-------40.-------60.-------80.------100.		NUMBER OF EXECUTIONS
1				1	1
2				2	1
3	3	00000			
4				4	1
5	5	00000			
...	...	00000	... ...		
7	7	00000			
8			XXXXXXXXXXXXXXXXXXXXXXXXXXXX	8	48
9			XXXXXXXXXX	9	18
10	10	00000			
11			XXXX	11	8
12			XXXXXX	12	10
13			X	13	2
14			XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	14	81
15			XXXX	15	8
16			XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	16	73
17	17	00000			
18				18	1
19			XXXX	19	7
20	20	00000			
			•	•	
			•	•	
			•	•	
69				69	1
70	70	00000			
71	71	00000			
72				72	1
73				73	1
74	74	00000			
...	...	00000	... ...		
98	98	00000			

TOTAL NUMBER OF DD PATH EXECUTIONS = 422

TOTAL OF 61 NOT EXECUTED EXECUTED 37/ 98 PERCENT EXECUTED = 37.76

Figure 2.16. DETAILED Report: Single-Test DD-Path Execution, One Module

3 USE OF FAVS IN SOFTWARE TESTING

The concepts implemented in FAVS address the basic problem of assembling systems of hardware and software to achieve desired behavior. By the very nature of a specification for a software system and the system built from it, the system's behavior includes both specified and unspecified behavior. The specified behavior may be acceptable (i.e., what was desired) or unacceptable (i.e., not desired but an unforeseen consequence of the system's behaving as specified). The unspecified behavior may be acceptable (i.e., fortuitously providing a capability not included in the specification) or unacceptable.

Acceptable behavior in a software system is approached in two ways: attempting to build software of inherently high quality, and attempting to identify failure by testing the software at various stages. FAVS augments the testing process.

3.1 RELATIONSHIP BETWEEN SOFTWARE TESTING AND SOFTWARE VALIDATION

Figure 3.1 shows the relationship between a software system functional specification, the software, and the process by which testing seeks to invert, or "validate," the software implementation phase. Ideally an automated verification system (AVS) would support software validation: the

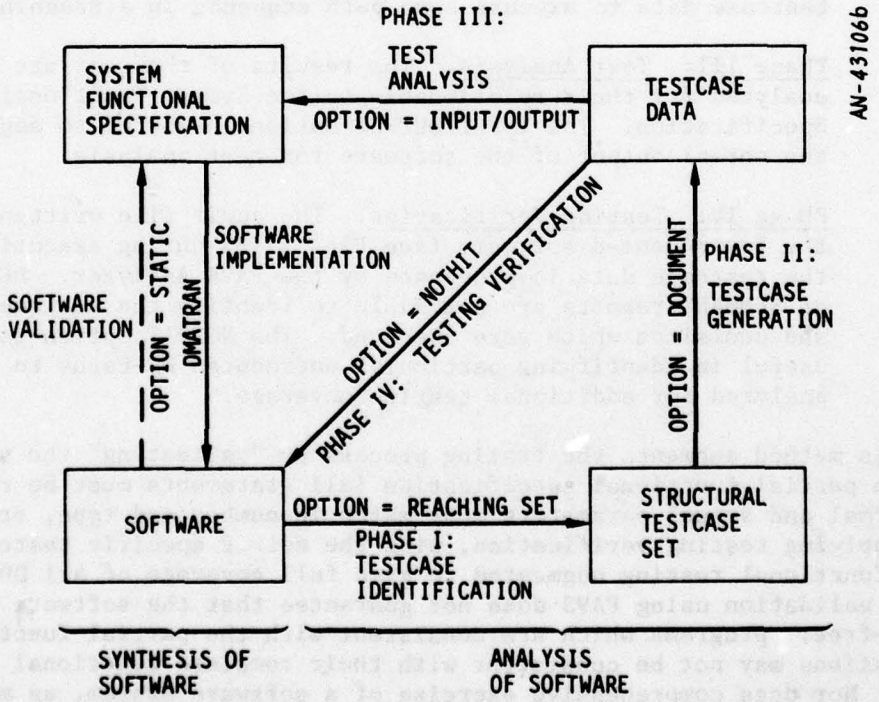


Figure 3.1. Software Testing and Validation Using FAVS

demonstration of consistency between the software and its specification, independent of any specific testcases. The discipline of program proving (which is not currently applicable to even moderate-sized software systems) aims most directly at this ideal. Practically, an AVS should support testing verification: the demonstration of correct behavior for a representative and thorough set of specific testcases.

Figure 3.1 also shows how FAVS fits the testing verification methodology. DMATRAN provides constructs for structured programming in FORTRAN, supports top-down programming in FORTRAN, and provides data access assertions. FAVS assists in partial software validation through the STATIC option and provides dynamic structural testing, untested-path identification, output augmentation, and documentation to assist in testing verification.

The four phases described in the figure are implemented with FAVS as follows:

- Phase I: Testcase Identification. The software is analyzed for path sequences to execute specified DD-Paths, yielding a collection of structural testcases. The REACHING SET option identifies code for structural testcases.

- Phase II: Testcase Generation. Supplying specific input values for a structural testcase converts it to an actual test. The DOCUMENT option provides reports useful in generating specific testcase data to execute some path sequence in a reaching set.

- Phase III: Test Analysis. The results of the test are then analyzed for their relationship to the System Functional Specification. The INPUT/OUTPUT option is useful to augment the normal output of the software for test analysis.

- Phase IV: Testing Verification. The audit file written by the instrumented software (see Fig. 2.13) during execution of the testcase data is processed by the FAVS Analyzer. DETAILED or SUMMARY reports are available to identify the statements and decisions which were executed. The NOTHIT option is most useful in identifying particular unexecuted DD-Paths to be analyzed for additional testing coverage.

This method augments the testing process by "validating" the software against a partial functional specification (all statements must be reachable, formal and actual parameters must match in number and type, etc.); and by applying testing verification, with the set of specific testcases used in functional testing augmented to give full coverage of all DD-paths. Software validation using FAVS does not guarantee that the software analyzed is error-free: programs which are consistent with the partial functional specifications may not be consistent with their complete functional specifications. Nor does comprehensive exercise of a software system, as measured by FAVS, guarantee that it is error-free. Nothing short of the impossible goal of executing all instances of all testcase sets would provide this

guarantee through testing. However, practical experience indicates that software validation and testing verification supported by FAVS will locate a very high proportion of errors. Hence, the use of partial software validation and testing verification as an approximation to full software validation seems to be reasonable and practical.

3.2 WRITING TESTABLE SOFTWARE

The problem of constructing software which performs its intended function can be approached from both "synthetic" and "analytic" viewpoints. This section concentrates on augmenting a broadly applicable "synthetic" approach--writing testable software which is error-free the first time it is tested. Although this is an ideal which cannot always be achieved, common sense and practical experience indicate that a carefully written system will work reliably sooner and with less testing effort than a hastily implemented system.

In this section we discuss some widely accepted guidelines for software implementation that reduce the cost of testing and improve the quality of the software. Reference 1 gives additional guidelines for the preparation of readable software.

We briefly list the guidelines and then go on to discuss how they are implemented and checked with DMATRAN and FAVS.

3.2.1 Structural Guidelines

Use Small Blocks of Code. The first guideline is the most important of all. It is to keep each block of code as small as possible; at most 50 lines of code or one printed page. When a programmer believes that more than one page of code is necessary to describe the functions in a block, the block should be split in two.

Small blocks can be tested more thoroughly for a large set of values than large blocks. While in a large system it is impossible to exercise all possible combinations of paths, it is not unrealistic to assume that all possible combinations can be exercised in each block individually.

Use Single-Entry, Single-Exit Control Structures. Much has been written on the well-structured program.[2-4] It has been shown that three control structures are all that is necessary to write any computer program,[5] that less time is required to write a program using structured-programming techniques,[6] and that the use of such techniques eliminates the need for flow charts.

The most important feature of a well-structured program is the restriction to single-entry, single-exit control structures. While this restriction is often recommended to improve readability and to eliminate decision errors, it also eases testing.

Use Few Types of Control Structures. Modern programming languages provide a wealth of control structures. Moderation in the types of control structures used simplifies both the testing and the understanding of a program. A good guideline to the types to use is an article by Mills,[7] who stated that the only truly useful one beyond a DO loop and an IF decision is a form of the CASE statement.

Use Few Paths. The guideline of keeping a block small helps keep the number of paths small. Even so, it helps to pay explicit attention to the number of paths. The amount of testing required for a program is an exponential function of the number of decisions; hence each unnecessary decision point adds greatly to the testing effort.

Very often the designer of a program can decrease the number of paths dramatically by altering an algorithm or choosing another one. An example is in the use of trigonometric functions. All trigonometric functions can be computed by algorithms that provide results only for the range 0 to 45°. But then the program that uses the algorithms is forced to decide which functions should be used, after mapping the angle into the range 0 to 45°. If the trigonometric functions are altered to respond to the full range of values that the machine can calculate, the number of paths at the tester level is decreased.

Do Not Use Implied Loops. An implied loop is one which uses an IF test and a GOTO to form a loop in the program. It is often used by programmers to obtain loop features normally unavailable, such as:

1. Alteration of the control variable by uneven increments
2. Use of floating-point variables as control variables
3. Overlapped nesting of loops
4. Transfer of control into a loop
5. A loop structure not provided in the programming language

Implied loops are particularly susceptible to errors, especially infinite loops. It is often not apparent to the tester that such a loop exists, and therefore it will not be tested in the way that a loop would be tested.

Do Not Use Multiway Transfers. Some programming languages contain multiway transfers. The most notorious is the FORTRAN arithmetic IF statement. It is a single-entry, triple-exit statement that should not be used because it is hard to follow, and hard to test because it allows an arbitrary transfer of control. In most cases a logical IF statement can replace an arithmetic IF, reducing the number of paths to two and resulting in a well-structured program. Where multiple paths are needed, a CASE construct or one of its equivalent IF...THEN...ELSE...ENDIF constructs should be used.

Exclusive use of the DMATRAN control constructs assists in conforming to all the structural guidelines for testable software. The first guideline, "use small blocks of code," may be difficult to follow in a large FORTRAN

implementation. Software systems consisting of hundreds of separately compilable modules often exceed an operating system's constraints, and may cause maintenance problems. The DMATRAN BLOCK construct provides a solution to this problem by allowing large routines to be internally modularized into small, comprehensible blocks of code. DMATRAN provides five single-entry, single-exit control structures, thus satisfying the guidelines "use single-entry, single-exit control structures" and "use few types of control structures". The next guideline, "use few paths," can be largely enforced by limiting indentation within any one block of code to a level of 6. Implied loops and unstructured multiway transfers (which are prohibited in the last two guidelines), cannot be written using the DMATRAN control constructs.

3.2.2 Symbolic Guidelines

The symbolic guidelines are intended to make the association between names, objects, types, and physical quantities as clear as possible to a tester.

Use Meaningful Names. Most programming languages today allow the use of meaningful names such as TIME, SPEED, HEIGHT. A tester has a better feel for the function of a program if such names are used instead of names like X1, X2, X3.

Use One-to-One Matching Between Names and Objects. One name should refer to only one object, and one object should have only one name. Often, unfortunately, one object has different names. For example, a FORTRAN equivalence statement can allow the same area in memory to be referred to as A or as B . The excuse may be made that equivalence in FORTRAN is necessary to allow the definition of table structures. However, there is no reason to use it otherwise. Nor should multiple names be used in languages which allow tables to be defined.

Identify Constants. When it is known that a name represents a constant, the tester or test tool can make various simplifications. For example, it is known that a predicate stated in terms of constants need not be traced back to determine its value on input.

There are three types of constants that can be identified:

1. Preset by the compiler
2. Initialized by an assignment statement
3. Read as data from an external device

While provision has been made in some languages to identify the first, the others need identification as well so that a test can be made that they are set only once and are indeed constants.

Use One Physical Type for One Name. To save memory or to save names, programs are often written where a variable contains different physical quantities at different times in the program. For example, at one instant the

variable represents height in feet and at another height in miles. This multiple association between physical types and names should not be done, because it confuses the tester, who has to keep track of the physical units from one instant to the next.

Use One Data Type for One Name. In some languages a variable name can be used to store either integer data or character data. This allows handling characters in limited languages, but makes it difficult to test for legal values in a variable. When a variable is to be used to contain characters, its use should be restricted to characters only.

Use Local Variables in Preference to Global Variables. This guideline is not intended to result in extra unnecessary variables, but to cut down on the use of global variables as local variables. For example, temporary variables should always be local, never global variables. The number of global variables should be kept at a minimum, since they are more difficult to keep track of.

Separate Inputs from Outputs. Different variable names should be used for the inputs and the outputs of a module. A module should not be called with the same actual parameter in more than one parameter position. If a variable is used both as an input and as an output, the invoking module may require that the variable not be changed, which is a difficult problem to trace. This guideline may increase the number of variables.

The guideline "*use meaningful names*" is reinforced by the ability to give DMATRAN BLOCKS long, mnemonic names. This feature adds to the readability of properly designed and implemented DMATRAN programs. The use of INPUT and OUTPUT statements processed by DMATRAN and FAVS makes it apparent when the guideline "separate inputs from outputs" has been violated.

3.2.3 Loop Guidelines

Limit Types of Loops. Most languages provide too many means of generating a loop structure. The best loop structure for testing and verification analysis is the DO WHILE loop, with one integer control variable that changes monotonically with equal increments. DMATRAN provides only two looping constructs.

Keep Invariant Conditions. In loops, indicate the invariant conditions on the variables within the loop. Where there is a choice between making a condition sometimes true and making it always true, change the algorithm to make it always true. Such a condition is termed an invariant. The SQRT function in Fig. 2.2 computes the square root by successive approximations. An invariant condition for the loop in SQRT is that the next approximation is at least as good as the previous approximation. The condition

```
ABS(A - X ** 2) .GE. ABS(A - ((X + A/X)/2) ** 2)
```

inserted as a comment immediately after the DO WHILE statement would precisely describe this loop invariant.

Maintain Monotonic Control Variables. If the control variable changes monotonically on every path through the loop, one can then hope to prove that the loop will terminate. Otherwise an infinite-loop condition can arise.

3.2.4 Interface Guidelines

Use Top-Down Design Techniques. There are arguments for both top-down and bottom-up testing. Testing from the bottom up on a per-module basis is worthwhile. However, it is only with top-down testing that the whole system is tied together from the beginning. Bottom-up testing requires the design of many drivers, interface errors remain hidden, and the operating system interacts only late in the testing.

Use a Support Subroutine Library. The use of a subroutine library allows testing on a module basis, as well as checking the correctness of each module's interfaces, and encourages modularity. Well designed support subroutine interfaces can considerably decrease testing expense and effort.

State Data Access Rights. Data access rights for each global variable and for each parameter can be listed with the DMATRAN INPUT and OUTPUT assertion. The use of each such variable should be stated for each module.

Another use of the BLOCK construct is to support modular top-down programming. Several major problems exist in using FORTRAN for top-down programming. The first is that as additional detail is being added to the current implementation, interfaces between related routines must be frequently updated. This requires updating all instances of invocations to modified routines and updating all instances of modified common blocks. Since the BLOCK construct allows modularity within one compilable unit in which all program variables are global, no such interface maintenance problems are encountered. Also the development history of a FORTRAN top-down design is largely lost, while the long BLOCK name capability of DMATRAN allows some of the history to be embedded into the implementation.

The FAVS STATIC option encourages the use of a support subroutine library by automatically detecting many errors in the invocation of support routines. A restart file which describes the formal parameters of all support subroutines can be constructed with the FAVS LIST option. Only the properties of the formal parameters need be included in the source input to FAVS for building the restart file. For instance, the FORTRAN support library routine IABS (absolute value) can be described as

```
INTEGER FUNCTION IABS (I)

INTEGER I

INPUT (I)

RETURN

END
```

This defines IABS as an integer function with one formal parameter, which is an integer variable that is used as input (see Sec. 3.2.2). Both system support library routines and user support library routines can be described on a FAVS restart file for use in analyzing FORTRAN and DMATRAN routines.

3.2.5 Comprehensive Synthesis Approach Using DMATRAN and FAVS

A method for using the DMATRAN precompiler and the FAVS STATIC option to implement high-quality DMATRAN source programs is shown in Fig. 3.2. After the initial version of the software is written (according to guidelines similar to those just discussed), an iterative process is begun which aims at removing errors before the software is ever executed. The first step in this process is performed by the DMATRAN precompiler. It produces a listing of the DMATRAN source program which is automatically indented to show its structure, and has any invalid structure constructs flagged. The intermediate FORTRAN source program produced by the precompiler is next checked by the FORTRAN compiler for syntax errors. FORTRAN statements which are in error can be traced directly to the corresponding DMATRAN source statements. Finally the STATIC option of FAVS is used to analyze the DMATRAN source program, including all calls to support library subroutines. After a module has passed all these error checks, single-module testing (Sec. 3.3) can proceed with the expectation of minimum testing expense.

3.3 SYSTEMATIC SINGLE-MODULE TESTING VERIFICATION

The testing verification process for single-module coverage has a single objective: to construct test cases which cause the execution of DD-paths not yet executed. Testing is complete when all DD-paths have been exercised, or when those which have not been exercised are shown by the program tester to be logically unexecutable.

This process is diagrammed in Fig. 3.3. The software must first be prepared for testing (analyzed for its structural properties and instrumented for testing). This is done by processing the software through the INSTRUMENT option of FAVS. The resulting data base contains all information necessary for subsequent testing activities. Selecting the DOCUMENT option at this time provides the basic reference material to use in testcase data generation. The testing process begins by executing whatever testcases for the module already exist; this initial test, performed with the assistance of the INSTRUMENT facility of FAVS, results in a coverage report which identifies the DD-paths which have <u>not</u> been exercised (the NOTHIT option). If there are none, then testing is finished.

The next step is to choose a likely DD-path upon which to concentrate the testing. After this choice is made (see below), the tools already described are employed, as appropriate, to assist in generating testcases which will increase the percentage of DD-paths that have been exercised. These additional testcases are added to the previously generated ones, additional test executions are made, and the ANALYZER facilities are used to provide the updated coverage report.

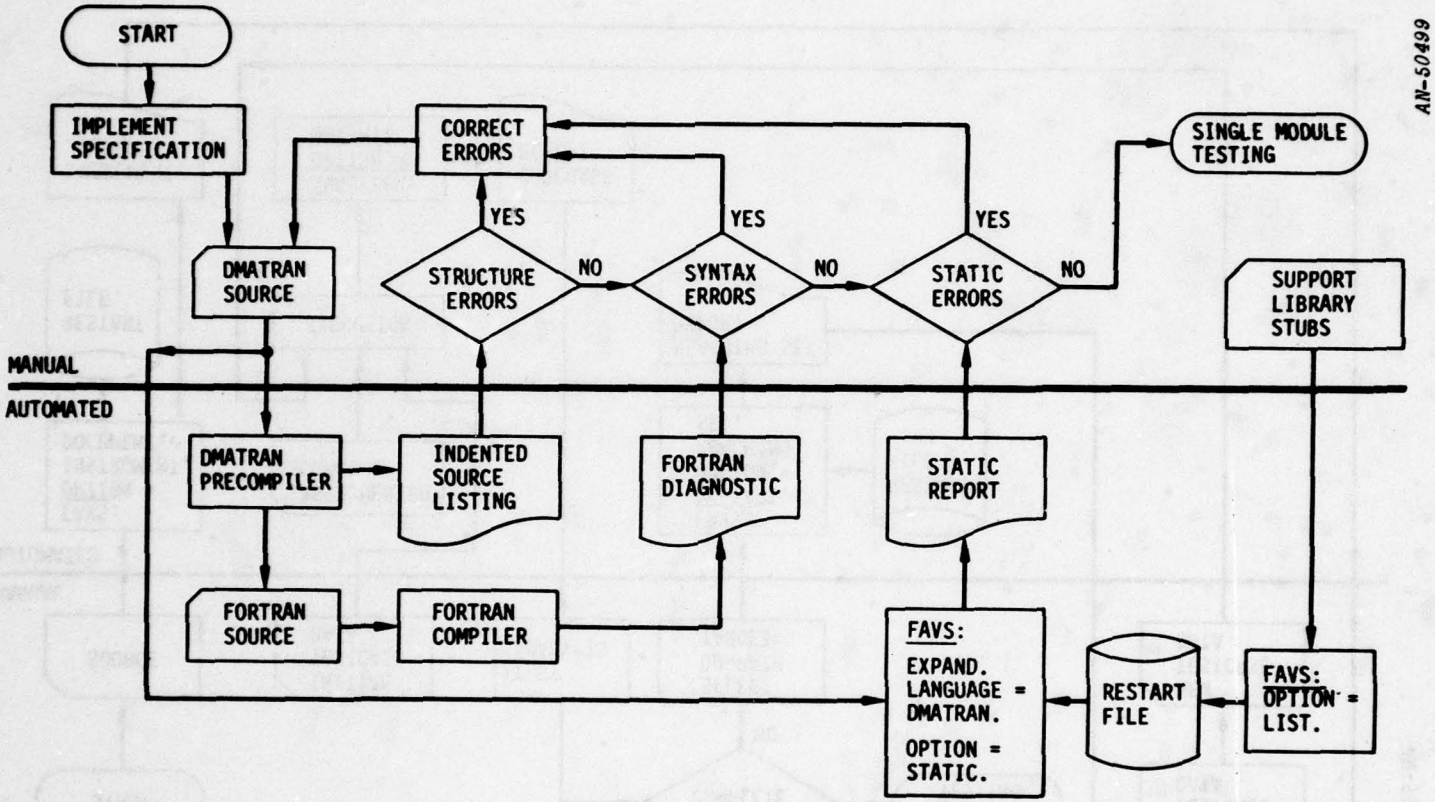


Figure 3.2. Writing Testable Software Using FAVS

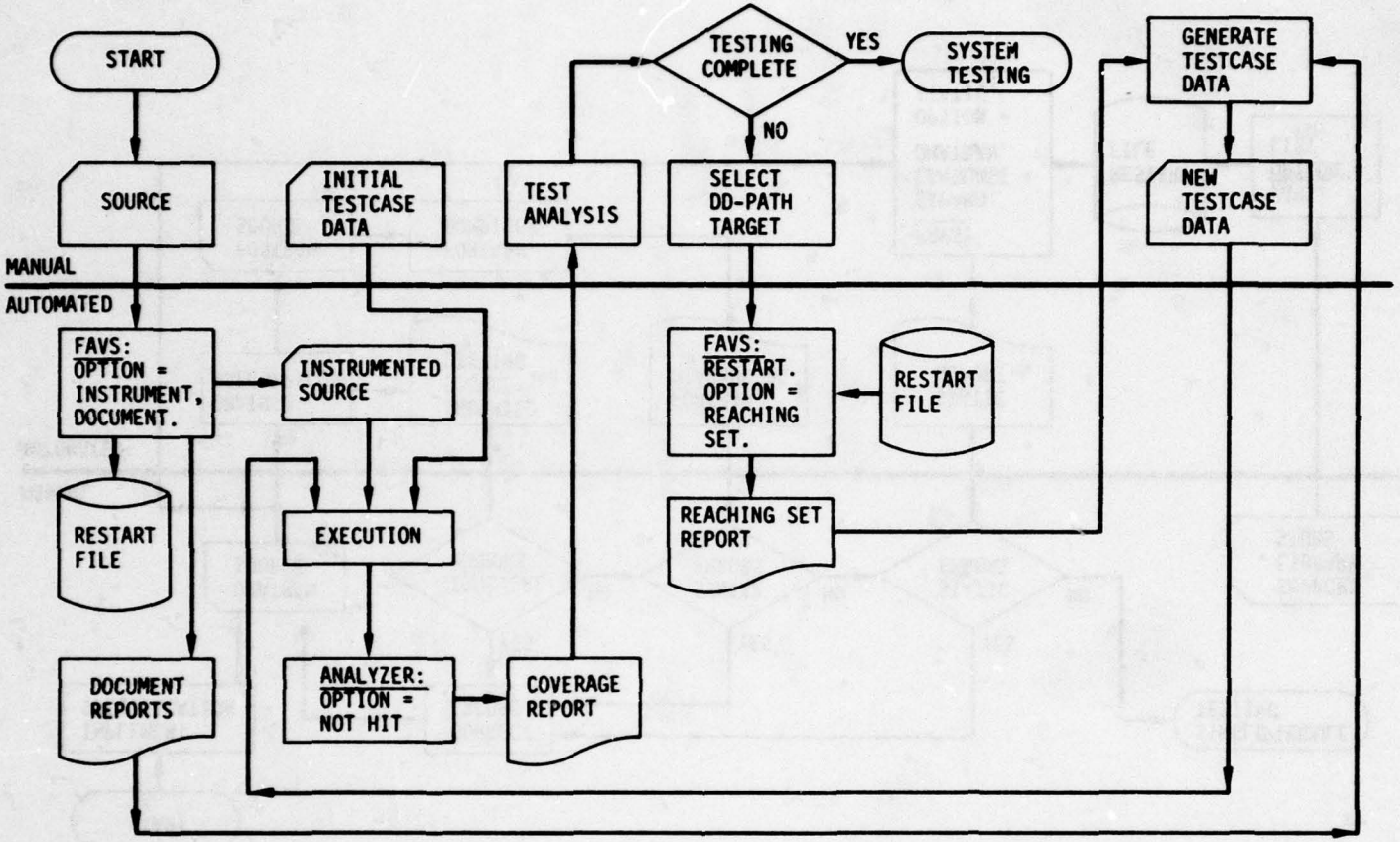


Figure 3.3. Single-Module Testing Verification Using FAVS

The effectiveness of this scheme for program testing verification depends to some extent on the mechanism used to select the next target DD-path for testcase generation. This testing target should be one of the unexecuted DD-paths; when there is more than one untested DD-path to choose from, the choice among them can affect the amount of "collateral testing", and thereby influence the efficiency of the testcase set.

The DD-path selection criteria used should attempt to maximize collateral testing. On the other hand, maximum collateral testing coverage may make testcase data generation very difficult. The selection function actually used should depend on the nature of the program being analyzed. The following guidelines may be of value:

1. Choose a DD-path which resides on the highest possible decision level.

    a. This assures a high degree of collateral testing, since after the target DD-path is executed the program must still finish executing and, in the process, may hit a large number of other untested DD-paths.

    b. If such a DD-path has not yet been executed, and all DD-paths are to be exercised, then it will have to be dealt with at some time anyway. Better sooner than later.

2. Choose a DD-path which is at the end of a fairly long reaching sequence. The reasons for this are similar to (1.a), but involve an additional observation: the more complex a set of logical conditions dealt with in generating a testcase, the more likely that the resulting testcase dataset will resemble data which corresponds to the functional nature of the program being analyzed.

3. If a prior testcase carries the program execution near one of the untested DD-paths, it may be more economical to determine how that testcase can be modified to exercise the untested DD-path.

4. If the analyses required for a particular DD-path selection are difficult, then attempt to choose a DD-path which lies along the lower-level portions of its reaching sequence(s). Doing this simplifies the analysis problem, but may still achieve a high degree of collateral testing.

These analyses are supported by the FAVS DOCUMENT option (see Sec. 2).

3.4 SYSTEMATIC SOFTWARE SYSTEM TESTING VERIFICATION

The system testing verification effort can be organized according to two fundamentally distinct strategies: (1) bottom-up system testing, and (2) top-down system testing.

Bottom-Up Testing: This testing strategy attempts to provide comprehensive system testing coverage by building test cases from the bottom of the

system invocation hierarchy first, and extending these test cases upward during the continuing and concluding testing phases. Bottom-up testing may require the use of special testing environments (see below), but is likely to achieve the best overall testing coverage.

Top-Down Testing: This testing strategy deals with an entire software system first, and, after subsystem (or component) testedness is measured, proceeds downward through the software system's invocation structure. Test case data is added only at the topmost level and, as a result, a set of systemwide test cases is developed directly.

The optimum system testing strategy for a particular system generally combines the two strategies. The choice is based on the level of coverage achieved, the difficulty of proceeding upward or downward in the system organization, and the effort required to establish a testing environment in each case.

The basic ingredients of systematic software system testing are the following:

- The ability to perform comprehensive single-module testing for each invokable module
- Knowledge of the system's invocation structure
- Previous (and initial) system testing coverage measures
- A next-testing-target selection function to allocate testing effort.

The general form of system testing is shown in Fig. 3.4, which emphasizes the continuous use of a system testing coverage measure. The interaction between the system testing coverage measure and the process of selective application of the single-module testing procedure is described next.

Systemwide testing coverage can be measured in terms of the coverage for each module, or in terms of the coverage for an identifiable subset of related modules (i.e., a component). The coverage measure can be used to select the best next testing target. The simple per-module coverage measure will direct testing effort toward the module which is the least tested. The per-component coverage measure directs testing effort toward the component which is the least tested.

The measure actually used should depend on the internal structure, and possibly the functional requirements, of the software system as a whole. The measure should unambiguously identify the module(s) least tested, but should tend to identify a number of possible testing targets. The choice between them should be made within the confines of the invocation hierarchy, and by considering the two important variations of testing strategy: top-down testing, and bottom-up testing.

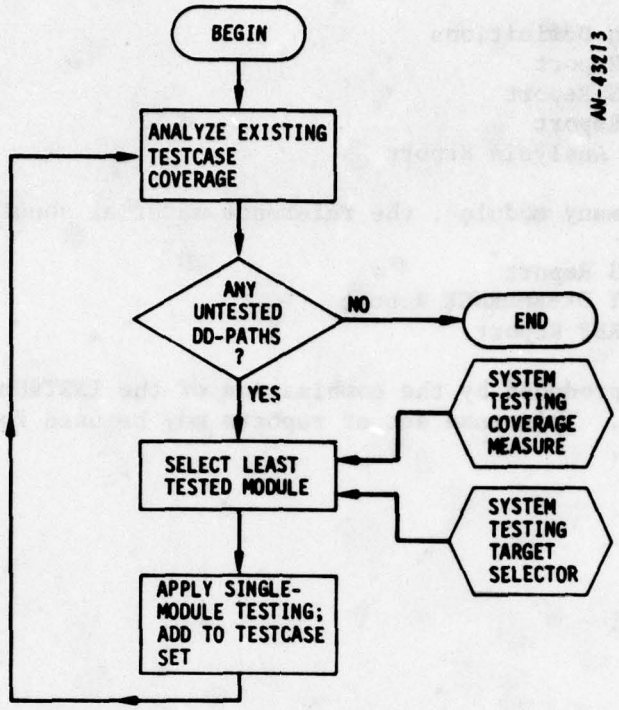


Figure 3.4. System Testing Procedure

3.4.1 General Strategy

The best approach for systematically testing a large software system will depend on the specifics of that system's elements; it is not possible to state a universally applicable strategy. Mixtures of the top-down and bottom-up approaches may well cost the least, and may result in the greatest testing coverage.

FAVS has facilities which directly assist in the testing of large software systems. The DOCUMENT option includes analyses which assist the program tester in grouping modules into subsystems and in constructing suitable testcases (see Sec. 2).

3.5 SUMMARY

The many options offered by the FAVS commands permit the user to tailor FAVS processing for a particular testing activity. As an additional benefit, some of the FAVS reports can be very useful in software documentation and code optimization. As with many other software packages, how FAVS is utilized varies greatly among users. Quite often several reports are used together to provide more insight into the specific problem at hand.

For program testing purposes, a basic set of reference material for each module is the following:

DD-Path Definitions
READS Report
INVOKES Report
BANDS Report
Static Analysis Report

For a system with many modules, the reference material should also include:

COMMONS Report
LIBRARY DEPENDENCE Report
CROSS REF Report

These reports are produced by the combination of the INSTRUMENT, DOCUMENT, and STATIC options. This same set of reports may be used for software documentation purposes.

REFERENCES

1. B. W. Kernighan and P. J. Pluger, The Elements of Programming Style, McGraw-Hill, 1974.

2. O. J. Dahl, E. W. Dijkstra, and C. Hoare, Structured Programming, Academic Press, 1972.

3. E. W. Dijkstra, "GO TO Statements Considered Harmful," Communications of the ACM, Vol. 11, March 1968, pp. 147-148.

4. R. E. Noonan, "Structured Programming and Formal Specifications," IEEE Transactions on Software Engineering, Vol. 1, No. 4, December 1975, pp. 421-425.

5. C. Bohm and C. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," Communications of the ACM, Vol. 9, May 1966, pp. 366-371.

6. F. T. Baker, "Structured Programming in a Production Programming Environment," Proceedings of the International Conference on Reliable Software, Los Angeles, April 1975.

7. H. B. Mills, "The New Math of Computer Programming," Communications of the ACM, Vol. 18, January 1975, pp. 43-48.

8. E. F. Miller, Jr., Methodology for Comprehensive Software Testing, General Research Corporation CR-1-465, June 1975.

[redacted]