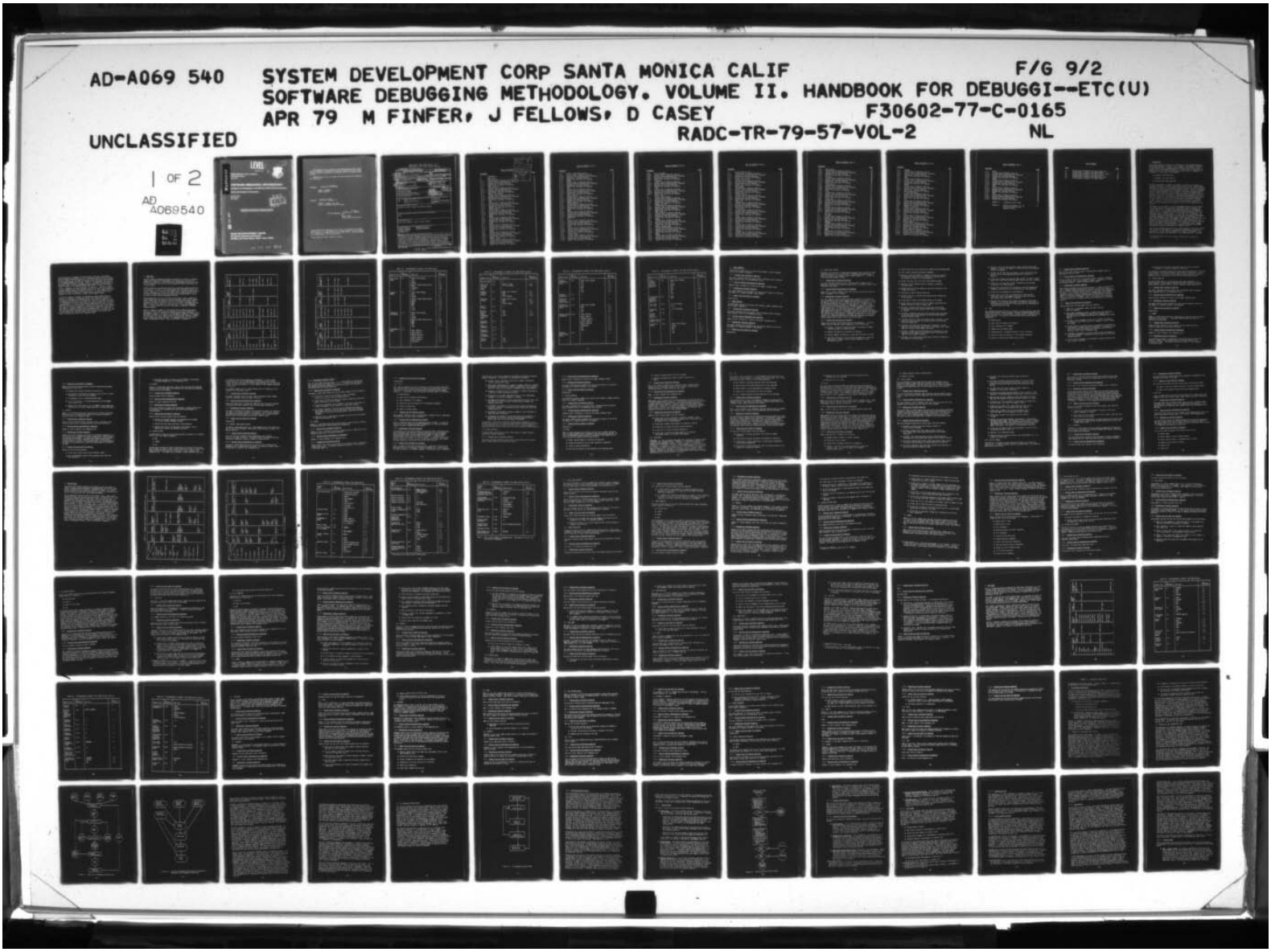
AD-A069 540 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME II. HANDBOOK FOR DEBUGGI--ETC(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-2 NL

1 OF 2
AD A069540



AD A069540

LEVEL III

AD A069541 SC


RADC-TR-79-57, Vol II (of three)
Final Technical Report
April 1979

# SOFTWARE DEBUGGING METHODOLOGY

## Handbook for Debugging in the MULTICS/GCOS/RTM Environments

System Development Corporation

Marcia Finfer
Jon Fellows
Dan Casey

DDC
RECEIVED
JUN 7 1979
C

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DDC FILE COPY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 06 06 014

SOFTWARE DEBUGGING METHODOLOGY, VOL II

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-57, Vol II (of three) has been reviewed and is approved for publication.

APPROVED: Frank S. LaMonica

FRANK S. LAMONICA
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, COL, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

(19) TR-79-57-VOL-2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: RADC-TR-79-57, Vol II (of three)

2. GOVT ACCESSION NO.

(9) 3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): SOFTWARE DEBUGGING METHODOLOGY, Volume II, Handbook for Debugging in the MULTICS/GCOS/RTM Environments

5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, Sep 77 - Oct 78

6. PERFORMING ORG. REPORT NUMBER: N/A

(10) 7. AUTHOR(s): Marcia Finfer, Jon Fellows, Dan Casey

(15) 8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0165

9. PERFORMING ORGANIZATION NAME AND ADDRESS: System Development Corporation, 2500 Colorado Avenue, Santa Monica CA 90406

(16) (17) 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810295, 02

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

12. REPORT DATE: April 1979

13. NUMBER OF PAGES: 124

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same

(12) 122 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

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 Debugging
Software Testing
Debugging Tools
Debugging Techniques
Debugging Methodology

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

A debugging study was conducted which surveyed current research being performed in the area of software debugging during integration level testing. Particular emphasis was placed on assessing debugging tools and techniques which were applicable to embedded software developments. The purpose of the debugging study was to define a software debugging methodology applicable to diverse environments to be utilized during integration testing of system software. The results of the study are contained in three volumes. This volume presents the application of the debugging methodology to three specific environments.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339 900

slk

NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

## LIST OF TABLES

## 1. INTRODUCTION

This volume presents a scenario for utilization of the software debugging methodology developed in Volume I of this report. The scenario is a table-driven procedure that is geared to the information requirements of each process within the debugging methodology. In particular, it identifies the vendor-supplied tools that are available to satisfy those information requirements in three specific environments:

- Honeywell 6180 under GCOS
- Honeywell 6180 under Multics
- SEL 85/86 and 32/55 under RTM.

The SEL minicomputer programming environment is considered in addition to the RADC environment in order to analyze the effect of a less sophisticated debugging environment on the debugging process model and to allow a comparison with the more complex programming environments at RADC. The SEL environment was used in the development of the Telemetry Integrated Processing Subsystem (TIPS), a complex network of mini-computers in a real-time communication system.* (This project presented an unique opportunity to survey a new site's debugging capabilities at a point in time when coding/debugging activities were just commencing. The debugging capabilities were limited and consisted of mostly vendor supplied tools.) The coverage of the RADC environments includes certain non-vendor supplied tools that are generally available at RADC. Because of the evolving nature of the work being performed at RADC, application-dependent environmental factors are not addressed in this volume.

The intent of this volume is to provide debugging analysts with an insight into how particular tools available in the RADC or SEL environments can satisfy the generic information requirements developed in the debugging process model. (It is expected that the information contained herein will require on-site refinement due to the limited availability and use of the debugging tools by the study team.) This volume is also intended to provide RADC software engineers with an insight into the types of tools needed to support debugging. By comparing the information requirements of the model and the descriptions of the generic tools contained in Volume I with the descriptions of the current tool capabilities at RADC contained in this volume, a software engineer should be able to define areas in which development of new capabilities is desirable. The information in this volume should also assist a software engineer in evaluating proposed development environments offered by contractors in response to RFPs.

Each of the subsequent sections of this volume contains a table that relates the information requirements of the debugging methodology to the tools available to satisfy those requirements in a specific environment. Each section also contains the description of the identified tools. Each des-

* Further discussion of this environment can be found in Section 2.3 of Volume III.

cription contains a statement of the functions of the tool, the program state information captured by the tool, the abstract machine level at which the tool displays this information, the individual steps within the processes of the debugging model that the tool supports, and the capabilities and usability features offered by the tool. While it is assumed that the reader is thoroughly familiar with the debugging process model and the terminology defined in Volume 1, the debugging methodology defined in Section 3 of that volume is repeated, for ready reference, in Appendix A. Appendix B summarizes non-debugging tools and techniques which can be used in the software development phases preceding integration testing (i.e., requirements analysis, design, and code and checkout) to enhance overall system reliability and traceability and, in that manner, support the debugging process.

To use this volume as a scenario for debugging a given problem within a specific environment, the debugging analyst first determines the generic information requirements by referencing the description (in Appendix A) of the debugging process step he is performing. Then he translates those generic requirements to specific ones based on the given problem and his current understanding of it. Next he uses the environment-specific table to determine which tools supply the desired information and the level of abstraction they support. There are numerous tools available to generate and/or capture each type of information required by the debugging model. By referring to the tool descriptions contained within each section of this volume, the reader can evaluate the individual capabilities and merits of the tools and select the ones best suited to his immediate needs.

## 2. GCOS TOOLS

The GCOS (General Comprehensive Operating Supervisor) system is a Honeywell system which supports multiprogramming, multiprocessing, and local and remote batch processing. Batch processing, remote-access processing, transaction processing, and time sharing are merged and operate concurrently on a single system.

A GCOS simulator is available on the Multics operating system (see Section 3) but there are significant differences between the simulated and actual versions of GCOS. In this section, the debugging tools are described according to their implementation on the actual GCOS system. Readers who are using the GCOS simulator should check the appropriate reference manuals for descriptions of GCOS simulator debugging features.

The GCOS debugging tools which include various dumps, traces, comparisons, symbol tables, maps, and selected location outputs are described in respect to the information requirements and steps of the debugging process model presented in Volume I, Sections 2 and 3, of this study. No attempt has been made to instruct the reader as to the specifics of using these debugging tools. To obtain these instructions, the reader is referred to the GCOS reference manuals.

Table 2-1 presents a summary of the program state information provided by GCOS tools and the level of abstract machine at which the information is represented. Table 2-2 presents the correspondence between the generic tools described in Appendix A of Volume I and the specific GCOS tools. In addition to these tools, the study identified tools and techniques available to RADC which, although not usually thought of as debugging tools, can be used to support the debugging process. They are described in Appendix B.

Table 2-1. Program State Information Provided by GCOS Tools

| TOOL \ INFORMATION | TRAVERSED PATHS | SYMPTOM LOCALITY | NAME-SPACE VALUES | EXTERNAL SYSTEM STATUS | SIMULTANEOUS EVENTS | SPACE ALLOCATION AND TIMING |
|---|---|---|---|---|---|---|
| 1. ALGOL Compiler | compiler-level | compiler-level | compiler-level | | | |
| 2. COBOL Compiler | compiler-level | compiler-level | compiler-level | | | |
| 3. Debug Trace | basic computer | basic computer | basic computer | basic computer | basic computer | basic computer |
| 4. FAVS | compiler-level | compiler-level | compiler-level | | | compiler-level |
| 5. FORTRAN Compiler | compiler-level | compiler-level | compiler-level | | | |
| 6. GEBORT | | basic computer | basic computer | basic computer | | basic computer |
| 7. GECHEK | | basic computer | basic computer | basic computer | basic computer | basic computer |
| 8. GELAPS | | | | | | basic computer |
| 9. General Loader | | compiler/assembler | compiler/assembler | | | compiler/assembler |
| 10. General Loader Debug | basic computer | basic computer | basic computer | | | basic computer |
| 11. GESNAP | | basic computer | basic computer | basic computer | | basic computer |
| 12. JAVS | compiler-level | compiler-level | compiler-level | | | compiler-level |
| 13. JOVIAL Compilation | compiler-level | compiler-level | compiler-level | | | |

Table 2-1. Program State Information Provided by GCOS Tools (Cont'd)

| TOOL \ INFORMATION | TRAVERSED PATHS | SYMPTOM LOCALITY | NAME-SPACE VALUES | EXTERNAL SYSTEM STATUS | SIMULTANEOUS EVENTS | SPACE ALLOCATION AND TIMING |
|---|---|---|---|---|---|---|
| 14. MACRO Assembler | assembler-level | assembler-level | assembler-level | | | |
| 15. RBUG | basic computer | basic computer | basic computer | basic computer | basic computer | basic computer |
| 16. STRUCTRAN-1 | | | | | | |
| 17. TDS | basic computer | basic computer | basic computer | basic computer | basic computer | basic computer |
| 18. TPOS | basic computer | basic computer | basic computer | basic computer | basic computer | basic computer |
| 19. Trace | basic computer | basic computer | basic computer | | basic computer | |
| 20. Utility Package | | | basic computer | basic computer | | |
| 21. $ EXECUTE | | basic computer | basic computer | | | basic computer |
| 22. $ SET | basic computer | basic computer | basic computer | | | |

Table 2-2. Correspondence to Generic Tool Descriptions

| Generic Tool | Vol. I Reference | GCOS Tool | Vol. II Reference |
|---|---|---|---|
| Post-Mortem Dumps | A.1 | Debug Trace Package | 2.3 |
| | | FAVS | 2.4 |
| | | GEBORT | 2.6 |
| | | General Loader Debug Feature | 2.10 |
| | | JAVS | 2.12 |
| | | $ Execute | 2.21 |
| | | S SET | 2.22 |
| Snapshot Dumps | A.2 | Debug Trace Package | 2.3 |
| | | FAVS | 2.4 |
| | | GECHEK | 2.7 |
| | | General Loader Debug Feature | 2.10 |
| | | GESNAP | 2.11 |
| | | JAVS | 2.12 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| | | $ SET | 2.22 |
| Breakpoint/ Trap Dumps | A.3 | Debug Trace Package | 2.3 |
| | | FAVS | 2.4 |
| | | GECHEK | 2.7 |
| | | General Loader Debug Feature | 2.10 |
| | | JAVS | 2.12 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| Programmed-In Dumps | A.4 | ALGOL Compiler | 2.1 |
| | | COBOL Compiler | 2.2 |
| | | FORTRAN Compiler | 2.5 |
| | | JOVIAL Compiler | 2.13 |
| | | MACRO Assembler | 2.14 |
| | | STRUCTRAN-1 | 2.16 |

Table 2-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | GCOS Tool | Vol. II Reference |
|---|---|---|---|
| Monitor Dumps | A.5 | --- | --- |
| Auxilliary Storage/ Utility Dumps | A.6 | General Loader<br>Utility Package | 2.9<br>2.20 |
| Dynamic Internal Traces | A.7 | Debug Trace Package<br>FAVS<br>JAVS<br>Trace Package | 2.3<br>2.4<br>2.12<br>2.19 |
| Recorded Traces | A.8 | Debug Trace Package<br>FAVS<br>JAVS<br>Trace Package | 2.3<br>2.4<br>2.12<br>2.19 |
| Monitor Traces | A.9 | --- | --- |
| Set-Use Matrix/ Cross Reference Analysis Tool | A.10 | FAVS<br>JAVS<br>$ SET | 2.4<br>2.12<br>2.22 |
| Hardware Monitors | A.11 | --- | --- |
| Monitor Computers | A.12 | --- | --- |
| Computer Emulators | A.13 | --- | --- |
| Computer Simulators | A.14 | --- | --- |
| Graphic Output | A.15 | FAVS<br>JAVS | 2.4<br>2.12 |

Table 2-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | GCOS Tool | Vol. II Reference |
|---|---|---|---|
| Software Breakpoints/ Traps | A.16 | Debug Trace Package | 2.3 |
| | | FAVS | 2.4 |
| | | JAVS | 2.12 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| Hardware Breakpoints | A.17 | --- | --- |
| Reversible Execution/ Backtracking | A.18 | --- | --- |
| Interactive Modification Tools | A.19 | Debug Trace Package | 2.3 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| Recompilation/ Correction | A.20 | ALGOL Compiler | 2.1 |
| | | COBOL Compiler | 2.2 |
| | | FORTRAN Compiler | 2.5 |
| | | JOVIAL Compiler | 2.13 |
| | | MACRO Assembler | 2.14 |
| | | STRUCTRAN-1 | 2.16 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| Hardware Modification Tools | A.21 | --- | --- |

Table 2-2. Correspondnce to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | GCOS Tool | Vol. II Reference |
|---|---|---|---|
| Program Execution-Oriented Recording/ Reduction | A.22 | Debug Trace Package | 2.3 |
| | | FAVS | 2.4 |
| | | GELAPS | 2.8 |
| | | JAVS | 2.12 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |
| Input/Output-Oriented Recording/ Reduction | A.23 | FAVS | 2.4 |
| | | JAVS | 2.12 |
| Test Input List Tools | A.24 | Utility Package | 2.20 |
| | | $ SET | 2.22 |
| Comparator Tools | A.25 | Utility Package | 2.20 |
| Problem Status Reporters | A.26 | --- | --- |
| System Status Summary Tools | A.27 | --- | --- |
| Checkpoint Tools | A.28 | Debug Trace Package | 2.3 |
| | | GECHEK | 2.7 |
| | | RBUG | 2.15 |
| | | TDS | 2.17 |
| | | TPOS | 2.18 |

## 2.1 ALGOL COMPILER

The GCOS ALGOL compiler has an option that produces a symbol reference table which supports debugging.

### 2.1.1 Program State Information Supplied

The symbol reference table can supply static information about traversed paths, symptom locality, and name-space values.

### 2.1.2 Abstract Machine Representation Supported

Compiler-level information is given by the symbol reference table.

### 2.1.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program.

### 2.1.4 Capabilities and Usability Features

Undetermined.

## 2.2 COBOL COMPILER

The GCOS COBOL compiler has options which produce a symbol reference table which supports debugging.

### 2.2.1 Program State Information Supplied

The symbol reference table can supply static information about traversed paths, symptom locality, and name-space values.

### 2.2.2 Abstract Machine Representation Supported

Compiler-level information is given by the symbol reference table.

### 2.2.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program.

### 2.2.4 Capabilities and Usability Features

Undetermined.

## 2.3 DEBUG TRACE PACKAGE

The Debug Trace Package is a comprehensive debugging tool available to GCOS time-sharing users. It allows the user to examine and/or modify data or code, to single step the program, and to insert breakpoints. In addition, it provides eight types of traces.

### 2.3.1 Program State Information Supplied

This tool provides information about traversed paths, symptom locality, name-space value setting, external system status, simultaneous events, space allocation, and timing.

### 2.3.2 Abstract Machine Representation Supported

Debug Trace is a basic computer level tool that can be used with compiler-level or assembler-level programs.

### 2.3.3 Methodology Processes Supported

This tool supports the verification/duplication process by allowing the user to examine the replicated program state directly or to save it so he can employ a comparison tool to compare the replicated program with a copy of the program saved at the time a failure occurred. It supports the localization and symptom accumulation processes by providing a variety of traces as well as by allowing the user to insert breakpoints and manually trace program flow. It also allows the user to examine the setting of name-space values before and after execution as well as during execution when breakpoints or tracing is requested. Debug Trace supports the hypothesis test process by allowing the user to temporarily patch and test the program. The user can insert the patches by modifying instructions or initial data values or by inserting breakpoints or traces and, when they are reached, modify program state values or alter the flow of control.

### 2.3.4 Capabilities and Usability Features

Debug Trace provides sophisticated machine-level debugging. Its various capabilities and features support debugging in the following ways:

- Displays, in data or instruction format, the contents of memory locations and modifies those contents.
- Displays and modifies the contents of registers.
- Converts numbers from decimal to octal and from octal to decimal.
- Finds the location(s) of one or more occurrences of a specified data pattern in memory, with the search commencing at any designated location.

- Inserts conditional and unconditional breakpoints and deletes them.
- Calls another subsystem and passes parameters.
- Executes a specified number of program instructions, starting with the next instruction to be executed, in the controlled environment provided by the trace mechanism.
- Locates the entry point for one or more specified names.
- Saves the current program state and restores a program to its saved state.
- Resumes execution of the program at the next instruction or at a specified location.
- Terminates or aborts program execution.
- Performs tracing by interpretively executing the instructions of the program.
- Allows user to specify the program regions to be traced.
- During tracing, notifies the user of illegal instructions and permits the user to alter or bypass them.
- Provides a transfer trace that displays every instruction, in the specified tracing regions, which results in a transfer of control.
- Provides an operation code trace that displays every occurrence, in the specified tracing regions, of the use of one or more selected operation codes.
- Provides a modifier trace that displays every occurrence, in the specified tracing regions, of the use of one or more selected instruction modifiers.
- Provides a use trace that displays every instruction, in the specified tracing regions, which uses, or references, one or more specified registers and/or memory locations.
- Provides a change trace that displays every instruction, in the specified tracing regions, which changes one or more specified registers and/or memory locations.
- Provides a full trace that displays every instruction executed in the specified tracing regions.

- Provides a map trace that displays a report showing partitioned segments of core and the number of instructions which were executed in each such segment.

- Provides an own code trace that permits a user subroutine to gain control from the trace mechanism before each instruction is executed.

- Allows user to queue only the last (most recent) 25 lines of trace output which would have otherwise been issued to the user's terminal.

- Allows user to selectively display information which has been collected by the trace mechanism.

- Allows user to selectively or collectively initialize queues and frequency counters which are maintained by the trace mechanism.

- Allows user to disengage the trace mechanism.

- Allows user to place the trace mechanism in step mode which returns the user to the command level before each instruction which meets trace conditions is executed.

- Requires that the Debug Trace Package be bound with the target program by including an object deck in the General Loader activity which generates the LODX H* file and requires that the package be invoked at execution time.

## 2.4 FAVS

FAVS (FORTRAN Automated Verification System) is a tool available under GCOS that provides static information about a FORTRAN or DMATRAN program and/or alters the source text of the program to provide dynamic information during program execution. In particular, FAVS provides the following types of information:

- Program paths

- Inter- and intra-module relationships

- Cross reference of all symbols

- Common block and symbol usage

- Inconsistencies in program structure or use of variables

- Statistics about the execution of the program statements.

- Values of variables upon module entry or exit

### 2.4.1 Program State Information Supplied

FAVS provides some information about traversed paths, symptom locality, name-space values and timing.

### 2.4.2 Abstract Machine Representation Supported

FAVS is a compiler-level tool for programs written in FORTRAN or DMATRAN. It uses the source text of the program and symbolic commands as its input and produces outputs in terms of program defined symbols.

### 2.4.3 Methodology Processes Supported

FAVS primarily supports the localization process. It does so by providing static information about where each variable can be set and/or used as well as dynamic information about the values of variables upon program entry and exist. It also identifies the (static) program paths, gives summary information about the paths traversed during program execution and the number of times each statement was executed, and identifies all possible paths to any particular statement. And it indicates any inconsistencies in program structure or variable use that are not detected by the FORTRAN compiler.

### 2.4.4 Capabilities and Usability Features

FAVS provides the following capabilities and features:

- Identifies, by statement number, the path from each decision statement.
- Shows module dependencies both in summary form which identifies the modules that invoke/are invoked by other modules and in detailed form which identifies the specific statements that invoke modules.
- Provides a cross reference listing for all symbols set/used by all modules on the library as well as a summary listing of common block usage.
- Performs checks for possible misuse of constants and variables, for inconsistency in number and type of parameters in procedure invocations, for variable use before set conditions, and for control structure errors such as unreachable code.
- Instruments the program to collect statistics about the number of times each statement is executed and prints those statistics after program execution.
- Inserts output statements in the program to capture the entry and exit values of variables.

- Prints the list of paths from module entry or from a designated statement to a specific statement in the module.

The contents of the object program are altered when instrumentation is inserted in the program. The resultant program, while logically equivalent to its unaltered form, will be larger and take longer to execute because it contains additional object instructions in order to monitor program execution.

## 2.5 FORTRAN COMPILER

The GCOS FORTRAN compiler has two options that support debugging. One option produces a printed symbol reference table for direct use while the other produces a symbol table for use by the General Loader debug feature.

### 2.5.1 Program State Information Supplied

The symbol reference table can supply static information about traversed paths, symptom locality, and name-space values.

### 2.5.2 Abstract Machine Representation Supported

Compiler-level information is given by the symbol reference table.

### 2.5.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program.

### 2.5.4 Capabilities and Usability Features

Undetermined.

## 2.6 GEBORT

GEBORT is a GCOS master mode entry routine which is used to indicate abnormal termination of an activity. It performs program wrapup and prints an optional postmortem dump.

### 2.6.1 Program State Information Supplied

GEBORT provides information about symptom locality, name-space value, external system status, and space allocation.

### 2.6.2 Abstract Machine Representation Supported

GEBORT is a basic computer level debugging tool.

### 2.6.3 Methodology Processes Supported

The localization process is supported by allowing the user to examine the contents of memory after program termination and thus determine the values of variables. The symptom accumulation process is also supported by the memory dump.

### 2.6.4 Capabilities and Usability Features

GEBORT provides the following capabilities and features upon abnormal termination of an activity:

- Releases core storage allocated to the activity.
- Deallocates all peripherals allocated to the activity, issuing dismounting instructions when necessary.
- If requested, prints a postmortem memory dump.
- If all I/O activity is not terminated before calling GEBORT, the I/O may be lost.
- Requires that user include a call to GEBORT in the program and allows user to include a reason code that is printed upon abort.

## 2.7 GECHEK

GECHEK is a GCOS master mode entry routine which initiates check point dumps and sets up a process which enables the requesting program to be read from the last checkpoint made.

### 2.7.1 Program State Information Supplied

GECHEK provides information about symptom locality, name-space value, external system status, simultaneous events, and space allocation.

### 2.7.2 Abstract Machine Representation Supported

GECHEK is a basic computer level debugging tool.

### 2.7.3 Methodology Processes Supported

GECHEK can be used to support localization and symptom accumulation by allowing the user to examine the values of some or all variables used in the program being debugged. It supports the hypothesis test process by allowing the user to take a checkpoint dump, enter a patch during program execution, and then, if the desired results are not obtained by the patch, to try another patch using the program state at the time of the dump.

### 2.7.4 Capabilities and Usability Features

GECHEK has the following features:

- Allows user to obtain one or more checkpoint dumps.
- Sets up bookkeeping system so that program can be read from last checkpoint.

- Requires the user to include a call to GECHEK in the program by either assembling it in or patching it in.

## 2.8 GELAPS

GELAPS is a GCOS master mode entry routine which provides the requesting program with the total amount of processor time the program has expended up to the time of the request.

### 2.8.1 Program State Information Supplied

GELAPS provides timing information.

### 2.8.2 Abstract Machine Representation Supported

GELAPS is a basic computer-level tool.

### 2.8.3 Methodology Processes Supported

The timing information supports the localization, symptom accumulation, and hypothesis test processes when the problem being debugged is a timing problem.

### 2.8.4 Capabilities and Usability Features

GELAPS has the following capabilities and features:

- Returns the elapsed processor time expressed as the number of 1/64 millisecond increments elapsed.
- Returns the time right-justified in the Q-register.
- Requires the user to include code in the program to call GELAPS and to output the returned time value.

## 2.9 GENERAL LOADER

The GCOS General Loader has the following options, selected via a $ OPTION control card, that support debugging:

- MAP
- SYMREF
- SET

The MAP option produces an object program map that provides the correspondence between the compiler-level source program and the basic computer level object program and helps the user interface with basic computer level debugging tools. The SYMREF option produces a symbol reference table such

as produced by the various compilers and assemblers. The SET option allows the user to set allocated memory to any desired octal pattern (rather than zero) and may help the user debug problems such as occur when a value is being used before it is set or when an invalid memory reference is occurring.

(The Generic Loader also has a debug feature which is treated as a tool separate from the General Loader.)

2.9.1 Program State Information Supplied

The symbol reference table can supply static information about symptom locality, name-space values and space allocation.

2.9.2 Abstract Machine Representation Supported

Compiler/assembler-level information (depending upon the source language) is given by the symbol reference table and the object program map.

2.9.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program. The SET option supports the symptom accumulation process by allowing the user to observe program behavior when allocated memory has not been cleared to zero.

2.9.4 Capabilities and Usability Features

Undetermined.

2.10 GENERAL LOADER DEBUG FEATURE

The General Loader Debug Feature is a GCOS debugging tool that enables the user to obtain various dumps. GCOS control cards are used to specify the desired option(s).

2.10.1 Project State Information Supplied

This tool provides information about name-space values, symptom locality, and space allocation. Information about traversed paths may be obtained by requesting a series of dumps at selected points.

2.10.2 Abstract Machine Representation Supported

The General Loader Debug Feature can be used as a basic computer level debugging tool by requesting various dumps. By supplying a symbol table, it can be used as a compiler level debugging tool.

### 2.10.3 Methodology Processes Supported

This tool can support the verification/duplication process by allowing the user to examine the replicated program state. It supports the localization and symptom accumulation processes by providing the user with name-space value setting during program execution.

### 2.10.4 Capabilities and Usability Features

The General Loader Debug Feature has the following capabilities and features:

- Allows user to dump single variables, arrays, and memory locations at any point during program execution.
- Allows user to control the amount of output by specifying conditions and/or iterations on which to dump.
- Allows user, debugging a batch program in the direct access mode, to insert a breakpoint to gain control at a specified location.
- For symbolic debugging, requires that the FORTRAN user generate a debug symbol table by using the STAB option; other users generate the table by using macros and pseudo-options supplied by the GMAP assembler.
- Requires that all debug cards be placed in front of the first object deck of the program or link.

## 2.11 GESNAP

GESNAP is a GCOS master mode entry routine which is used to obtain printouts of selected portions of core storage during debugging.

### 2.11.1 Program State Information Supplied

GESNAP can supply information concerning symptom locality, name-space values, external system status, and space allocation.

### 2.11.2 Abstract Machine Representation Supported

GESNAP is a basic computer level debugging tool.

### 2.11.3 Methodology Processes Supported

GESNAP supports the localization process by allowing the user to examine the program state values. The symptom accumulation process is also supported by the same capability.

### 2.11.4 Capabilities and Usability Features

Undetermined.

## 2.12 JAVS

JAVS (JOVIAL Automated Verification System) as a tool available under GCOS that provides static information about a JOVIAL/J3 program and/or alters the source text of the program to provide dynamic information during program execution. In particular, JAVS provides the following types of information:

- Program paths
- Inter- and intra-module relationships
- Cross reference of all symbols
- Statistics about the execution of the program statements
- Trace of program flow
- Trace of variable values

### 2.12.1 Program State Information Supplied

JAVS provides information about traversed paths, symptom locality, name-space value setting, and some timing information.

### 2.12.2 Abstract Machine Representation Support

JAVS is a compiler-level tool for programs written in JOVIAL. It uses the source text of the program and symbolic commands as its input and produces outputs in terms of program defined symbols.

### 2.12.3 Methodology Processes Supported

JAVS supports the localization process by providing static information about possible program paths and about variable usage and by providing dynamic trace information about actual program paths, number of times statements were executed, timing, and the values of variables. It supports the symptom accumulation process in a similar manner. It supports the execution analysis process by allowing the user to make assertions about the program and then dynamically testing the validity of each assertion.

### 2.12.4 Capabilities and Usability Features

JAVS is a powerful tool that provides the user a good deal of control over the amount and type of debugging information produced, but it does require the user to master a rich command language. JAVS extracts static information

from the text of the source program and instruments the program to provide dynamic information. In particular, it provides the following:

- Provides a cross reference listing for all names in the entire library, showing their usage.
- Shows module dependencies in a number of summary forms which identify the modules that invoke/are invoked by the modules, and in detailed form which identifies the specific statements that invoke modules.
- Produces a stylized picture of the pattern of program flow implicit in the set of decision-to-decision paths.
- Prints the list of paths from module entry or from a designated statement to a specific statement in a module.
- Instruments the program to trace program execution and to print, for each decision-to-decision path, the description, outcome, and relevant source text.
- Instruments the program to collect statistics about the number of times each statement is executed and prints those statistics after program execution.
- Instruments the program to produce a summary of the time spent in executing specified modules.
- Instruments the program to verify the validity of assertions, to monitor that the value of selected variables remains within a given range, and to trace the values of variables.

The contents of the object program are altered when instrumentation is inserted in the program. The resultant program, while logically equivalent to its unaltered form, will be larger and take longer to execute because it contains additional object instructions in order to monitor program execution.

## 2.13 JOVIAL COMPILER

The GCOS JOVIAL compiler has an option that produces a symbol reference table which supports debugging.

### 2.13.1 Program State Information Supplied

The symbol reference table can supply static information about traversed paths, symptom locality, and name-space values.

2.13.2 Abstract Machine Representation Supported

Compiler-level information is given by the symbol reference table.

2.13.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program.

2.13.4 Capabilities and Usability Features

Undetermined.

2.14 MACRO ASSEMBLER

The GCOS MACRO Assembler (GMAP) has an option that produces a symbol reference table which supports debugging.

2.14.1 Program State Information Supplied

The symbol reference table can supply static information about traversed paths, symptom locality, and name-space value setting.

2.14.2 Abstract Machine Representation Supported

Assembler-level information is given by the symbol reference table.

2.14.3 Methodology Processes Supported

The symbol reference table supports the localization process by indicating the usage of all symbols defined in the program.

2.14.4 Capabilities and Usability Features

Undetermined.

2.15 RBUG

RBUG is a GCOS subsystem that provides the user with a dynamic debugging tool for batch programs using a teletype as the conversational device. By using the BREAKPOINT pseudo variable on the General Loader DEBUG control card, the user can request the following actions:

- Obtain snapshot dumps of locations and registers.
- Change the contents of one or more locations or registers.
- Insert or delete breakpoints.
- Continue the execution of the program at the interrupt point.

- Transfer to another point within the program.
- Terminate the execution either by normal termination or abort.

### 2.15.1 Program State Information Supplied

The various options of RBUG provide information about traversed paths, symptom locality, name-space value settings, external system status, simultaneous events, space allocation, and timing.

### 2.15.2 Abstract Machine Representation Supported

RBUG is a basic computer level debugging tool.

### 2.15.3 Methodology Processes Supported

RBUG can be used to support verification/duplication by allowing the user to examine program state information. The breakpoint feature and the various printout capabilities support the localization and symptom accumulation processes by allowing the user to monitor program flow and examine program state values. The ability to change the contents of locations and registers and to transfer control within the program supports the hypothesis test process by allowing the user to make program patches.

### 2.15.4 Capabilities and Usability Features

RBUG has the following capabilities and features:

- Allows user to print information in decimal (integer), octal, double precision floating point, complex, ASCII, Hollerith (BCI), or logical (T or F) representation.
- Allows user to modify the contents of a specific location.
- Allows user to display or modify register contents.
- Allows user to insert and delete breakpoints.
- Allows user to continue program execution, to terminate the program normally, or to abort the program.

## 2.16 STRUCTRAN-1

STRUCTRAN-1 is a tool available under GCOS that translates a program written in DMATRAN into standard FORTRAN. (DMATRAN is a programming language that augments standard FORTRAN to permit its use as a basis for structured programming.) STRUCTRAN-1 indirectly affects debugging by allowing the user to program in a language that supports structured programming and thereby can have a positive affect on program reliability. But it also has a negative affect on debugging in that most of the debugging tools operate on the FORTRAN or object code representation of the program rather than the DMATRAN representation with which the user is familiar.

## 2.17 TDS

TDS (terminal debug subroutine) is a GCOS debugging tool used with a subsystem that is to be checked. It requires only a small amount (2K) of core and allows the user to perform the following operations:

- Gain control at selected locations within the subsystem.
- Display and/or patch specified portions of the subsystem.
- Return normally to the subsystem at an interrupt point.
- Transfer to a stipulated point within the system.
- Set or remove breakpoints during execution of the subsystem.

### 2.17.1 Program State Information Supplied

The various options of TDS give information concerning traversed paths, symptom locality, name-space value settings, external system status, simultaneous events, space allocation, and timing.

### 2.17.2 Abstract Machine Representation Supported

TDS is a basic computer level debugging tool that requires that an assembly-level instruction be inserted into and assembled with the subsystem.

### 2.17.3 Methodology Processes Supported

TDS can be used to support the verification/duplication process by allowing the user to examine program state information. The breakpoint feature and the various display capabilities support the localization and symptom accumulation processes by allowing the user to monitor program flow and examine program state values. The ability to change the contents of locations and registers and to transfer control within the program supports the hypothesis test process by allowing the user to make program patches.

### 2.17.4 Capabilities and Usability Features

TDS has the following capabilities and features:

- Allows user to display and modify the contents of memory locations.
- Allows user to display and modify the contents of registers.
- Allows user to insert conditional or unconditional breakpoints and to delete breakpoints.
- Allows user to continue execution from its current location or to transfer to a specified location.

- Requires that the instruction YED TDS be inserted into and assembled with the subsystem.

- Requires only 2K of core.

## 2.18 TPOS

TPOS (Transaction Processing Operating System) executes under GCOS and provides a vehicle to process in a real-time mode a number of transactions to a wide variety of previously defined data structures, including concurrent execution of the same transaction processing application programs. TPOS has a master terminal capability for debugging, named DBUG, which is accessed via a DAC connect to TPOS. DBUG allows the user to examine and/or modify data or code, to insert breakpoints, and to control the program's flow.

### 2.18.1 Program State Information Supplied

DBUG can provide information about traversed paths, symptom locality, name-space value setting, external system status that is resident in memory, simultaneous events, and space allocation.

### 2.18.2 Abstract Machine Representation Supported

DBUG is a basic computer level tool for programs operating under TPOS.

### 2.18.3 Methodology Processes Supported

DBUG supports the verification/duplication process by allowing the user to examine the replicated program state. It supports the localization process by allowing the user to insert breakpoints as a means to trace program flow and to determine the setting of name-space values during program execution. It also allows the user to determine data values after program execution. DBUG supports the symptom accumulation process in a similar manner. It supports the hypothesis test process by allowing the user to temporarily patch and test the program. The user can insert the patches by modifying instructions or by inserting breakpoints and then, when the breakpoint is reached, change program state values or alter the program flow.

### 2.18.4 Capabilities and Usability Features

DBUG allows the user to request the following:

- Snapshot dumps of memory or process registers

- Insertion of octal patches

- Insertion and subsequent deletion of breakpoints

- Display of location and size associated with a specified assembly symbol, provided the symbol was assembled into the symbol table.

- Process register display or modification
- Return to caller.

## 2.19 TRACE PACKAGE

The trace package available under GCOS provides nine different types of traces which gather program state and traversed path information as the program is being executed. The user selects the type of trace by assembling or patching the appropriate CALL into the program at the point tracing is to begin.

### 2.19.1 Program State Information Supplied

This tool can provide information about traversed paths and symbol locality as well as some information about name-space value setting and simultaneous events.

### 2.19.2 Abstract Machine Representation Supported

The trace package is a basic computer level tool that can be used with both assembler-level programs and with compiler-level programs.

### 2.19.3 Methodology Processes Supported

The various types of traces support the localization and symptom accumulation processes by indicating the paths traversed during program execution and the values of the data referenced by the traced instruction. By allowing the user to call a user's routine before the execution of each instruction, this tool can also provide additional information about name-space values.

### 2.19.4 Capabilities and Usability Features

The trace package provides the following capabilities and features:

- Provides a transfer trace that displays transfer of control within a program.
- Provides a change trace that displays every instruction which changes one or more specified registers and/or memory locations.
- Provides a use trace that displays every instruction which references one or more specified registers and/or memory locations.
- Provides an address modification trace that displays every instruction which uses the type(s) of address modification specified.
- Provides an operation code trace that displays every execution of one or more specified operation codes.

- Provides a full trace that displays every instruction executed.
- Provides a subroutine trace that displays all entries to and exits from up to ten specified subroutines, showing all the arguments and registers when entering and the registers when exiting.
- Provides a map trace that displays a report showng the frequency of use of every instruction.
- Provides an own code trace that permits a user to supply his own code which is performed before the execution of each instruction.
- Requires that the user assemble or patch the appropriate trace CALL into the program at the point at which tracing is to begin.
- Allows user to limit the amount of trace output by queuing only the most recent, user-specified number of words.
- Allows user to request that trace output not start until a specified number of trace snapshots has been reached.
- Allows user to request that the run be aborted after a specified number of trace snapshots has been reached.
- Allows user to control the regions in which tracing is to occur.
- Outputs, for each trace snapshot, the contents of the registers, the instruction counter, the instruction in both assembly language and memory image format, the effective operand address, the trace snap counter, and, if the instruction being traced references a memory location(s), the contents of the referenced location(s).
- Allows only one type of trace to be performed in one run and can be called only once in that run.
- Operates in an interpretive mode.
- Requires the use of another routine, such as MME GESYOT (P*), to output the trace snaps.

## 2.20 UTILITY PACKAGE

The GCOS Utility Pacakge provides peripheral storage device processing capabilities. It permits copying, copying with merge capability, comparing, positioning, and printing.

### 2.20.1 Program State Information Supplied

The various utility functions can provide some information about name-space value settings and external system status which is written on perhipheral storage.

### 2.20.2 Abstract Machine Representation Supported

All of the utility functions provide basic computer level information.

### 2.20.3 Methodology Processes Supported

The copy and compare functions can support the verification/duplication process both by assisting the user in replicating the required program configuration and, by comparing the replicated program with a previously saved copy, verifying that the required configuration was indeed replicated. The print and compare functions can support the localization and symptom accumulation processes by allowing the user to examine the contents or differences in contents of files produced during program execution.

### 2.20.4 Capabilities and Usability Features

The utility functions, accessed by the $ FUTIL control card, have the following features:

- Allow user to specify the number of lines of a file to be copied, compared, or printed.
- Allow user to print portions of two files in ASCII, ASCII and octal, BCD, or BCD and octal format.
- Can be called unconditionally (by use of the $ UTILITY control card) or conditionally upon abnormal program termination (by use of the $ ABORT control card).

## 2.21 $ EXECUTE

$ EXECUTE is a GCOS control card that supports debugging by allowing the user to control the amount of information dumped upon abnormal termination of the program being executed.

### 2.21.1 Program State Information Supplied

The dump produced upon abnormal program termination supplies information about symptom locality, name-space value setting, and space allocation.

### 2.21.2 Abstract Machine Representation Supported

The dump provides basic computer level information.

2.21.3 Methodology Processes Supported

The localization and symptom accumulation processes are supported by the information contained in the dump.

2.21.4 Capabilities and Usability Features

This control card allows the user to select one of two types of dumps:

- Dump of slave core, program registers, and slave program prefix.
- Dump of only program resisters and slave program prefix.

2.22 $ SET

$ SET is a GCOS control card which can be used to set or reset bits in the program switch word to obtain dumps, symbol tables, and other information.

2.22.1 Program State Information Supplied

The symbol table/cross reference option supplies static information about traversed paths, symptom locality, and name-space value settings. The dump options give information about symptom locality and name-space value settings.

2.22.2 Abstract Machine Representation Supported

The various dump options are basic computer level debugging tools while the symbol table/cross reference option is a compiler level debugging tool.

2.22.3 Methodology Processes Supported

The localization process and symptom accumulation process are supported by the various dump options and the symbol table/cross reference capability. The update option can be used to support the verification/duplication process if information from previous runs was saved.

2.22.4 Capabilities and Usability Features

$ SET can be used to request the following types of tasks:

- Obtain dumps.
- Obtain symbol or cross reference tables.
- Obtain source, object, or compressed deck listings.
- Request various compiler options.
- Update files.
- Clear core.

## 3. MULTICS TOOLS

The Multics (Multiplexed Information and Computing Service) system is a general computer system developed at the Massachusetts Institute of Technology in cooperation with Honeywell Information Systems, Inc. General MULTICS features include multiple man-machine interaction, sequential processing of absentee user jobs, and system programming.

Multics debugging features include various dumps, traces, symbol tables, comparisons, maps, and selected location outputs. The specific tools which provide these capabilities are described in the following sections. In accordance with the purpose of this study, the tools are described with respect to the information requirements and steps in the debugging process model presented in Volume I, Sections 2 and 3. The intent of the following sections is not to instruct the reader about the specifics of using each tool, but rather to describe the capabilities of each debugging tool and to relate their capabilities with debugging, as presented in this study. The reader is referred to the MULTICS reference manuals for specific information regarding the use of these debugging tools. Table 3-1 presents a summary of the program state information provided by Multics and the level of abstract machine at which the information is represented. Table 3-2 presents the correspondence between the generic tools described in Appendix A of Volume I and the specific Multics features and tools. In addition to these tools, the study identified tools and techniques available to RADC which, although not usually thought of as debugging tools, can be used to support the debugging process. They are described in Appendix B.

Table 3-1. Program State Information Provided by Multics Tools

| Tool \ Information | Traversed Paths | Symptom Locality | Name-Space Values | External System Status | Simultaneous Events | Space Allocation and Timing |
|---|---|---|---|---|---|---|
| 1. cancel_cobol_program | | | compiler-level | | | |
| 2. change_error_mode | operating system | operating system | | | | |
| 3. COBOL Compiler | | compiler-level | compiler-level | | | compiler-level |
| 4. compare | | | basic computer | | | |
| 5. cumulative_page trace | operating system | | | | | |
| 6. debug | basic computer/compiler-level | basic computer/compiler-level | basic computer/compiler-level | basic computer/compiler-level | basic computer/compiler-level | basic computer/compiler-level |
| 7. display_cobol_run_unit | compiler-level | | | | | compiler-level |
| 8. display_pllio_error | | | operating system | operating system | | |
| 9. dump_segment | | | basic computer | basic computer | | basic computer |
| 10. FORTRAN Compiler | | compiler-level | compiler-level | | | compiler-level |

Table 3-1. Program State Information Provided by Multics Tools (Cont'd)

| Tool \ Information | Traversed Paths | Symptom Locality | Name-Space Values | External System Status | Simultaneous Events | Space Allocation and Timing |
|---|---|---|---|---|---|---|
| 11. page_trace | operating system | | | | | |
| 12. PL/I Compiler | | compiler-level | compiler-level | | | compiler-level |
| 13. probe | compiler-level | compiler-level | compiler-level | compiler-level | compiler-level | compiler-level |
| 14. profile | compiler-level | | | | | compiler-level |
| 15. progress | | | | | | basic computer |
| 16. reprint error | operating system | operating system | | | | |
| 17. run_cobol | | | compiler-level | | | |
| 18. stop_cobol_run | | | compiler-level | | | |
| 19. trace | basic computer | | basic computer | | basic computer | basic computer |
| 20. trace_stack | operating system | operating system | operating system | | operating system | |
| 21. URL/URA * | | | | | | |

* See Section 3.2.1 for discussion of its use in debugging.

Table 3-2. Correspondence to Generic Tool Descriptions

| Generic Tool | Vol. I Reference | Multics Tool | Vol. II Reference |
|---|---|---|---|
| Post-Mortem Dumps | A.1 | cancel_cobol_program | 3.1 |
| | | change_error_mode | 3.2 |
| | | debug | 3.6 |
| | | dump_segment | 3.9 |
| | | probe | 3.13 |
| | | reprint_error | 3.16 |
| | | run_cobol | 3.17 |
| | | stop_cobol_run | 3.18 |
| Snapshot Dumps | A.2 | debug | 3.6 |
| | | probe | 3.13 |
| Breakpoint/Trap Dumps | A.3 | debug | 3.6 |
| | | probe | 3.13 |
| Programmed-In Dumps | A.4 | COBOL Compiler | 3.3 |
| | | FORTRAN Compiler | 3.10 |
| | | PL/I Compiler | 3.12 |
| Monitor Dumps | A.5 | --- | --- |
| Auxiliary Storage/ Utility Dumps | A.6 | compare | 3.4 |
| | | dump_segment | 3.9 |
| Dynamic Internal Traces | A.7 | debug | 3.6 |
| | | probe | 3.13 |
| | | trace | 3.19 |
| Recorded Traces | A.8 | cumulative_page_trace | 3.5 |
| | | display_cobol_run_unit | 3.7 |
| | | page_trace | 3.11 |
| | | trace | 3.19 |
| | | trace_stack | 3.20 |
| Monitor Traces | A.9 | --- | --- |

Table 3-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | Multics Tool | Vol. II Reference |
|---|---|---|---|
| Set-Use Matrix/ Cross Reference Tool | A.10 | COBOL Compiler | 3.3 |
| | | FORTRAN Compiler | 3.10 |
| | | Pl/1 Compiler | 3.12 |
| Hardware Monitors | A.11 | --- | --- |
| Monitor Computers | A.12 | --- | --- |
| Computer Emulators | A.13 | --- | --- |
| Computer Simulators | A.14 | run_cobol | 3.17 |
| | | GCOS Cimulator | * |
| Graphic Output | A.15 | --- | --- |
| Software Breakpoints/ Traps | A.16 | debug | 3.6 |
| | | probe | 3.13 |
| Hardware Breakpoints | A.17 | --- | --- |
| Reversible Execution/ Backtracking | A.18 | --- | --- |
| Interactive Modification Tools | A.19 | debug | 3.6 |
| | | probe | 3.13 |
| | | trace | 3.19 |
| Recompilation/ Correction | A.20 | COBOL Compiler | 3.3 |
| | | debug | 3.6 |
| | | FORTRAN Compiler | 3.10 |
| | | Pl/1 Compiler | 3.12 |
| | | probe | 3.13 |
| Hardware Modification Tools | A.21 | --- | --- |
| Program Execution Oriented Recording/ Reduction | A.22 | debug | 3.6 |
| | | probe | 3.13 |
| | | profile | 3.14 |

* See Section 2 for description of GCOS system.

Table 3-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | Multics Tool | Vol. II Reference |
|---|---|---|---|
| Program Execution Oriented Recording/ Reduction (cont'd) | | progress | 3.15 |
| | | trace | 3.19 |
| Input/Output Oriented Recording/ Reduction | A.23 | change_error_mode | 3.2 |
| | | compare | 3.4 |
| | | display_pllio | 3.8 |
| | | dump_segment | 3.9 |
| | | reprint_error | 3.16 |
| Test Input List Tools | A.24 | compare | 3.4 |
| | | debug | 3.6 |
| | | dump_segment | 3.9 |
| Comparator Tools | A.25 | compare | 3.4 |
| Problem Status Reporters | A.26 | --- | --- |
| System Status Summary Tools | A.27 | Multics Operating System | 3 |
| Checkpoint Tools | A.28 | --- | --- |
| Requirements and Design Analysis | Note 1 | URL/URA | 3.21 |

Note 1: This is not primarily a debugging tool. See Section 3.2.1 for its application to debugging.

## 3.1 CANCEL_COBOL_PROGRAM

Cancel_cobol_program is a Multics command that indirectly supports debugging by allowing the user to retain data associated with the programs in a COBOL run unit after cancelling one or more programs in the current run unit.

### 3.1.1 Program State Information Supplied

Cancel_cobol_program allows the user to preserve name-space values for debugging purposes.

### 3.1.2 Abstract Machine Representation Supported

Cancel_cobol_program is a compiler-level tool for programs written in COBOL and for programs produced by any compiler that provides a meaningful interface with COBOL programs (e.g., PL/I and FORTRAN).

### 3.1.3 Methodology Processes Supported

This tool supports the use of other debugging tools during the verification/duplication, localization, and symptom accumulation processes.

### 3.1.4 Capabilities and Usability Features

Cancel_cobol_program provides the following capabilities and features:

- Allows user to request that the data segment associated with the program be left intact for debugging purposes.
- Restores the data to its initial state the next time the program is invoked in the run unit.

## 3.2 CHANGE_ERROR_MODE

Change_error_mode is a Multics command that allows the user to obtain more information than normally printed by the default handler for system conditions. (Also allows user to obtain abbreviated messages.)

### 3.2.1 Program State Information Supplied

Change_error_mode provides some symptom locality and traversed path information.

### 3.2.2 Abstract Machine Representation Supported

This tool is an operating system level tool.

### 3.2.3 Methodology Processes Supported

Change_error_mode supports the localization and symptom accumulation processes.

### 3.2.4 Capabilities and Usability Features

This tool has the following capabilities and features:

- If error condition was detected in a support procedure, prints the name of that procedure in addition to the name of the most recent user procedure.
- If a segment that signalled a condition (or caused it to be signalled) is bound, prints both the offset relative to the base of the procedure and the offset relative to the base of the segment.

## 3.3 COBOL COMPILER

The Multics COBOL compiler has the following options that support debugging, either directly or indirectly:

- runtime-check
- symbols
- table
- map

The runtime-check option causes the compiler to produce an object program in which parameters are validated according to number and type, bounds are checked on all subscripted references, string references are checked on all variable length string references, and index name modifications are verified for validity. The symbols option causes the compiler to produce a cross-reference listing of all data names defined in the program. The table option generates a full symbol table that supports the operation of Multics symbolic (i.e., compiler-level) debugging commands such as probe, debug, and trace. The map option produces an object program map that provides the correspondence between the compiler-level source program and the basic computer level object program and helps the user interface with basic computer level debugging tools.

### 3.3.1 Program State Information Supplied

Execution of a program compiled with the runtime-check option can provide symptom locality information for problems caused by invalid parameter/subscript references. The cross reference produced by the symbols option can also provide symptom locality, name-space value setting and space allocation information, but this information is static, rather than dynamic, information.

### 3.3.2 Abstract Machine Representation Supported

This tool supports programs written in COBOL.

### 3.3.3 Methodology Processes Supported

The information produced by executing a program compiled with the runtime-check option supports the localization process by identifying where invalid parameter/subscript references occur. It can also support the symptom accumulation process by indicating whether or not the invalid references occur under changed conditions. The information contained in the cross reference produced as a result of the symbols option supports the localization process by indicating the usage of all data names defined in the program.

### 3.3.4 Capabilities and Usability Features

The COBOL compiler options mentioned, with the exception of runtime-check, provide static information and do not alter the contents of the object program. The runtime-check option provides dynamic (i.e., execution) information but it does alter the contents of the object program and can thus cause the program to behave differently than its unaltered form.

## 3.4 COMPARE

Compare is a Multics command that compares segments in the storage system and lists their differences. The comparison is a word-by-word check and can be made with a mask so that only specified parts of each word are compared.

### 3.4.1 Program State Information Supplied

Compare provides information about differences in the setting of name-space values.

### 3.4.2 Abstract Machine Representation Supported

Compare is a basic computer level tool, with inputs and outputs expressed in octal.

### 3.4.3 Methodology Processes Supported

Compare can be used to support the verification/duplication process by verifying that the proper program state was replicated (provided the program state was saved at the time the problem occurred). It can be used to support the symptom accumulation process by printing out the differences in program state information caused by different inputs. It can also be used to support the hypothesis test and problem resolution processes by printing out the differences in program state information resulting from program changes.

### 3.4.4 Capabilities and Usability Features

Compare can be a useful tool for comparing sets of data, especially when the sets contain more than just a small amount of data, provided there exists a one-to-one relationship between the two sets. If this relationship does not exist, this tool may be of little use since it makes no attempt to realign the data once a difference is detected. In particular, compare has the following capabilities and features:

- Allows user to specify a mask to cause partial word comparison.
- Allows user to limit the number of words to be compared.
- Detects the end of a segment and, if the segments being compared are of unequal length, prints the words remaining in the longer segment.
- Prints an error message if a user-specified offset value exceeds the end of the segment.
- Requires that the information to be compared exists within the storage system.
- Does not attempt to realign the two segments when a difference is detected.

## 3.5 CUMULATIVE_PAGE_TRACE

Cumulative_page_trace is a Multics command that accumulates page* trace data so that the total set of pages used during the invocation of a command or subsystem can be determined. The command operates by sampling and reading the system trace array after invocation of a command and at repeated intervals.

### 3.5.1 Program State Information Supplied

Cumulative_page_trace supplies a limited amount of traversed path information by identifying the pages that are referenced. It does not identify the actual paths taken within those pages.

### 3.5.2 Abstract Machine Representation Supported

This command operates on the level of the operating system.

### 3.5.3 Methodology Processes Supported

Cumulative_page_trace can be used to support localization since it provides some information about traversed paths.

### 3.5.4 Capabilities and Usability Features

The use of cumulative_page_trace is very limited due to the low level of abstract machine supported. It should only be used in situations when other tools/techniques have failed to provide any insight into the cause of the problem. The capabilities and features of this command include:

---

* A page is a 1024-word subdivision of a segment.

- Accumulates data from one invocation of the command to the next.
- Outputs data in a tabular format showing all pages that have been referenced by the user's process.
- Allows user to request a printout showing only the page number of each page referenced; or the page number of each page and the number of faults* for each page; or the total number of page faults, total number of segment faults, and number of pages referenced for each segment.
- Allows user to clear primary memory before each invocation of the process that is being traced and after each interrupt.
- Allows user to interrupt execution of the process that is being traced at a user-specified time interval for page fault sampling.
- Allows user to call the process to be traced a specific number of times.
- Allows user to reset the table of accumulated data.
- Allows user to specify a time period after each call to the process during which no tracing is to occur.
- Allows user to accumulate linkage fault information and to print that information.

## 3.6 DEBUG

Debug is a Multics command that provides interactive debugging facilities. It allows the user to examine and/or modify data or code. It also allows the user to insert breakpoints which stop program execution and allow the user to examine and/or modify the program's state and to control the program's flow.

### 3.6.1 Program State Information Supplied

Debug can provide information about traversed paths, symptom locality, name-space value setting, external system status that is resident in the storage system, simultaneous events, and space allocation.

* A fault occurs when a referenced page/segment is not in memory. The Multics system intervenes at that point and processes the page fault by locating the desired page in the storage system and transporting it into main memory.

### 3.6.2 Abstract Machine Representation Supported

Debug is a sophisticated basic computer level tool. It allows the user to examine and modify data, code, machine registers, and the stack. Debug is also a compiler-level tool, allowing symbolic references to data and statement labels. Such usage requires that the program being debugged be compiled using the "table" option. Debug's command language is primarily at the basic computer level.

### 3.6.3 Methodology Processes Supported

Debug supports the verification/duplication process by allowing the user to examine the replicated program state. It supports the localization and symptom accumulation processes by allowing the user to insert breakpoints to trace program flow and to examine the setting of name-space values during as well as before and after program execution. It also allows the user to examine program state values after an error interrupt or after a run-away or looping program has been stopped by issuing a quit signal. Debug supports the hypothesis test process by allowing the user to temporarily patch and test the program. The user can insert the patches by modifying instructions or initial data values or by inserting a breakpoint and, when the breakpoint is reached, modifying program state values or altering program flow.

### 3.6.4 Capabilities and Usability Features

Debug provides sophisticated machine-level debugging. In particular, it allows the user to do the following:

- Examine data or code.
- Modify data or code.
- Set a breakpoint.
- Perform (possibly non-local) transfers.
- Call procedures.
- Trace the stack being used.
- Examine procedure arguments.
- Control and coordinate breakpoints.
- Continue execution after a breakpoint.
- Change the stack reference frame.
- Print machine registers.
- Execute Multics commands.

## 3.7 DISPLAY_COBOL_RUN_UNIT

Display_cobol_run_unit is a Multics command that displays the current state of a COBOL run unit. The minimal information displayed tells which programs compose the run unit. Optionally, more detailed information can be displayed concerning active files, data location, and other aspects of the run unit.

### 3.7.1 Program State Information Supplied

This tool supplies some information about traversed paths and space allocation.

### 3.7.2 Abstract Machine Representation Supported

This is a compiler-level tool for programs written in COBOL and for programs produced by any compiler that provides a meaningful interface with COBOL programs (e.g., PL/I and FORTRAN).

### 3.7.3 Methodology Processes Supported

Display_cobol_run_unit can support the verification/duplication process by supplying information about the replicated program. It can support the localization process, particularly if the problem being debugged is a looping or other timing problem.

### 3.7.4 Capabilities and Usability Features

The capabilities and features of this tool allow the user to do the following:

- Control the amount of information displayed.
- Display current state of the files that have been referenced during the execution of the current run unit.

## 3.8 DISPLAY_PL1IO_ERROR

Display_pl1io_error is a Multics command designed to be invoked after the occurrence of an I/O error signal during a PL/I I/O operation. It describes the most recent file on which a PL/I I/O error was raised and displays diagnostic information associated with that type of error.

### 3.8.1 Program State Information Supplied

This tool can supply some information about name-space values and external system status.

### 3.8.2 Abstract Machine Representation Supported

This tool is an operating system level tool.

### 3.8.3 Methodology Processes Supported

Display_pl1io_error supports the localization process.

### 3.8.4 Capabilities and Usability Features

This tool explains the cause of a PL/I I/O error.

## 3.9 DUMP_SEGMENT

Dump_segment is a Multics command that prints, in octal or hexadecimal format, selected portions of a segment. It can optionally be instructed to print out an edited version of the ASCII, BCD, EBCDIC (in 8 or 9 bits), or 4-bit byte representation.

### 3.9.1 Program State Information Supplied

Dump_segment can provide information about name-space values and external system status which occupy a segment in the storage system. It can also provide some information about space allocation.

### 3.9.2 Abstract Machine Representation Supported

Dump_segment is a basic computer level tool.

### 3.9.3 Methodology Processes Supported

Dump_segment can be used during the verification/duplication process to record the state of the replicated program. It can also be used during the symptom accumulation process to obtain a printout of segments produced during program execution.

### 3.9.4 Capabilities and Usability Features

Dump_segment provides the user with the following capabilities and features:

- Dump of entire segment or a specified portion of the segment with each word optionally preceded by its address.
- Dump formatted in blocks of a specified number of words separated by a blank line.
- Output in octal with 12 octal digits per word or in hexadecimal with either eight or nine hexadecimal digits per word.
- Output in 4-bit byte, BDC, ASCII, 8-bit EBCDIC, or 9-bit EBCDIC in addition to the octal or hexadecimal dump.
- Optional header that identifies the segment being dumped and the date-time.

## 3.10 FORTRAN COMPILER

The Multics FORTRAN compiler has the following options that support debugging, either directly or indirectly:

- subscriptrange
- profile
- table or brief table
- map

The subscriptrange option causes the compiler to produce an object program in which checks are made to determine if subscript values exceed the declared bounds of the dimension. The profile option causes the compiler to produce an object program which gathers statistics about actual program execution. This option supports the use of the Multics profile command which must be used to gain access to the accumulated statistics. The table option generates a full symbol table that supports the operation of Multics symbolic (i.e., compiler-level) debugging commands such as probe, debug, and trace. The map option produces an object program map that provides the correspondence between the compiler-level source program and the basic computer level object program and helps the user interface with basic computer level debugging tools. It also produces a cross reference listing indicating the line on which each name and each program label is declared and a list of lines on which it is used.

### 3.10.1 Program State Information Supplied

Execution of a program compiled with the subscriptrange option can provide symptom locality information for problems caused by invalid subscript references. The cross references listing produced by the map option can also provide symptom locality information, name-space value setting, and space allocation information, but this information is static, rather than dynamic.

### 3.10.2 Abstract Machine Representation Supported

This tool supports programs written in FORTRAN.

### 3.10.3 Methodology Processes Supported

The information produced by executing a program compiled with the subscript-range option supports the localization process by identifying where invalid subscript usage occurs. It can also support the symptom accumulation process by indicating whether or not the invalid usage occurs under changed conditions. The information contained in the cross reference produced as a result of the map option supports the localization process by indicating where data names and program labels are used.

### 3.10.4 Capabilities and Usability Features

The subscriptrange and profile options provide dynamic (i.e., execution) information but, to do so, they alter the contents of the object program. This can cause the program to behave differently than its unaltered form. The other options have no affect on the contents of the object program but the table option does affect the object program size.

## 3.11 PAGE_TRACE

Page_trace is a Multics command that prints a recent history of page faults* and other system events within the calling process.

### 3.11.1 Program State Information Supplied

Page_trace supplies a limited amount of traversed path information by identifying the pages that are referenced. It does not identify the actual paths taken within those pages.

### 3.11.2 Abstract Machine Representation Supported

This command operates on the level of the operating system.

### 3.11.3 Methodology Processes Supported

Page_trace can be used to support the localization process since it provides some information about traversed paths.

### 3.11.4 Capabilities and Usability Features

The use of page_trace is very limited due to the low level of abstract machine supported. It should only be used in situations when other tools/techniques have failed to provide any insight into the cause of the problem. The capabilities and features of this command include:

- Allows user to request output of only a specified number of system events (mostly page faults) or all the entries in the system trace list for the calling process.
- Outputs data in a tabular format showing the type of trace entry, the real time since the previous entry's event occurred, the ring number in which a page fault occurred, the page number for entries where this is appropriate, the segment number for entries where this is appropriate, and the entry or path name of the segment for entries where this is appropriate.
- Prints entry and path names which are the current ones appropriate for the given segment number but may not be correct since segment numbers can be reused within a process but only segment numbers (not entry names or path names) are kept in the trace array.

---

* A page is a 1024-word subdivision of a segment. A page fault occurs when a referenced page is not in memory. The Mutlics system intervenes at that point and processes the page fault by locating the desired page in the storage system and transporting it into main memory.

- Prints events occurring while inside the supervisor.

## 3.12 PL/I COMPILER

The Multics PL/I Compiler has the following options that support debugging, either directly or indirectly:

- profile
- table or brief-table
- map

The profile option causes the compiler to produce an object program which gathers statistics about actual program execution. This option supports the use of the Multics profile command which must be used to gain access to the accumulated statistics. The table option generates a full symbol table that supports the operation of Multics symbolic (i.e., compiler-level) debugging commands such as probe, debug, and trace. The map option produces a listing showing the storage location of each name declared in the program, the storage location of each program statement, and a cross reference listing. The storage location information provides the correspondence between the compiler-level source program and the basic computer level object program and helps the user interface with basic computer level debugging tools.

### 3.12.1 Program State Information Supplied

The listing produced by the map option provides static symptom locality, and name-space values, storage allocation information.

### 3.12.2 Abstract Machine Representation Supported

This tool supports programs written in PL/I.

### 3.12.3 Methodology Processes Supported

The information contained in the cross reference listing produced as a result of the map option supports the localization process by indicating where data names and statement labels are used.

### 3.12.4 Capabilities and Usability Features

The profile option causes the compiler to alter the contents of the object program which can cause the program to behave differently than its unaltered form. The other options have no affect on the contents of the object program but the table option does have an affect on the object program size.

## 3.13 PROBE

Probe is a Multics command that provides symbolic, interactive debugging facilities for programs compiled with the Multics PL/I, FORTRAN, or COBOL compiler. Its features permit a user to interrupt a running program at a particular statement, examine and modify program variables in their initial state or

during execution, examine the stack of block invocations, and list portions of the source program.

### 3.13.1 Program State Information Supplied

Probe can provide information about traversed paths, symptom locality, name-space value setting, external system status that is resident in the storage system, simultaneous events, and space allocation.

### 3.13.2 Abstract Machine Representation Supported

Probe is a compiler-level tool that can be used with programs written in PL/I, FORTRAN, or COBOL. Its commands are also at a compiler-level, using keywords to request probe actions and symbolic names to specify operands. It does require that the program being debugged be compiled using the "table" option.

### 3.13.3 Methodology Processes Supported

Probe supports the verification/duplication process by allowing the user to examine the replicated program state. It supports the localization and symptom accumulation processes by allowing the user to insert breakpoints to trace program flow and to examine the setting of name-space values during as well as before and after program execution. It also allows the user to examine program state values after an error interrupt or after a run-away or looping program has been stopped by issuing a quit signal. Probe supports the hypothesis test process by allowing the user to change program state values or alter program flow during or before program execution and thus perform temporary patching to determine the effect of these changes on program execution and resultant output.

### 3.13.4 Capabilities and Usability Features

Probe provides a wide range of debugging support on a symbolic level. Its use is documented in a clear, easy to understand manner. In particular, probe allows the user to do the following:

- Precede any request or list of requests by a conditional predicate whose value determines if and when the request it modifies will be executed.
- Display the value of a variable, expression, or group of array entries.
- Set the value of a variable or group of array entries to an expression.
- Call a procedure and pass arguments to that procedure.
- Transfer control to a specified statement and initiate program execution at that point.
- Position to a specified source statement or specified character string in the source program.

- Display one or more source statements beginning with the current one where the current one is either the statement at which program execution was suspended or the statement positioned to by the user.
- Trace the stack of procedure and block invocations.
- Display the attributes of a specified variable and the name of the block in which its declaration is found.
- Set a breakpoint before or after a specified statement and execute the request(s) associated with that breakpoint.
- Stop processing after a breakpoint and enter requests from the terminal.
- Delete breakpoints that were set previously.
- Restart a program that has been suspended by a breakpoint or return to command mode.
- Step through the program one statement at a time.
- Execute one or more Multics command lines.

## 3.14 PROFILE

Profile is a Multics command that prints statistics about the execution of each statement in a PL/I or FORTRAN program that was compiled with the "profile" option and then executed.

### 3.14.1 Program State Information Supplied

Profile provides information about all the paths traversed for a given program execution. It also provides some general timing information.

### 3.14.2 Abstract Machine Representation Supported

Profile is a compiler-level tool that can be used with programs written in PL/I or FORTRAN. It does require that the program being debugged be compiled using the "profile" option.

### 3.14.3 Methodology Processes Supported

Profile can support the localization process by identifying all the paths traversed and the number of times each statement was executed. It can support the symptom accumulation process by showing the effect different inputs have on program execution.

### 3.14.4 Capabilities and Usability Features

Profile has the following capabilities and features:

- For each statement in a program, prints number of times the statement was executed, cost of executing the statement measured in number of instructions executed online plus the number of PL/I operators invoked (where each instruction and each operator invocation count as only one unit), and the names of all PL/I operators used by the statement.
- Requires that the content of the object program be altered by the compiler so that statistics are accumulated, which can cause the program to behave differently than its unaltered form.

## 3.15 PROGRESS

Progress is a Multics command that executes a specified command line and prints information about how its execution is progressing in terms of CPU time, real time, and page faults.

### 3.15.1 Program State Information Supplied

Progress supplies some timing information.

### 3.15.2 Abstract Machine Representation Supported

Progress is a basic computer level tool.

### 3.15.3 Methodology Processes Supported

This tool can support the localization and symptom accumulation processes when the problem being debugged is a timing or looping problem.

### 3.15.4 Capabilities and Usability Features

Progress has the following capabilities and features:

- Allows user to control amount and frequency of output.
- Prints number of virtual CPU seconds used so far, number of real seconds used so far, ratio of virtual to real time, incremental virtual time, incremental real time, ratio of incremental virtual to real time, and number of page faults per second of virtual CPU time.

## 3.16 REPRINT_ERROR

Reprint_error is a Multics command that allows the user to obtain more information than normally printed by the system condition handler about a condition that has already been handled and for which stack history is preserved. (Also allows user to obtain abbreviated messages.)

### 3.16.1 Program State Information Supplied

This tool supplies some symptom locality and traversed path information.

### 3.16.2 Abstract Machine Representation Supported

Reprint_error is an operating system level tool.

### 3.16.3 Methodology Processes Supported

This tool supports the localization and symptom accumulation processes.

### 3.16.4 Capabilities and Usability Features

Reprint_error has the following capabilities and features:

- If error condition was detected in a support procedure, prints the name of that procedure in addition to the name of the most recent user procedure.
- If a segment that signalled a condition (or caused it to be signalled) is bound, prints both the offset relative to the base of the procedure and the offset relative to the base of the segment.

## 3.17 RUN_COBOL

Run_cobol is a Multics command that indirectly supports debugging by allowing the user to retain data associated with the programs in a COBOL run unit after a STOP RUN statement is executed in a COBOL object program.

### 3.17.1 Program State Information Supplied

Run_cobol allows the user to preserve name-space values for debugging purposes.

### 3.17.2 Abstract Machine Representation Supported

Run_cobol is a compiler-level tool for programs written in COBOL and for programs produced by any compiler that provides a meaningful interface with COBOL programs (e.g., PL/I and FORTRAN).

### 3.17.3 Methodology Processes Supported

This tool supports the use of other debugging tools during the verification/duplication, localization, and symptom accumulation processes.

### 3.17.4 Capabilities and Usability Features

Run_cobol provides the following capabilities and features:

- Allows user to set one or more of the eight COBOL defined "external switches".

- Allows user to handle the signal caused by the execution of a STOP RUN statement himself using other Multics commands.

## 3.18 STOP_COBOL_RUN

Stop_cobol_run is a Multics command that indirectly supports debugging by allowing the user to retain data associated with the programs in a COBOL run unit after termination of the current run unit.

### 3.18.1 Program State Information Supplied

Stop_cobol_run allows the user to preserve name-space values for debugging purposes.

### 3.18.2 Abstract Machine Representation Supported

Stop_cobol_run is a compiler-level tool for programs written in COBOL and for programs produced by any compiler that provides a meaningful interface with COBOL programs (e.g., PL/I and FORTRAN).

### 3.18.3 Methodology Processes Supported

This tool supports the use of other debugging tools during the verification/duplication, localization, and symptom accumulation processes.

### 3.18.4 Capability and Usability Features

Stop_cobol_run has the following capabilities and features:

- Allows user to retain the value of all data referenced in the run unit in its last used state.

## 3.19 TRACE

Trace is a Multics command that lets the user monitor all calls to a specified set of external procedures.

### 3.19.1 Program State Information Supplied

Trace provides some information about traversed paths, name-space value setting, simultaneous events, and timing.

### 3.19.2 Abstract Machine Representation Supported

Trace is primarily a basic computer level tool for external procedures compiled by the Multics PL/I or FORTRAN compiler.

### 3.19.3 Methodology Processes Supported

Trace supports the localization and symptom accumulation processes by tracing program flow in terms of procedure invocation and by printing the values of the procedure's arguments. It can also support these processes and the

hypothesis test process when the problem being debugged involves timing by indicating the amount of time spent in each procedure as a result of varying inputs and changing the program code.

### 3.19.4 Capabilities and Usability Features

This tool allows the user to request the following:

- Printout of the arguments on procedure entry, exit, or both.
- Stop at procedure entry, exit, or both.
- Control the frequency with which messages are printed.
- Execute a Multics command line at procedure entry, exit, or both.
- Meter the time spent in the various procedures being monitored.
- Watch the contents of a set of memory cells and print the values of those cells when a value is changed (values are checked at every entry to and exit from every traced procedure).

## 3.20 TRACE_STACK

Trace_stack is a Multics command that prints a detailed explanation of the current process' stack history, printing the most recent entry first.

### 3.20.1 Program State Information Supplied

Trace_stack supplies information about symptom locality and a limited amount of information about traversed paths, name-space value setting, and simultaneous events.

### 3.20.2 Abstract Machine Representation Supported

Trace_stack is primarily an operating system level tool. It does attempt to locate and print the last source line associated with the last instruction executed in each procedure but to do this, it requires that the program being debugged be compiled using the "table" option.

### 3.20.3 Methodology Processes Supported

Trace_stack supports the localization and symptom accumulation processes by identifying the address at which a fault/error occurred and the sequence of procedure calls that led to that address.

### 3.20.4 Capabilities and Usability Features

This command is most useful after a fault or other error condition. It has the following capabilities and features:

- For each stack frame, prints all available information about the procedure which established the frame including, if possible, the text of the source statement last executed, the arguments to that (the owning) procedure, and the condition handlers established.

- Prints the machine registers at the time of the fault, an explanation of the fault, and the text of the source line in which it occurred, if possible.

### 3.21 URL/URA

URL/URA, and its associated utility programs (dump, restore, pr23, data-base-statistics, usage-monitor-user, usage-monitor-command and specification-generator)* form a requirements analysis support and specification generation subsystem operating under control of the Multics operating system, which supports integration-level debugging, both directly and indirectly.

URL/URA is also known as CADSAT** (Computer-Aided Design and Specification Analysis Tool). URL, which stands for User Requirements Language, is a HOL-like language which can be used to describe a computer programming system and its interrelationships. URA, which stands for User Requirements Analyzer, is a programming system which checks the syntax of the URL statements, calls a data base management system to enter the statements in a data base, and produces reports about that data base which analyzes it for consistency and completeness, and which provide information about functional interrelationships. The associated utility programs provide data-base upkeep capabilities and generalized specification generation capabilities.

The CADSAT system, unlike most other systems described in this document, is not, by itself a debugging tool. That is, it does not display details of the software's operations before, during and/or after the occurrence of an error. It is, rather, what might be called an indirect debugging tool. If the software development methodology used include extensive use of URL/URA to produce specifications and aid in requirements and design analysis, then the benefits to the debugging analyst noted in Section 3.21.3 below will be attained.

---

* Version 3.3 of URL/URA is described.

** These two terms will be used interchangably except that URL/URA will always be used when referring to the programs as programs.

### 3.21.1 Program State Information Supplied

None.

### 3.21.2 Abstract Machine Representation Supported

Not applicable.

### 3.21.3 Methodology Processes Supported

URL/URA supports the following processes in the debugging methodology:

- Localization: By providing application software description information in a convenient manner, URL/URA aids the debugging analyst in isolating the reported error. The analyst must go through a process in which he, by analyzing the symptoms of the reported error, must determine its source. URL/URA provides him not only with information about the specified requirement that is being violated, but also with information which aids him in tracing that requirement through design to the module or modules that may be operating incorrectly.

- Execution Analysis: In this process, the analyst makes a series of assertions about the operations of the program. If URL/URA is used only to produce a Requirements Specification, it aids indirectly in this process by providing information about the function the program is to perform. If it is also used to support the Design Specification process, it can help to provide direct information about the assertions to be made.

- Hypothesis Test: URL/URA aids in hypothesis testing by providing ready access to and traceability among the Application Program System Descriptions.

### 3.21.4 Capabilities and Usability Features

CADSAT is a stand-alone system which need not even be run on the same computer as the system under development in order to provide useful information. Its operation, therefore, has no effect on the object program.

## 4. SEL TOOLS

The SEL (Systems Engineering Laboratories) 85/86 Real Time Monitor is a disc-oriented system which supports the execution of multiple programs under a sixty-four level software priority system. This multiprogramming environment supports concurrent execution of multi-foreground programs along with one background (batch) job.

The reader is advised that the SEL 85/86 system is a forerunner of the SEL 32 series of systems and all of the debugging tools described in the following sections, except DEBUGGER, FDP (FORTRAN Debug), LIST, and TIME, are available as both the SEL 85/86 and the SEL 32 systems. DEBUGGER, FDP (FORTRAN Debug), LIST, and TIME are available only on the SEL 85/86 system.

With respect to debugging, the SEL system offers capabilities which allow for various dumps, traces, comparisons, maps, symbol tables, and selected location outputs. The specific tools which provide these capabilities are briefly described in the following sections. The tools are described with reference to the information requirements and steps of the debugging process model presented in Volume I, Sections 2 and 3, of this debugging study. It is emphasized that no attempt has been made to instruct the reader about the specifics of actually using the SEL system and its various debugging tools. To obtain these instructions, the reader is referred to the SEL Reference Manuals.

Table 4-1 presents a summary of the program state information provided by SEL tools and the level of abstract machine at which the information is represented. Table 4-2 presents the correspondence between the generic tools described in Appendix A of Volume I and the specific SEL tools.

Table 4-1. Program State Information Provided by SEL Tools

| TOOLS / INFORMATION | TRAVERSED PATHS | SYMPTOM LOCALITY | NAME-SPACE VALUES | EXTERNAL SYSTEM STATUS | SIMULTANEOUS EVENTS | SPACE ALLOCATION AND TIMING |
|---|---|---|---|---|---|---|
| 1. Cataloger | compiler/ assembler level | | compiler/assembler level | | | compiler/assembler level |
| 2. COREDUMP | | | basic computer | | | |
| 3. DEBUG | basic computer | basic computer | basic computer | basic computer | basic computer | basic computer |
| 4. DEBUGGER | basic computer | | basic computer | | | |
| 5. DUMP | | | basic computer | | | |
| 6. EXAMINE | | | basic computer | | | |
| 7. FDP | compiler-level | | compiler-level | | | |
| 8. FILL | | | basic computer | | | |
| 9. FORTRAN IV Compiler | | | compiler-level | | | |
| 10. LIST | | | basic computer | | | |
| 11. MACRO Assembler | assembler-level | assembler-level | assembler-level | | | |
| 12. Media Conversion | | | basic computer | basic computer | | |
| 13. MODIFY | | | | | | |
| 14. SEARCH | | | basic computer | | | |
| 15. SNAP | | | basic computer | | | |
| 16. TIME | | | | | | operating system |

Table 4-2. Correspondence to Generic Tool Descriptions

| Generic Tool | Vol. I Reference | SEL Tool | Vol. II Reference |
|---|---|---|---|
| Post-Mortem Dumps | A.1 | COREDUMP | 4.2 |
| | | DEBUG | 4.3 |
| | | DUMP | 4.5 |
| | | EXAMINE | 4.6 |
| Snapshot Dumps | A.2 | DEBUG | 4.3 |
| | | DEBUGGER | 4.4 |
| | | DUMP | 4.5 |
| | | SNAP | 4.15 |
| Breakpoint/ Trap Dumps | A.3 | DEBUG | 4.3 |
| | | DEBUGGER | 4.4 |
| | | FDT | 4.7 |
| Programmed-In Dumps | A.4 | FORTRAN Compiler | 4.9 |
| Monitor Dumps | A.5 | COREDUMP | 4.2 |
| | | DEBUG | 4.3 |
| | | DEBUGGER | 4.4 |
| | | DUMP | 4.5 |
| | | SNAP | 4.15 |
| Auxilliary Storage/ Utility Dumps | A.6 | LIST | 4.9 |
| | | Media Conversion Processor | 4.12 |
| Dynamic Internal Traces | A.7 | DEBUG | 4.3 |
| | | FDP | 4.7 |
| Recorded Traces | A.8 | --- | --- |

Table 4-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | SEL Tool | Vol. II Reference |
|---|---|---|---|
| Monitor Traces | A.9 | --- | --- |
| Set-Use Matrix/ Cross Reference Analysis Tool | A.10 | MACRO Assembler | 4.11 |
| Hardware Monitors | A.11 | --- | --- |
| Monitor Computers | A.12 | --- | --- |
| Computer Emulators | A.13 | --- | --- |
| Computer Simulators | A.14 | --- | --- |
| Graphic Output | A.15 | --- | --- |
| Software Breakpoint/ Traps | A.16 | DEBUGGER | 4.4 |
| Hardware Breakpoints | A.17 | --- | --- |
| Reversible Execution/ Backtracking | A.18 | --- | --- |
| Interactive Modification Tools | A.19 | DEBUGGER<br>EXAMINE<br>FDP | 4.4<br>4.6<br>4.7 |

Table 4-2. Correspondence to Generic Tool Descriptions (Cont'd)

| Generic Tool | Vol. I Reference | SEL Tool | Vol. II Reference |
|---|---|---|---|
| Recompilation/ Correction | A.20 | DEBUG | 4.3 |
| | | DEBUGGER | 4.4 |
| | | FILL | 4.8 |
| | | MACRO Assembler | 4.11 |
| | | MODIFY | 4.13 |
| | | FORTRAN Compiler | 4.9 |
| Hardware Modification Tools | A.21 | --- | --- |
| Program Execution-Oriented Recording/ Reduction | A.22 | FDP | 4.7 |
| Input/Output Oriented Recording/ Reduction | A.23 | --- | --- |
| Test Input List Tools | A.24 | LIST | 4.9 |
| | | Media Conversion Processor | 4.12 |
| Comparator Tools | A.25 | Media Conversion Processor | 4.12 |
| | | SEARCH | 4.14 |
| Problem Status Reporters | A.26 | --- | --- |
| System Status Summary Tools | A.27 | Cataloger | 4.1 |
| | | TIME | 4.16 |
| Checkpoint Tools | A.28 | --- | --- |

## 4.1 CATALOGER

The SEL Cataloger, which creates load modules from programs in object code format, supports debugging by producing a module map and an optional symbol table listing. The module map provides the correspondence between the compiler- or assembler-level source program and the basic computer level object program and helps the user interface with basic computer level debugging tools. The symbol table identifies the external symbols referenced by each module.

### 4.1.1 Program State Information Supplied

The Cataloger supplies some static information about traversed paths, name-space value setting, and space allocation.

### 4.1.2 Abstract Machine Representation Supported

Compiler/assembler-level information is given by the module map and the symbol table.

### 4.1.3 Methodology Processes Supported

The symbol table supports the localization process by identifying where external symbols are referenced. The module map support the use of basic computer level debugging tools during various processes.

### 4.1.4 Capabilities and Usability Features

The symbol table output by the cataloger is in a format similar to object records output by the assembler.

## 4.2 COREDUMP

COREDUMP is a stand-alone SEL program whose function is to dump contents of specified memory locations. This program is part of the SEL User's Group Software Library.

### 4.2.1 Program State Information Supplied

COREDUMP provides name-space values information.

### 4.2.2 Abstract Machine Representation Supported

COREDUMP is a basic computer level debugging tool.

### 4.2.3 Methodology Processes Supported

COREDUMP supports the localization and symptom accumulation processess by allowing the user to determine the value of variables used in the program being debugged.

### 4.2.4 Capabilities and Usability Features

COREDUMP outputs the specified memory locations in hexadecimal.

## 4.3 DEBUG

DEBUG is a SEL background (i.e., batch) processor that allows the user to obtain memory dumps, insert snapshot dumps, insert breakpoints, modify the contents of registers or memory, fill portions of memory with a specified value, and trace program execution.

### 4.3.1 Program State Information Supplied

DEBUG can provide information about traversed paths, symptom locality, name-space value setting, external system status, simultaneous events, and space allocation.

### 4.3.2 Abstract Machine Representation Supported

DEBUG is a basic computer level tool requiring that memory locations and values be specified in hexadecimal and producing output in hexadecimal.

### 4.3.3 Methodology Processes Supported

DEBUG supports the verification/duplication process by allowing the user to examine the replicated program state. It supports the localization and symptom accumulation processes by allowing the user to trace program execution and to examine name-space value setting before, during, and after program execution. DEBUG supports the hypothesis test process by allowing the user to temporarily patch program instructions and data values and to observe effect on program execution.

### 4.3.4 Capabilities and Usability Features

DEBUG provides the user with the following capabilities and features:

- Dumps user's program status word, general purpose registers, and specified core locations.
- Fills all memory locations between, and including, specified limits with a specified value.
- Modifies one or more general purpose registers or memory locations with specified value(s).
- Performs snapshot dumps at specified locations, dumping one or more locations.
- Stops program execution at a specified address and processes user directives.

- Resumes program execution after a stop
- Traces program execution, providing a comprehensive listing of all activity occurring within the portion of the program being traced.

## 4.4 DEBUGGER

The DEBUGGER program is part of the SEL User's Group Software Library. This processor or subroutine gives the user control of his program and allows the user to examine and/or change memory and register contents, set snapshot requests with multiple dump ranges and execution count, insert breakpoints, continue program execution after a breakpoint, and make conditional DEBUGGER requests.

### 4.4.1 Program State Information Supplied

DEBUGGER gives name-space values information and can be used to get information about traversed paths. The conditional request can also be used to give information about traversed paths.

### 4.4.2 Abstract Machine Representation Supported

The DEBUGGER program is a basic computer level debugging tool.

### 4.4.3 Methodology Processes Supported

DEBUGGER supports the verification/duplication process by allowing the user to replicate a previous program state (provided data from the previous program state has been retained) and verify that it was properly replicated. It supports the localization and symptom accumulation processes by allowing the user to determine the values of the variables used and to observe the path a certain set of input data takes. It supports the hypothesis test process by allowing the user to make temporary program patches and observe the results.

### 4.4.4 Capabilities and Usability Features

The DEBUGGER program has the following capabilities:

- Address request defines the upper and lower memory limits of the user program in core.
- Current user base can be changed.
- Values in memories and registers can be changed.
- Breakpoints can be set and execution continued.
- Dumps may be requested.
- Conditional commands may be given.

## 4.5 DUMP

DUMP is a SEL input command whose function is to output the contents of specified memory locations. The output is in side-by-side ASCII-coded hexidecimal with ASCII format. DUMP can also output a dump upon a program abort.

### 4.5.1 Program State Information Supplied

DUMP provides name-space value setting information.

### 4.5.2 Abstract Machine Representation Supported

DUMP is a basic computer level debugging tool.

### 4.5.3 Methodology Processes Supported

DUMP supports the localization and symptom accumulation process by enabling the user to examine the contents of specified word locations.

### 4.5.4 Capabilities and Usability Features

DUMP has the following features:

- Dumps memory locations specified by starting and ending addresses.
- Can be requested to dump the program if it is aborted.

## 4.6 EXAMINE

EXAMINE is a SEL input command whose function is to output the contents of specified memory words.

### 4.6.1 Program State Information Supplied

EXAMINE gives name-space values information.

### 4.6.2 Abstract Machine Representation Supported

EXAMINE is a basic computer level debugging tool utilizing hexadecimal input and output.

### 4.6.3 Methodology Processes Supported

EXAMINE supports the localization and symptom accumulation processes by enabling the user to inspect the contents of selected memory words.

### 4.6.4 Capabilities and Usability Features

EXAMINE displays the hexidecimal contents of a specified word when its hexadecimal word address is given.

## 4.7 FDP (FORTRAN Debug)

FDP is a program in the SEL User's Group Software Library which provides a complete or selective trace of FORTRAN programs. The trace is output at run time via logical stream.

### 4.7.1 Program State Information Supplied

FPD provides information about traversed paths and name-space values.

### 4.7.2 Abstract Machine Representation Supported

FDP is a compiler-level debugging tool for programs written in FORTRAN.

### 4.7.3 Methodology Processes Supported

FDP supports the localization and symptom accumulation processes by allowing the user to observe the paths taken by some or all of the variables used in the program being debugged.

### 4.7.4 Capabilities and Usability Features

FDP has the following features:

- At run time, tracing can be selectively started and stopped at locations within the program.
- Selected variables may be omitted or included in the trace.
- Commands may be inserted at run time.

## 4.8 FILL

FILL is a SEL input command which puts a specified value into a selected area of memory.

### 4.8.1 Program State Information Supplied

FILL can provide some name-space values information.

### 4.8.2 Abstract Machine Representation Supported

FILL as a basic computer level debugging tool.

### 4.8.3 Methodology Processes Supported

FILL can support the verification/duplication process by enabling the user to replicate a previous program state provided data from the previous program state has been retained. It can support the localization process when debugging problems such as occur when a value is used before it is set or when an invalid memory reference is made.

4.8.4 Capabilities and Usability Features

All arguments to the FILL command must be given in hexidecimal. The FILL command response is a line feed.

4.9 FORTRAN IV COMPILER

The SEL FORTRAN IV Compiler consists of an initialization routine and three overlay segments. The functions of the overlay segments are syntactical and expression analysis, code optimization and generation, and output of binary text and chaining of externals. The FORTRAN IV Compiler supports debugging by allowing for programmed-in dumps.

4.9.1 Program State Information Supplied

This tool provides information about name-space values.

4.9.2 Abstract Machine Representation Supported

The FORTRAN IV Compiler is a compiler-level debugging tool.

4.9.3 Methodology Processes Supported

The FORTRAN IV Compiler supports the verification/duplication process by allowing the user to duplicate a previous program state if data from the previous program state has been retained. The localization and symptom accumulation processes are also supported by enabling the user to examine dumps and thus determine the values of the various variables used in the program being debugged.

4.9.4 Capabilities and Usability Features

The FORTRAN IV Compiler allows for programmed-in dumps.

4.10 LIST

LIST is a SEL routine which lists any blocked or unblocked file on disc to the line printer. This function can be used for debugging by allowing the debugging analyst to examine the contents of certain files.

4.10.1 Program State Information Supplied

LIST gives information about name-space values.

4.10.2 Abstract Machine Representation Supported

LIST is a basic computer level debugging tool.

4.10.3 Methodology Processes Supported

LIST supports the localization and symptom accumulation processes by allowing the user to inspect the contents of selected files and thus determine the values of the variables used or produced in the program being debugged.

#### 4.10.4 Capabilities and Usability Features

LIST has the following features:

- LIST will print blocked files until EOF is reached.
- Since unblocked files have no EOF, LIST will continue to print the entire contents of the file. An ABORT LIST command will suppress the printing of undesired material.

### 4.11 MACRO ASSEMBLER

The SEL MACRO Assembler has an option that produces a cross reference table which supports debugging.

#### 4.11.1 Program State Information Supplied

The cross reference listing provides static information about traversed paths, symptom locality, and name-space values.

#### 4.11.2 Abstract Machine Representation Supported

The cross reference listing supplies assembler-level information.

#### 4.11.3 Methodology Processes Supported

The cross reference listing supports the localization process by indicating the usage of all symbols defined in the program.

#### 4.11.4 Capabilities and Usability Features

Undetermined.

### 4.12 MEDIA CONVERSION PROCESSOR

The SEL Media Conversion Processor, which operates as a system processor under the Real-Time Monitor, has two options that support debugging:

- VERIFY
- DUMP

The VERIFY option compares two files on a record-by-record basis. The DUMP option outputs the contents of a file in ASCII-coded hexadecimal.

#### 4.12.1 Program State Information Supplied

Both the VERIFY and DUMP can provide information about name-space values and external system status that are file resident.

#### 4.12.2 Abstract Machine Representation Supported

The Media Conversion Processor is a basic computer level debugging tool.

4.12.3 Methodology Processes Supported

VERIFY and DUMP support the localization and symptom accumulation processes by enabling the user to examine the contents of specified files.

4.12.4 Capabilities and Usability Features

This tool has the following features:

- VERIFY compares, record by record, one file with another with this activity terminating whenever an end-of-file is encountered on either file. Record numbers which do not compare are printed.
- DUMP produces an ASCII-coded hexadecimal dump of the requested file.

4.13 MODIFY

MODIFY is a SEL input command which allows any selected core memory word to be reset by the specified value under control of the specified mask.

4.13.1 Program State Information Supplied

None.

4.13.2 Abstract Machine Representation Supported

MODIFY is a basic computer level debugging tool.

4.13.3 Methodology Processes Supported

MODIFY supports the verification/duplication process by enabling the user to change the contents of selected core memory words. It also supports the hypothesis test process by allowing the user to make temporary patches to the program.

4.13.4 Capabilities and Usability Features

All arguments of the MODIFY command must be given in hexidecimal.

4.14 SEARCH

SEARCH is a SEL input command which causes core memory to be searched within specified addresses for a specified value under control of a specified mask. A logical "AND" is performed between the memory word and mask word and this result is compared to the value. If equality holds, the memory address and contents are printed.

4.14.1 Program State Information Supplied

SEARCH gives name-space values information.

4.14.2 Abstract Machine Representation Supported

SEARCH is a basic computer level debugging tool.

#### 4.14.3 Methodology Processes Supported

SEARCH supports the localization and symptom accumulation processes by allowing the user to search portions of core memory for selected value contents.

#### 4.14.4 Capabilities and Usability Features

SEARCH has the following features:

- All SEARCH arguments (i.e., starting address, ending address, comparison values, and word mask) must be given in hexidecimal.
- The SEARCH response is in hexadecimal.

### 4.15 SNAP

SNAP is a SEL input command whose function is to dump the contents of specified blocks of word locations. The output is in hexidecimal.

#### 4.15.1 Program State Information Supplied

SNAP provides information about name-space value settings.

#### 4.15.2 Abstract Machine Representation Supported

SNAP is a basic computer level debugging tool.

#### 4.15.3 Methodology Processes Supported

SNAP supports the localization and symptom accumulation processes by allowing the user to inspect the contents of selected locations.

#### 4.15.4 Capabilities and Usability Features

SNAP outputs, in hexidecimal, the contents of locations between two specified addresses.

### 4.16 TIME

TIME is a SEL User's Group Library program that provides timing information for job steps. The user receives the current system time which is recorded by the interrupt counter.

#### 4.16.1 Program State Information Supplied

TIME gives timing information.

#### 4.16.2 Abstract Machine Representation Supported

TIME is an operating system debugging tool.

### 4.16.3 Methodology Processes Supported

TIME supports the localization and symptom accumulation processes by letting the user observe the time involved in the execution of selected job steps or of selected groups of job steps.

### 4.16.4 Capabilities and Usability Features

The TIME program can be called by Operator Communications or by the appropriate control cards inserted within a program.

## APPENDIX A: A DEBUGGING PROCESS MODEL

The debugging process model defined in Volume I, Section 3, is repeated here to make Volume II a self-contained volume.

### 3. THE DEBUGGING METHODOLOGY

This section describes a methodology to be followed for debugging software errors in integration-level testing/debugging. It provides a generic description of the activities required of a debugging analyst for isolating and resolving an observed software error. The debugging methodology presented is based upon a structured and disciplined approach to software development. Two necessary components of this approach which must exist for the debugging methodology to be applicable are:

- Existence of the following information components:
  - Application system descriptions.
  - Abstract machine descriptions.
  - Run-time information reflecting actual program performance.
- A hierarchically designed software system in which single functionality has been allocated to program components that, in addition, interact with each other in a well-defined manner, and the operation of one component does not depend upon the internal structure of another component for its correct performance.

Section 3.1 presents an overview of integration-level testing in order to provide the context in which debugging, as indicated by the methodology, takes place. Section 3.2 presents an overview of the flow of information within the debugging process. It is presented to clarify the information relationships. The debugging methodology is presented in Section 3.3 by way of a process model. This model defines a sequence of activities to be performed during each instance of debugging.

### 3.1 OVERVIEW OF INTEGRATION-LEVEL TESTING

The integration testing described in this report is part of computer programming test and evaluation (CPT&E). (Figure 2-2 depicts the Full-Scale Development Phase relationships with integration testing/debugging.) CPT&E tests are tests conducted prior to and in parallel with preliminary or formal qualification tests. CPT&E tests are oriented to support the design and development process and the formal acceptance testing--preliminary qualification tests (PQTs) and formal qualification on tests (FQTs). As used in the context of this report, CPT&E testing includes, first, module-level testing and second, integration-level testing. Whereas module-level testing refers to the testing applied to a program component on a stand-alone basis, integration-level testing is testing applied to multiple program components. Integration-level testing may refer to combining sub-modules of a computer program component (CPC), or CPCs with CPCs, depending upon the definition and use of these terms by the development contractor. Further, integration-level testing, as referred to in this report, implies that the program components (i.e., modules and/or CPCs) are combined and tested on an incremental basis. Incremental integration testing focuses on:

- A sequential integration of functionally-related program components.
- Using outputs of one program component as inputs to the next.
- Verifying that the combined program components operate as designed and according to performance requirements.
- A planned and documented testing procedure which defines the relationship between inputs and required results.

The organized management of testing and debugging requires that a document such as a software discrepancy report be used to aid in controlling detection and resolution of the error. This documentation of software errors is essential when the testing responsibility is separate from the debugging responsibility. The software discrepancy report links the description of the observed error with an identification of the test configuration which together define the test environment which demonstrated the problem. (The software discrepancy report is described in Section 2.3.1.) The test configuration includes all that information which identifies the test cases and abstract machine, (i.e., hardware/software components) used during the test run. Status of the test configuration is necessary for problem duplication and resolution. (Section 3.4.3 of Volume III of this study discusses techniques for reconstructing the test configuration.)

Testing methodology is described in more detail in Section 3.1.7 of Volume III of this study. In addition, test plans and procedures are described in Sections 2.1.1 and 2.1.3 of this volume.

## 3.2 OVERVIEW OF THE INFORMATION FLOW

The methodology for software debugging is primarily based on the identification and use of the various structures of information needed to debug a software error. While the debugging process model attempts to offer a plausible view of the types of human thought activities needed in isolating and resolving a software error, it presents a rather primitive view of these activities. The information components upon which the analyst bases the problem solving activities are not primitive in nature, however, and are identified in relationship to the human thought activities. These information components, while generic in nature, must exist in one form or another for each particular system undergoing integration testing/debugging. Figure 3-1, Debugging Process/Information Components presents a detailed view of the flow of information (data) within the model. Figure 3-2 presents a simplified view of only the information components, and the manner in which they depend upon each other for their existence.

Existing before each instance of debugging are management information, the descriptions of the application software, abstract machines, and run-time information, specifically the software discrepancy report. These are all non-derived components of the debugging methodology, i.e., they are established during the software system life cycle before the need for debugging arises. During each instance of debugging, other information components are either derived by processes of the model which involve human understanding, or by use of software/hardware tools which collect and display information.

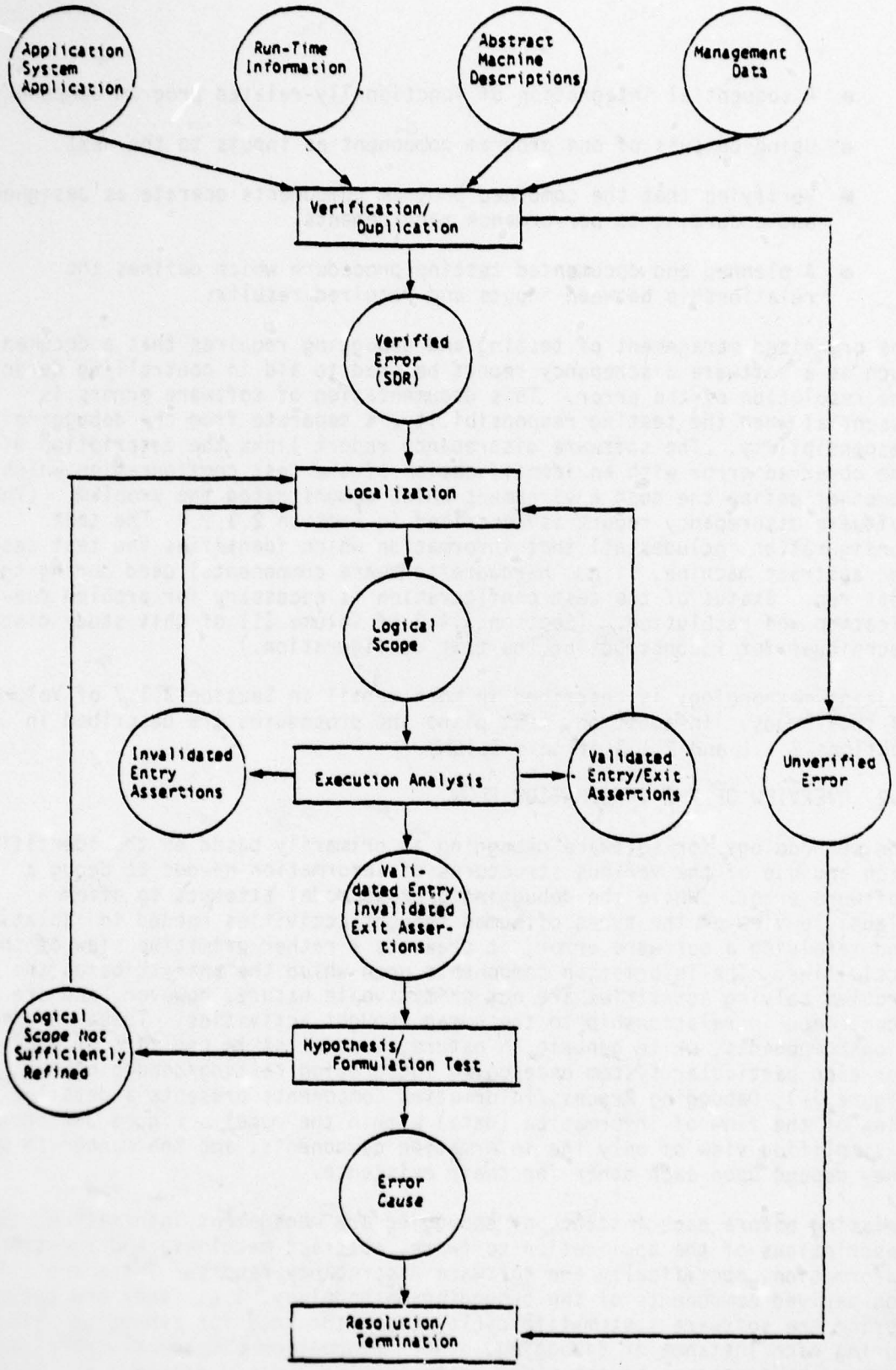



Figure 3-1. Data Flow in Debugging Process Model

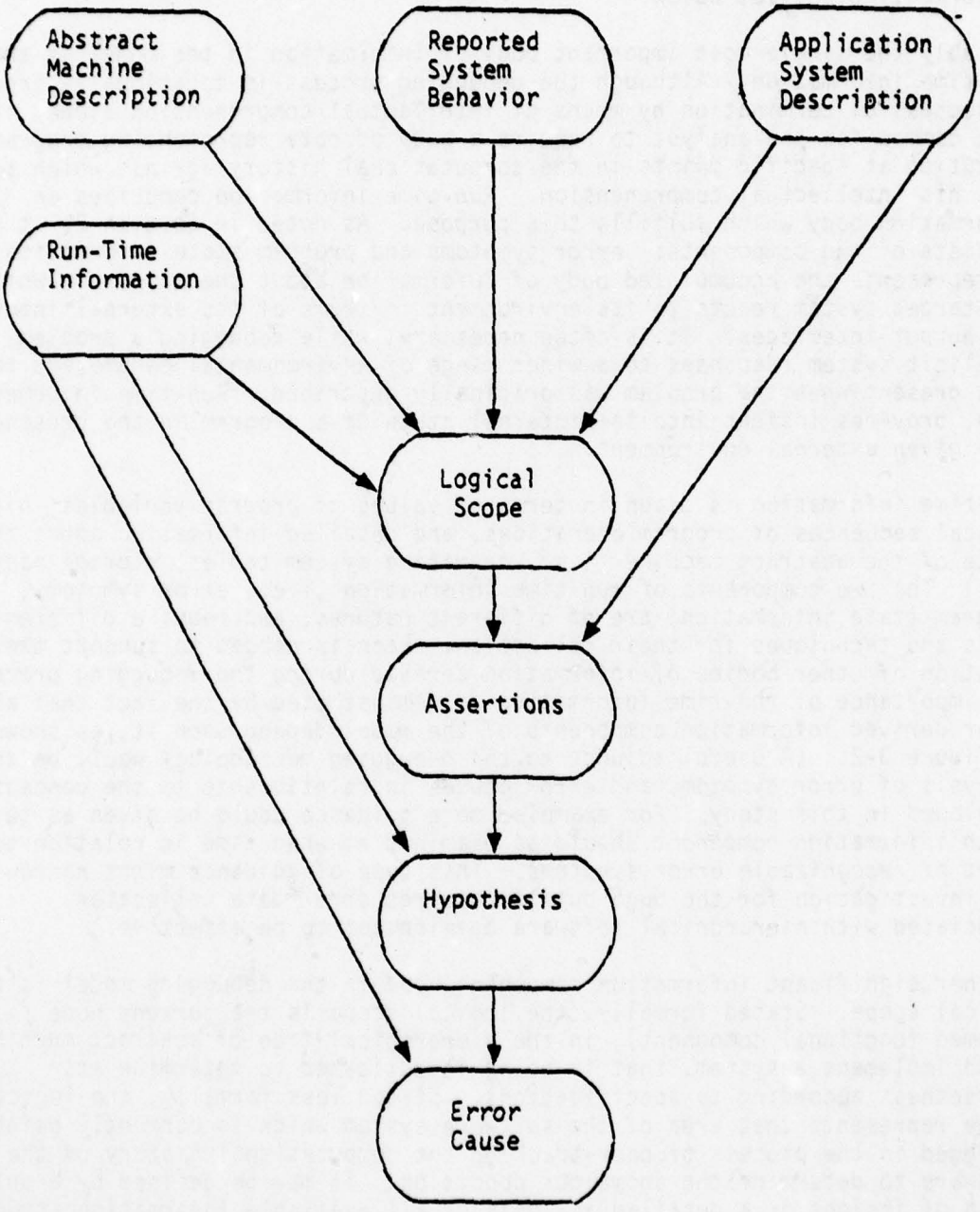



Figure 3-2. The Direct Dependent Relationship of Information Components of the Debugging Methodology

These derived information structures include run-time information, logical scope, assertions, hypothesis, and error cause. These information structures are briefly discussed below.

Probably the single most important body of information in the model is the run-time information. Although the debugging process is occasionally brought to successful termination by means of intellectual comprehension alone, it is more common for the analyst to require a body of data representing program execution at specific points in the computational history against which to test his intellectual comprehension. Run-time information comprises an information body which fulfills this purpose. As noted in Section 2, it consists of two components: error symptoms and program state information. It represents the accumulated body of information about the manner in which the target system reacts to its environment in terms of its external input and output interfaces. It is often necessary, while debugging a problem, to elicit system responses to a wider range of environmental parameters than were present when the problem was originally described. Run-time information, then, provides insight into the internal state of a program in the presence of a given external environment.

Run-time information is given in terms of values of program variables, historical sequences of program operations, and detailed information about the state of the abstract machine (i.e., operating system tables, storage maps, etc.) The two components of run-time information (i.e., error symptoms, program state information) are of different natures, and require different tools and techniques for their collection. Each is needed to support the creation of other bodies of information derived during the debugging process. The importance of run-time information is demonstrated by the fact that all other derived information components of the model depend upon it, as shown in Figure 3-2. (A useful adjunct to the debugging methodology would be an analysis of error symptoms and error causes in relationship to the concepts developed in this study. For example, more guidance could be given as to which information component should be examined at what time in relation to a set of recognizable error symptoms. This type of guidance might narrow the investigation for the bug, but it requires error data collection associated with hierarchical software development to be effective.)

Another significant information component used in the debugging model is the logical scope. Stated formally, the logical scope is the current node (i.e., a named functional component), in the hierarchical tree of abstract machines which implement a system, that is being investigated to determine its correctness according to specifications. Stated less formally, the logical scope represents that area of the software system which is currently being debugged in the process of back-tracking the computational history of the software to determine the anomalous condition. It may be defined by a quick flash of insight or a detailed analysis of all available information structures. In either case, the end result of an iteration of logical scope definitions is the isolation, or location, of the software error. Definition of logical scope depends upon the analyst's correct understanding of application system, its external interfaces with the abstract machine, the reported software performance and the resulting error. It also depends upon run-time information to provide insight into the manner of how a software design and source code implementation is flawed so as to produce results different than those required.

Two important concepts are associated with establishing the software boundaries represented by the logical scope. The first concept is that of definition of logical scope. This implies that the analyst is looking for the program component assigned to implement a specific functional area, as specified by a design, which demonstrated performance different from specified. Once the analyst has associated the earliest occurring software error with a functional component, the analyst refines the logical scope. This second concept, refinement of logical scope, implies that the analyst must examine the source code of the highest level program component and/or its interfaces with the abstract machine, or any program component in the hierarchy which assists it in performance of the functional capability until he has located the offending or missing construct(s).

The third derived information component is the set of assertions associated with a logical scope. On a global scale, these assertions represent the assumptions that lie behind the implementation of a design for a functional capability. On a more detailed level they are the values of a program's name space in a given point in time in response to a given set of input conditions. There are three types of assertions: (1) Entry assertions are those values of global data and input parameters which represent the assumptions a component makes about its environment when it is invoked. (2) Exit assertions are those values of data set by the module in the implementation of a given function at its termination point. (3) An intermediate assertion is an entry or exit assertion about the behavior of a component's sub-components.

The next information component, the hypothesis, is based on the assertions. The analyst formulates assertions in reference to a specific logical scope and tests their validity by comparison with run-time information. When the comparison of asserted values with actual values fails, the analyst has invalidated assertions for program values, provided by assertions, and actual program values, as indicated by run-time information. The hypothesis can attribute the discrepancy to any one or more of the existing information components: the application system description, the abstract machine description, the reported system performance, or the analyst's assertions. The creation of the hypothesis is an entirely intellectual process but, as modeled in this study, the hypothesis is based upon a discrepancy in the comparison of values asserted to hold true with information reflecting actual program performance.

The final derived information component is the error cause itself. This is implied by a hypothesis which has proven to be correct in light of the tests made on it by the analyst. It is the valid hypothesis, then which points to the resolution of the observed error. The manner in which the debugging methodology produces and uses the information components of the development process to isolate and resolve a software error is presented in the following section.

## 3.3 THE DEBUGGING PROCESS MODEL

This section presents the activities a debugging analyst must perform in isolating and resolving a software error. Many of these activities are intellectual activities which are highly individualistic in nature. The purpose of the debugging process model is to identify and suggest a sequence of debugging activities which may better organize the usual ad hoc approach taken in integration-level software debugging. Many debugging activities in the process model are actually not distinct steps since they are intellectual activities and may be performed almost intuitively and/or instantaneously. Further, the process model does not mean to suggest that the debugging methodology which it presents will necessarily solve all debugging problems. Lastly, the debugging process should be considered iterative in nature, some processes requiring computer usage and, perhaps, days in time, other process iterations occurring in seconds, or milliseconds.

The debugging methodology is presented by way of a directed graph depicted in Figure 3-3. A directed graph is a set of nodes (represented as named "places") and a set of edges connecting those nodes (represented as arrows connecting nodes). The nodes of the directed graph are the activities required in debugging; the edges represent the possible alternatives that exist in selecting debugging activities. The debugging model describes the function of each activity, its particular use of the information requirements, its relationship to other activities, and a generic list of tools which can be used to augment or produce information components. (These tools are described in more detail in Appendix A.) Also included in the description of debugging activities are the assumptions made for each activity and the probable sources for errors or problems.

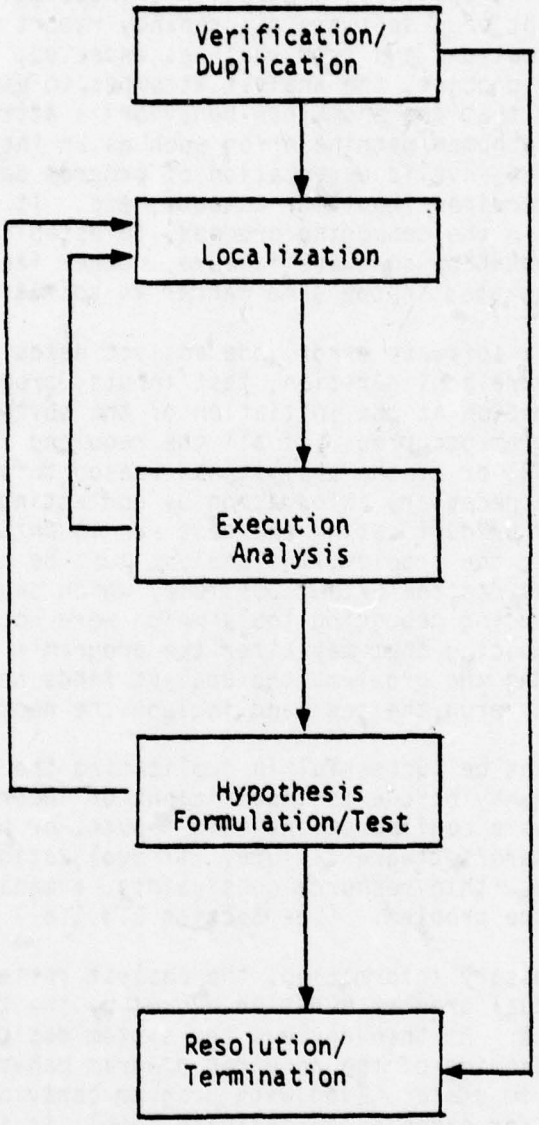



Figure 3-3. The Debugging Process Model

### 3.3.1 Verification/Duplication

Verification/duplication is the first process in the debugging methodology. It is initiated on receipt of a software discrepancy report (SDR), identifying a program behavior that differs from what was expected. During the verification/duplication process, the analyst attempts to establish that there is, indeed, an error and that the anomalous behavior is attributable to the software rather than to a human/machine error such as an incorrect hardware/software configuration, an invalid expectation of program behavior, an erroneous interpretation of required inputs or outputs, etc. It may not be possible, at this point in the debugging process, to establish conclusively that all problems are caused by software failure. Other factors, such as machine failures, are isolated in the same manner as software errors.

To verify that there is a software error, the analyst needs valid information about the hardware/software configuration, test inputs, program behavior, and program state information at the initiation of the software's operation and at the time the problem occurred. If all the required information is not supplied with the SDR, or if the analyst has reason to suspect the SDR's accuracy, he obtains the necessary information by contacting the person who reported the problem and by duplicating the test run in which the problem occurred. In duplicating the problem, the analyst must be careful to duplicate the exact conditions (to the extent possible) which caused the anomalous behavior, without introducing debugging tools which were not present in the original run since introducing them may alter the program's behavior. If, however, after duplicating the problem, the analyst finds he has inadequate information, he may then rerun the test and include the necessary tools.

The analyst may or may not be successful in duplicating the problem. Failure to duplicate the problem may be due to insufficient or incorrect information about the hardware/software configuration, test inputs, or program state, or to an intermittent hardware/software failure. If duplication of the anomalous behavior is not possible within resource constraints, a management decision is required to resolve the problem. (See Section 3.3.1.6.)

After obtaining the necessary information, the analyst reviews it to gain an understanding of the actual program behavior evoked by the test inputs and the initial program state. He then reviews the system design and the requirements to gain an understanding of the required program behavior for the given inputs and initial program state. Even with program behaviors that are obviously in error (as, for example, an infinite loop), it is wise for the analyst to determine the required program behavior as an aid to debugging the anomalous condition.

Depending upon the nature of the cause of the error and the error symptoms it generates, the analyst may verify, after comparing the actual and required program behavior, that a discrepancy actually exists. Its isolation and resolution may require an in-depth analysis or it may be an obvious logical flaw which is readily isolated and easily repaired. This methodology is not needed nor intended for the analysis and isolation of trivial software errors. It assumes that the majority of these types of errors have been removed during module-level testing. If error symptoms are recognizable and easily mapped to a faulty program component, many of the steps in the following processes can be omitted. Therefore, after verifying that a discrepancy

exists, the analyst proceeds to the next process in the debugging methodology, localization (See Section 3.3.2). Otherwise, he reports his findings and terminates the debugging process.

The steps in the verification/duplication process are depicted in Figure 3-4. The process is described in greater detail in the subsections that follow.

3.3.1.1 Process Steps

The steps in the verification/duplication process are:

- Duplication. This step is optional and can be omitted if sufficient valid information is supplied with the SDR. If additional information must be collected, the analyst does the following:

  - Reconstruct the operational status of the abstract machine by duplicating both the hardware/software configuration and the operational status of the external interfaces at the time the error was observed. The more complex the abstract machine's external interfaces, the more difficult it is to reconstruct the exact configuration.

  - Reconstruct the operational status of the application software by duplicating the exact software test configuration which exhibited the anomalous behavior.

  - Reconstruct the test environment by duplicating the test input state and initial conditions which evoked the reported error.

  If the analyst is unable to duplicate the problem within resource constraints, the problem is resolved by a management decision.

- Understanding. The analyst acquires an understanding of the actual program behavior by examining the information supplied by the SDR or collected as a result of duplication. The analyst acquires an understanding of the required program behavior for the given inputs and initial program state by examining the requirements and/or design specification. The analyst acquires an understanding of the abstract machine (i.e., hardware/software configuration) used by examining the supporting documentation. The level of understanding gained should be commensurate with the level needed to verify that a software error exists.

- Symptom Accumulation. If all of the information required by the debugging analyst to verify the existence of a software error is not supplied by the SDR, he may have to accumulate additional problem symptoms and run-time information. The analyst gathers error symptoms by examining the program's run-time external performance or behavior. He gathers program state information at the initialization of the run and after the occurrence of the error by the use of debugging tools that do not perturb the program's behavior.

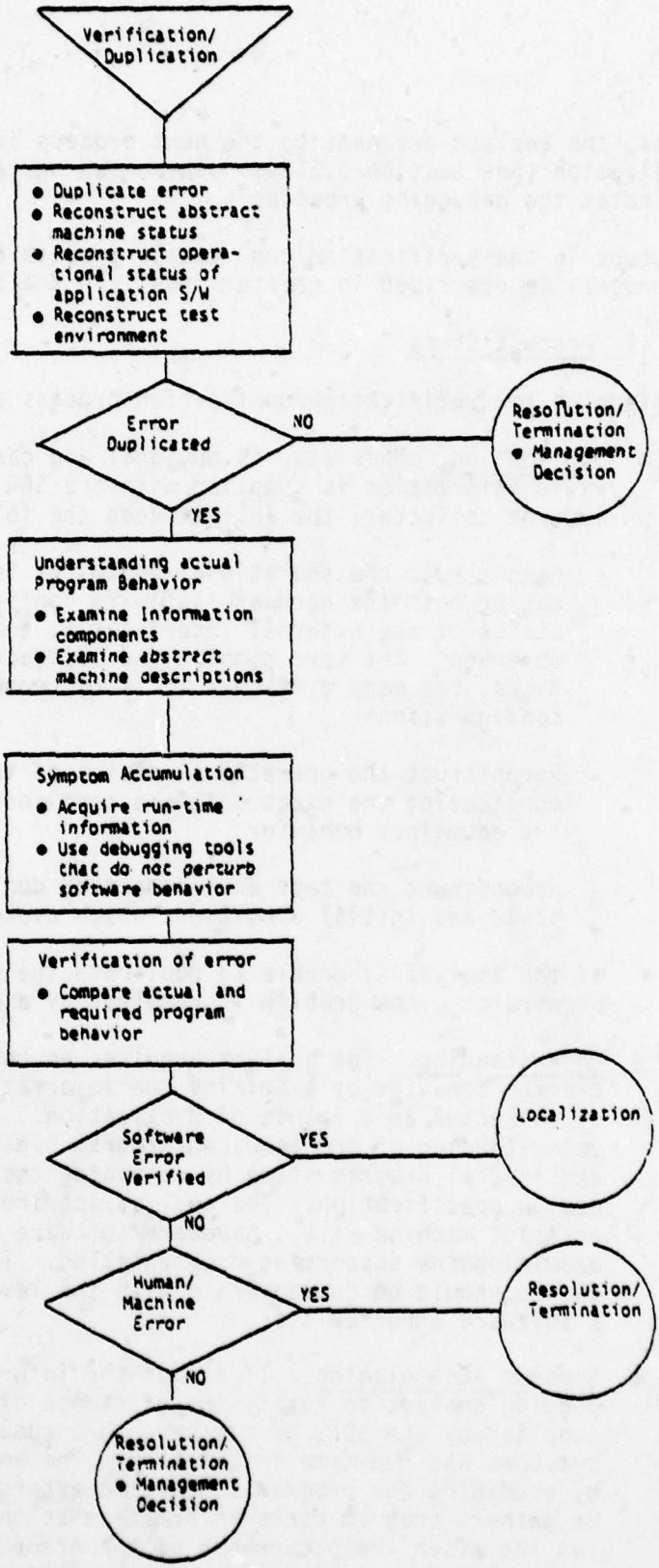



Figure 3-4. Verification/Duplication Process

- Verification. The analyst compares actual and required program behavior to determine if a software error exists. If the analyst finds there is, indeed, a software error, he proceeds to debug it. If he finds the problem was caused by a human/machine error, he reports his findings and terminates the debugging process. If he is unable to make a decision as to the existence or cause of an error due to ambiguities or errors in what constitutes required program behavior, he seeks a management decision to resolve the problem.

3.3.1.2 Process Relationships

The verification/duplication process relationships are shown in Figure 3-4 . This is the first debugging process and is initiated on receipt of a software discrepancy report, or equivalent. It has two successors: (1) the localization process for the definition of logical scope, entered when a software error has been verified to exist; and (2) the resolution/termination process, entered when it has been determined that actual results are not inconsistent with required results, or a management decision has been made to shelve the SDR (perhaps because of resource constraints).

3.3.1.3 Process Information Requirements

The following information components are used in the verification/duplication process:

- Error Symptoms. That information which directly relates the observed difference between actual and required software performance. It is often an error message generated by the application/operating system software. However, as noted in Section 2.3.1, it can also be infinite loops, abnormal termination, incorrect outputs, etc. It is used to verify the existence of an error in the computational history of the software under test.

- Program State Information. This information is available only if it accompanied the SDR or was obtained when the error was duplicated. It is used to verify and better define the software error, especially in terms of the difference between required and actual behavior, as represented by intermediate or final values for program variables. It may assist in identifying an inconsistency in the status of the hardware/software configuration, the status of simultaneous processing, and space and timing allocations which could account for or contribute to the error-causing condition.

- Application System Descriptions, specifically the requirements, design and source program specifications. That information relating to the functional area(s) of the application software which specify functional/performance requirements, design and implementation for the software component(s) specifically under test. This information should be contained on the SDR.* It is used to determine the required performance of the software.

* This strongly implies an organized testing methodology which is based on incremental integration testing. In addition, test cases used are clearly identified and based on testing component functionality and interfaces, if not also on internal control structures.

- Abstract Machine Descriptions. That information which describes the external interfaces of the application software with the software/hardware configuration components. It is used to understand the functional components of the abstract machine.

- Management Data. That information which is assumed to exist in all software development projects which have cost, performance and schedule contractual constraints. It is used in deciding on the action to take when this process has been unsuccessful in verifying and/or duplicating the software error.

3.3.1.4 Tool Usage

Verification of the software error may require duplicating the conditions which caused its occurrence. This sometimes can be difficult, if not nearly impossible, task. Generally, if the original testing procedures are methodically controlled, the debugging analyst will have a high degree of confidence in the original SDR and duplicating the software error is not necessary at this point. However, if it is necessary to duplicate the error, there are available, in a well-structured development environment, a number of tools for aiding the reconstruction of the exact test case, and hardware/software configuration including the data base and/or initial conditions. They include: *

- Program Support Library
- Test Input Listing Tool (See Appendix A, Section A.24)
- Script Tape (See Appendix A, Section A.24)
- System Status Summary Tool (See Appendix A, Sections A.26, A.27)
- Checkpoint/Restart (See Appendix A, Section A.28)

Verification of the software error may also require the gathering and analysis of run-time information by special debugging tools. These tools are used to obtain initial program state information and that information which is available after the occurrence of the error. Those tools are used in other processes of the debugging process model, but in the duplication/verification process, care should be taken to use tools which will not perturb the execution of the configuration of the system in response to a specific test case in such a way that the error will not be repeatable. Tools that operate during the program's operation, such as snapshot dumps, breakpoint/trap dumps, or trace program, should not be used. Tools that can be used include:**

- Post-mortem dumps (See Appendix A, Section A.1)
- Auxiliary dump (See Appendix A, Section A.6)

* Further discussion of these tools can be found in Section 5 and Appendix A, and Section 3.4.3 of Volume III.

** Further discussion of these tools can be found in Section and Appendix I, and Section 3.3.2 of Volume III.

3.3.1.5 Assumptions Made

This process, as all other processes in the model, assumes that the information requirements which are described for each process exist and are understandable to the debugging analyst. Obviously, if all of the information is not available, the analyst must verbally communicate with other development project personnel. Even when the project environment supports the development process in a well-structured manner so that documentation is available and accessible, complex debugging problems sometimes require interaction with hardware engineers and operating and support software system personnel.

Another assumption made by this and other tool using processes is that the analyst be able to effectively use the debugging support software to collect run-time information (i.e., error symptoms and program state information) and interpret it for debugging analyses.

3.3.1.6 Error Sources/Problems

A primary problem which can be encountered in this process is the failure to verify/duplicate the reported problem. The software discrepancy report may not accurately describe the software problem; the requirements specification may be amgibuous; the error may be manifesting itself intermittently; etc. When a software error is known to exist intermittently, a management decision regarding error resolution must be made based on cost, schedule, and performance contraints (see Section 2.5). This is necessary because duplication of the program state may require expenditure of a large amount of resources since it is sometimes complicated by intermittent hardware errors, CPU and I/O device timing variations, inability to replicate the exact software processing of the abstract machine, etc.

A problem may be difficult to reproduce if the SDR does not include sufficient information about the test, environment, and application system at the time of the error, or if the SDR is in error (i.e., factors which the test team believed to be involved may have been different than those actually involved). In theory, if a complete specification of the input space vector* is provided, sequential (as opposed to concurrent) programs exhibit perfectly reproducible behavior. In the real world, such complete specification of the environment is rare and much effort may be expended to reconstruct the real environment.

Systems which include concurrent processes may exhibit behavior which is not comprehensible in terms of the source specification of the functional processes. This can be due to several reasons. For example, if the operations of the abstract machine are not divisible when executed by the underlying machine, time dependent behavior can occur. Problems may arise due to small differences in the relative speed of concurrent processes which can alter program timing

* An input space vector is the totality of data values required to initiate the operation of the software in question. It includes all the data values set by external or interfacing hardware/software components.

to a sufficient degree to change program behavior. The basic problem involved is that the behavior of concurrent processes which share data generally depends on the order in which the process operations are performed. Unlike purely sequential programs, in which the order of execution of process operations is uniquely determined by input data, a concurrent program may exhibit non-reproducible behavior. It is difficult to duplicate a problem when such an error occurs, and it is necessary to recreate the sequence of events involved.

### 3.3.2 Localization

During the localization process, the analyst defines the logical scope, i.e., the boundaries within the software, in which he suspects the error occurred and on which he will concentrate his efforts. To define the logical scope, he first examines the error symptoms, as reported in the SDR or gathered during problem duplication (see Section 3.3.1), and any additional run-time information he may have generated, looking for an insight as to what caused the error. Then the analyst examines the design specification supported, perhaps, by cross reference information and requirements traceability matrices, to determine candidate logical scopes for where the error could have been caused. Depending on the type of error symptoms that exist, the analyst may find it necessary to run additional tests to gain further insight into the conditions under which the error manifests itself or to otherwise eliminate logical scopes until he has selected a single one. The selection of a logical scope may, at times, be a spontaneous act, driven by a quick insight. But, at other times, it will be a laborious task requiring a thorough investigation by the analyst.*

The localization process is iterative in nature as it is the mechanism used to isolate the boundaries of the specification which contains the error. The analyst first broadly defines the functional area in the software which is implicated by the difference between required and actual results as indicated by the test case, test procedure, and related design specification. Depending upon the size and complexity of the software under test, the analyst's familiarity with it and the associated error symptoms, the analyst may, at times, be able to easily define a logical scope within the source program representation which appears to be malfunctioning. At other times, the analyst must refer to the design specification related to the functional capability under test in order to formulate and test global assertions for a high-level logical scope which will then be iteratively refined as the scope of inquiry narrows. The intent for defining or refining the logical scope is to bound the focus of investigation during debugging according to a logical scheme which is based on examination of the computational history of the

* It is seen that a correlation of error causes with error symptoms would be a valuable mechanism by which to aid in formulating the logical scope. In addition, such a correlation might indicate that there is specific need to examine a particular system description representation instead of each of them. As noted earlier, the correlation of error symptoms/error causes, especially in relation to a hierarchically developed software system, has not been performed as a part of this study and such guidance is necessarily omitted.

software under test. Once a logical scope has been defined/refined, the analyst formulates and tests assertions regarding the logical scope in the execution analysis process (see Section 3.3.3). The purpose for the iterative definition/redefinition of logical scope is to isolate logical scope containing the error, (i.e., one in which no error existed on entry but did exist on exit). Eventually, the iterations result in identification of the program component, or construct within, which causes that component, or one in its hierarchy, to function in such a manner as to cause its ouput to be incorrect when processed as input by an interfacing component.

The localization process is similar to the process of hierarchical development. The analyst first looks at the specifications of the functional capability at a global level, then refines that scope until the implementation is found to be either correct or incorrect. If the implementation is found to be correct, the analyst must "debug" the design or requirements, a task which involves verification activities of another nature. In this process, as in all others, the analyst first assumes that the anomalous condition is an error in implementation, i.e., in the translation of the design into the source program representation. Only when the implementation is not found to be in error, does the analyst consider errors in the design or in the abstract machine.

There may be times when the analyst will not select a logical scope during localization. Numberous reasons exist for this decision. For example, it may be that the analyst realizes he made an error during the execution analysis process and decides to reenter it. Or he may be convinced, either on examination of additional information or intuitively, that the error resides in the selected logical scope and decides to reattempt the execution analysis process. There are also times when the analyst may select a previously examined logical scope, as when subsequent attempts seem to lead him around in a circle or when he has lost the trail of the error. Regardless, a methodical examination of the components in an hierarchy of programs and sub-programs based on the validity of entry and exit assertions should lead the analyst to a logical scope in which no error existed on entry but did exist upon exit, either by an error in implementation of the design or a deficiency of the design to meet the requirements.

The steps in the localization process are depicted in Figure 3-5. This process is described in greater detail in the subsections that follow.

3.3.2.1 Process Steps

There are two steps within localization process, both of which may need to be supported by an additional data collection step. These steps are described below.

- Defi. Logical Scope. This step is performed on initial entry to localization. It is also performed each time the execution analysis process (see Section 3.3.3) fails to confirm that the error resides in the previously selected logical scope. The analyst uses, as input to the definition task, the original error symptoms and all supporting run-time information. If the analyst has located a logical scope in which the error existed on entry, he uses the invalidated entry assertions, the error symptoms, and the computational history as additional inputs. Then he examines the design specification to

AD-A069 540 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DEBUGGING METHODOLOGY. VOLUME II. HANDBOOK FOR DEBUGGI--ETC(U)
APR 79 M FINFER, J FELLOWS, D CASEY F30602-77-C-0165
UNCLASSIFIED RADC-TR-79-57-VOL-2 NL

2 OF 2
AD A069540

END
DATE FILMED
7-79
DDC

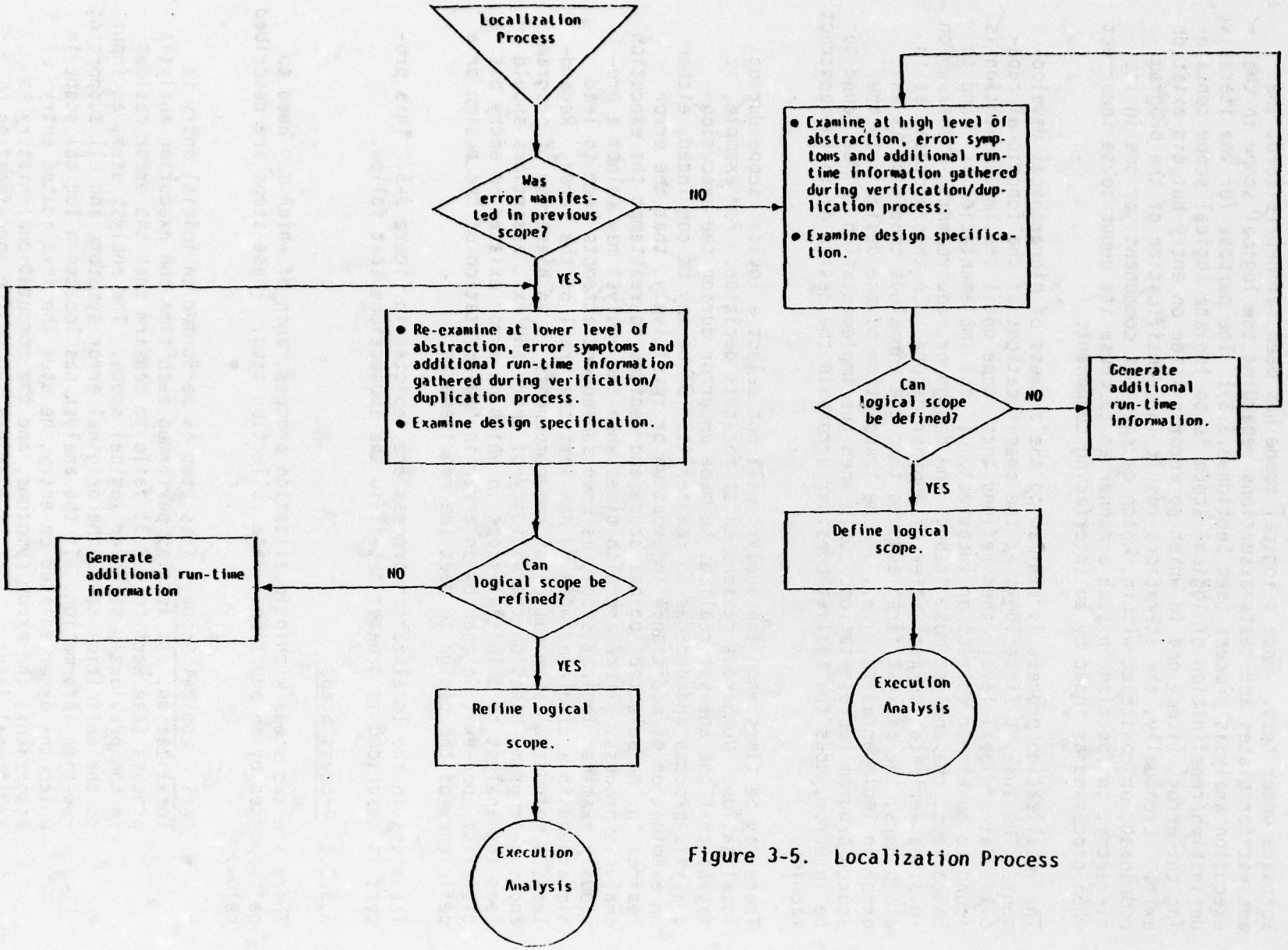



Figure 3-5. Localization Process

determine the functional area that may be responsible for the discrepancy between actual and required program behavior. In defining a logical scope, the analyst identifies the program components assigned to implement the selected functional area. This implies that he expects the error to reside in the highest level program component identified for a specific function and/or its interfaces with the abstract machine, or any program lower in the hierarchy which assists it in performing the specific function. Once the analyst has located a logical scope in which an error exists, he will try to refine (i.e., narrow) the logical scope on subsequent entries to localization until the error has been isolated. If, in the course of refining the logical scope, the analyst fails to select the one in which the error resides and reenters localization, he limits his scope definition attempts to those components included in the logical scope's hierarchy. At first, he selects as the new logical scope, subcomponents of the logical scope known to contain the error. If the analyst is unable to find any evidence of an error within the subcomponents, he then selects as the new logical scope, portions of the highest level component, paying particular attention to the interfaces with the subcomponents. If, after examining all possible logical scopes, the analyst is unable to locate any error (as evidenced by an invalidated assertion), the error is assumed to reside in the design and the analyst sets about to debug it. In doing so, he compares the design against the requiremenets, using the error symptoms and the collected run-time information as guides.

- Refine Logical Scope. This step is similar in nature to the stepwise refinement that occurs during the development of hierarchical software. It is performed each time the execution analysis process (see Section 3.3.3) confirms that the error resides in the previously selected logical scope but that logical scope is too broad to formulate a hypothesis as to the error's cause (see Section 3.3.4). Each time refinement is performed, the analyst narrows the logical scope to a program component and/or its subcomponents that are lower in the hierarchy than the highest level program component of the previously selected logical scope. To make the selection, the analyst uses the invalidated exit assertions and error symptoms for the previous logical scope as an additional input and examines the design specification to locate a lower level functional area that may be responsible for the error. The logical scope may eventually be narrowed to a single component, or a portion of one, which has no subcomponents. In this case, the analyst selects a portion of the component as the narrower logical scope. Iterative refinement continues until the logical scope has been narrowed sufficiently for the analyst to formulate a hypothesis about the error or until the selected logical scope is not the one in which the error resides. In the latter case, the analyst must define a new logical scope (see previous step) before continuing with refinement.

- Determine Computational History. The definition/redefinition of a logical scope must be based on the computational history, i.e., the actual operational sequence among the program components in response to the test inputs and initial conditions which caused the error to be manifested. Depending on how knowledgeable the analyst is with regards to the software under test and how successful he has been in defining the logical scope for investigation, he may not be able to determine the operational sequence by examining static hierarchy of program components described in the design and source language specifications. If this is the case, it will be necessary for him to collect dynamic flow information by using tools that selectively trace the program's execution.

3.2.2.2 Process Relationships

The localization process relationships are shown in Figure 3-5. This process is the second debugging process and it starts the iterative activities needed to isolate this software error. It is entered initially after verification/duplication to establish the initial boundaries on which to focus the analysis of the next processes. Subsequent entries from execution analysis and hypothesis formulation/test are made to refine the previous logical scope or define a different one. The localization process exits to execution analysis process each time a logical scope has been defined, or refined.

3.2.2.3 Information Requirements

The following information components are used in the localization process for defining the logical scope:

- Run-time Information, including:
  - Error Symptoms. That information which indicates an erroneous computational history in response to test inputs and initial conditions.
- Application System Descriptions, specifically the requirements, design and source program specifications. That information relating to the functional components of the software under test which manifested program performance which was different from that required.

3.2.2.4 Tool Usage

This process uses tools to help define or refine the logical scope. Definition of the initial logical scope can be aided by tools that relate error symptoms to functional program segments. For example, the Set-use Matrix/Cross Reference analysis tool (see Appendix A, Section A.10) can be used to identify all areas of the program where an error message is referenced or where anomolous data is set. This type of tool provides a static representation of the setting and using of all name-space values, which gives the analyst the first approximation of where erroneous processing can occur.

The analyst will also use tools which capture and display information relating to the operational sequence of program segments. While the entire computational history of the software can be recorded by traces, the analyst must be selective about such requests. The objective for the use of tools in this process is to support back-tracking analysis by displaying a selective area in the software during execution in order to determine how a given computation was influenced by previous computations. Tools which provide this information include:

- Snapshot Dumps (See Appendix A, Section A.2)
- Programmed-In Dumps (See Appendix A, Section A.4)
- Trace Routines (See Appendix A, Sections A.7, A.8, A.9, A.15)
- Program Flow Analyses (See Section 3.1.8.1, Volume III)

3.3.2.5 Assumptions Made

The definition and refinement of the logical scope as presented in this process assume a hierarchical software development which is well documented in the application system descriptions. A second assumption made is that requirements are directly traceable to both the design and the implementation of functional components

3.3.2.6 Error Sources/Problems

Given that the assumptions stated in Section 3.3.2.5 hold true, the principle source of problem in this process is determining the computational history of the software. The use of tools which provide dynamic control flow and dependency will alleviate some of this problem if they are available in the computing environment.

### 3.3.3 Execution Analysis

This process is entered from the localization process for the purpose of determining the correctness of the logical scope which localization produced. It is characterized as being a human thought process which is distinctly individualistic in nature. It is rudimentarily characterized in this methodology as the process of formulating and validating/invalidating assertions about the expected values of interfacing parameters and global data on entry to and exit from the selected logical scope. These assertions are made in light of the required performance in terms of a specific implementation. Depending upon where the analyst is in the back-tracking of the computational history and the nature of the error symptoms he is examining, the assertions being formulated may address the functionality of a high-level component or a particular control structure within a component.

Once a set of assertions has been formulated, the analyst compares actual program performance data with asserted values for program variables to ascertain if the program's operation is correct as he understands it. To do this, the analyst validates or invalidates the sets of assertions formulated. He does this by determining the actual values of the pertinent parameters and global data by examining existing run-time information or by generating any required additional run-time information. If the expected and actual values compare on entry to the logical scope (i.e., validated assertions) but fail to compare on exit (i.e., invalidated assertions), the analyst has located the logical scope in which the error resides and he proceeds to formulate a hypothesis that explains the cause of the error (see Section 3.3.4). If, however, the two sets of values do not compare on entry or compare on both entry and exit (i.e., the error occurred prior to entry or no error occurred), the analyst has failed to isolate the logical scope containing the error and reenters the localization process (see Section 3.3.2) to define a new one.

It should be recognized that the formulation of assertions is not an easy task. The resulting assertions may be too weak (i.e., too general) to detect a subtle error. They may be incomplete in that they do not address all the variables which affect or are affected by all the components in the logical scope. And they may be incorrect since assertion formulation tends to be an error prone activity.

The steps in the execution analysis process are depicted in Figure 3-6. This process is described in greater detail in the subsections that follow.

#### 3.3.3.1 Process Steps

The steps in the execution analysis are described below. Those steps which use run-time information may involve an additional step, symptom accumulation, for data collection.

- Formulate Entry Assertions. The analyst formulates assertions about the expected values of input parameters and interfacing global data on entry to the logical scope. These assertions, known as entry assertions, are generally derived from the design and source language specification descriptions of the highest level component in the logical scope. These assertions need only address the expected

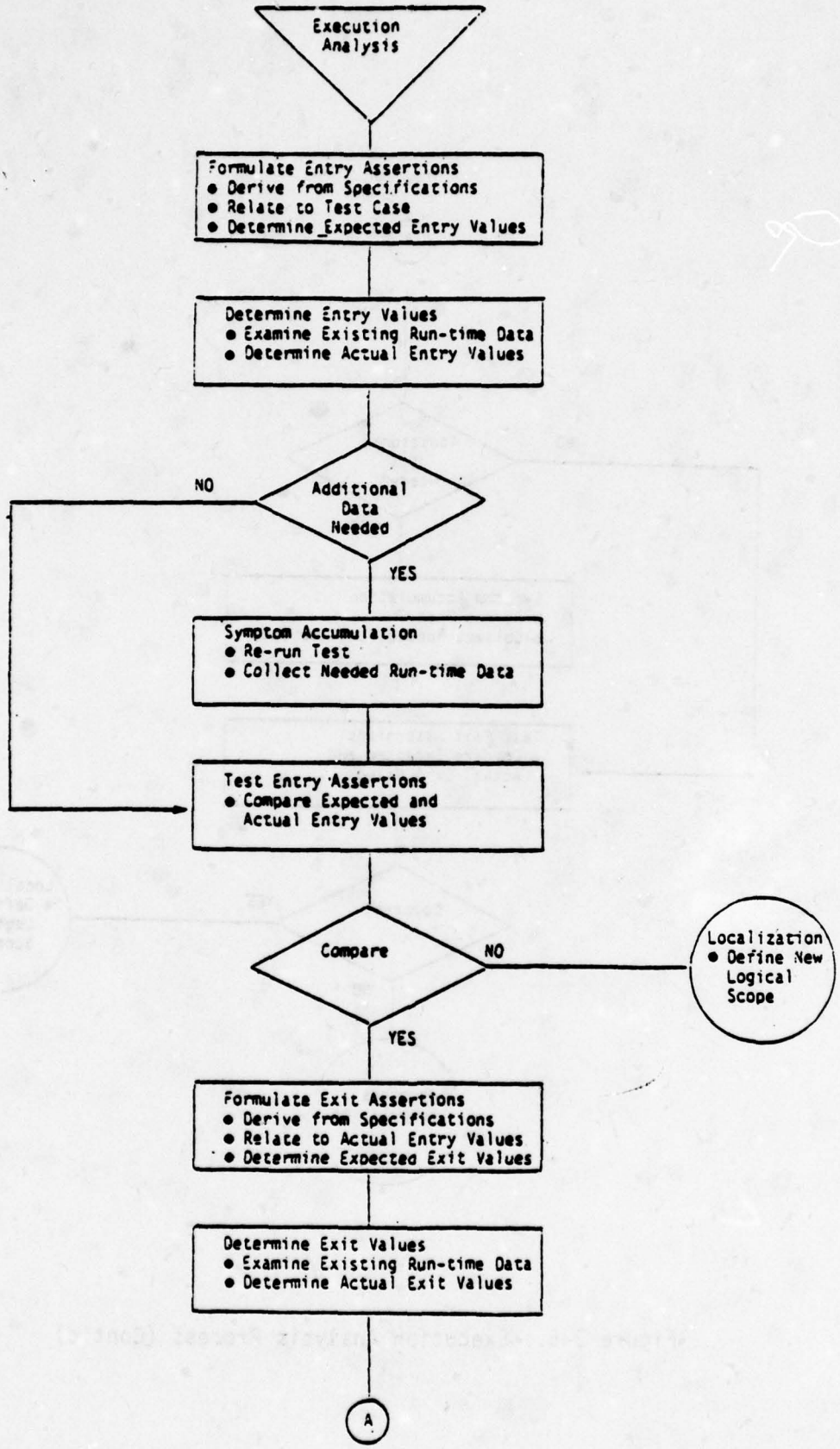



Figure 3-6. Execution Analysis Process

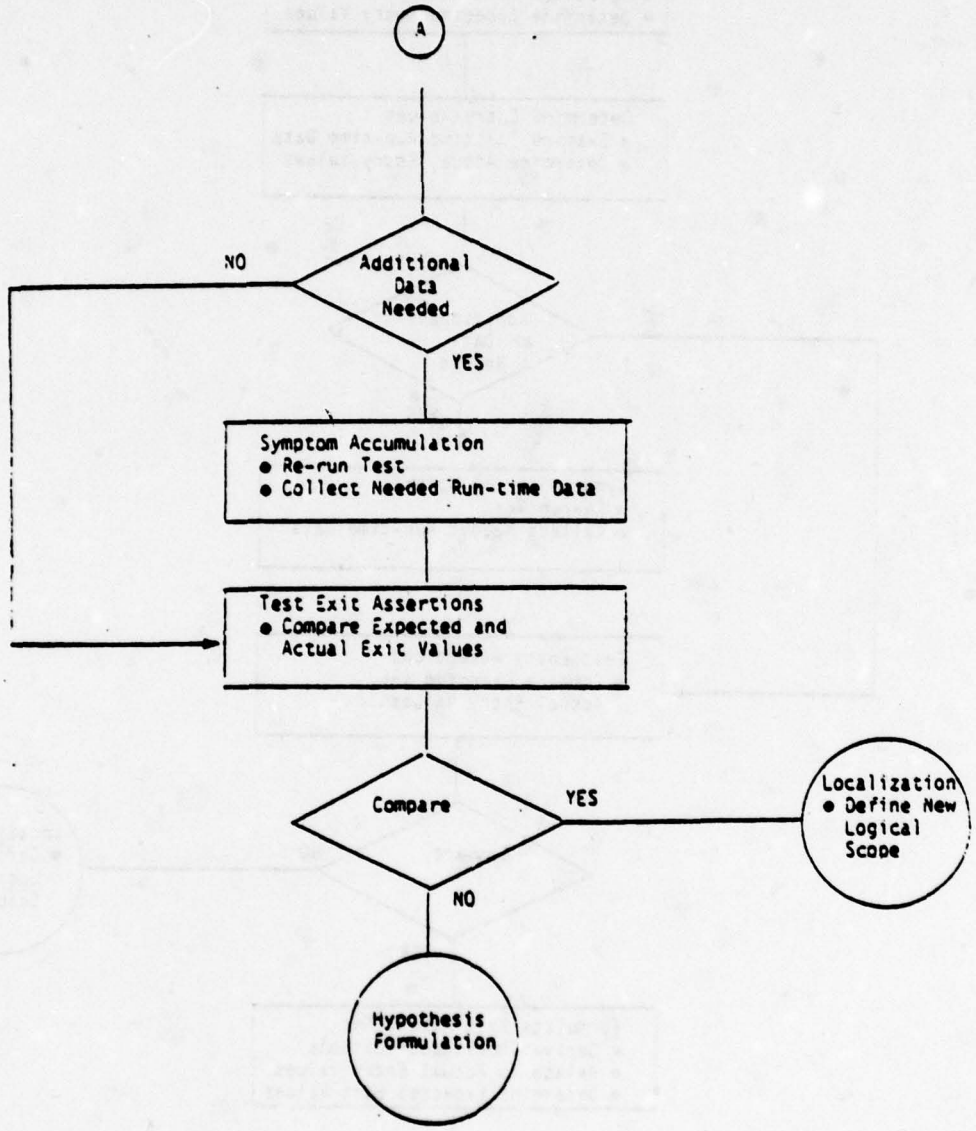



Figure 3-6. Execution Analysis Process (Cont'd)

program state in the presence of the input data defined by the current test case. The assertions are on a level of abstraction commensurate with the level of the component, i.e., for a high level component, the assertions may address the expected values for a data structure in its entirety while, for a low level component, they may address the expected values of individual variables.

- Determine Entry Values. The analyst next determines the actual values of the pertinent parameters and global data by examining existing run-time information. It may be necessary to rerun the test that exhibited the error and/or collect other run-time information (see symptom accumulation step below).

- Test Entry Assertions. The analyst compares the actual entry values with the expected values, as stipulated by the entry assertions. Because of the inherent difficulty in formulating adequate and valid assertions, the analyst would be wise to reexamine his assertions when he encounters a discrepancy between actual and expected values. If, after reexamination, the discrepancy still exists, the sought-after error occurred prior to entry into the selected logical scope and the analyst terminates the execution analysis process and re-enters localization (see Section 3.3.2) to define another logical scope.

- Formulate Exit Assertions. If the actual values agree with the expected entry values, the analyst formulates assertions about the output parameters and interfacing global data on exit from the logical scope. These assertions, known as exit assertions, are derived from the design and source program specifications of the highest level component and each component which it explicitly calls, and need only address expected program behavior in the context of the actual entry values. These assertions are also on a level of abstraction commensurate with the level of the component.

- Determine Exit Values As with the entry values, the analyst determines the actual values of the pertinent parameters and global data from existing or newly generated run-time information (see symptom accumulation step below).

- Test Exit Assertions. The analyst compares the actual exit values with the expected values, as stipulated by the exit assertions, re-examining the assertions in the light of the actual values. If the values fail to agree, an error occurred in the selected logical scope and the analyst enters the hypothesis formulation process (see Section 3.3.4) because the anomalous condition has been isolated. If, however, the exit assertions are validated by the run-time information, the error occurred in a different logical scope and the analyst reenters localization to define it.

- Symptom Accumulation. The analyst may need to re-run the test that exhibited the error in order to generate actual entry or exit data that can be compared with the expected values. The actual data consists of parameters and global data and is obtained by special debugging tools which collect and analyze program state information.

3.3.3.2 Process Relationships

The execution analysis process relationships are shown in Figure 3-6 . This process is entered from localization during the iterations required to isolate the program component containing the error. The successor to this process is either: (1) the localization process for the redefinition of the logical scope; or (2) the hypothesis formulation/test process entered when the logical scope is found to contain the error and the analyst attempts to formulate a hypothesis as to its cause.

3.3.3.3 Information Requirements

The following information components are used in the execution analysis process:

- Logical Scope. Used as the definition of boundaries in the software which is being analyzed to determine its correctness.
- Application System Description, specifically the design and program source specifications. Used to support the formulation of entry and exit assertions associated with the logical scope.
- Run-Time Information.
  - Error Symptoms. Used to help determine the variables about which assertions are formulated.
  - Program State Information. Used as a source of actual entry/exit values which are compared with the asserted (or expected) values.

3.3.3.4 Tool Usage

Tools are used in both the formulation of entry/exit assertions and in the generation of run-time entry/exit values. There are no tools that can help the analyst to determine the expected values for the entry/exit variables; he must determine those by inspecting the specifications. However, a set-use matrix/cross reference analysis tool (see Appendix A, Section A.10) can help the analyst determine the variables about which assertions need to be made. The analyst first determines which software components are contained within the selected logical scope. He then uses the set-use matrix to determine the entry and exit variables of the software components.

There are a number of debugging tools that gather run-time information and help the analyst determine the actual values of entry/exit variables in response to a given test case. These tools collect and analyze program state information. They include the following:

- Post-Mortem Dumps (see Appendix A, Section A.1)
- Snapshot Dumps (see Appendix A, Section A.2)
- Breakpoint/Trap Dumps (see Appendix A, Section A.3)
- Monitor Dumps (see Appendix A, Section A.5)
- Dynamic Internal Trace (see Appendix A, Section A.7)
- Recorded Trace (see Appendix A, Section A.8)
- Monitor Trace (see Appendix A, Section A.9)
- Software Breakpoint/Trap (see Appendix A, Section A.16)
- Hardware Breakpoints (see Appendix A, Section A.17)
- Reversible Execution/ Backtracking (see Appendix A, Section A.18)
- Program Execution-Oriented Recording/ Reduction (see Appendix A, Section A.22)
- Input/Output-Oriented Recording/Reduction (see Appendix A, Section A.23)

Sometimes it is not practicable to re-run an entire test case. The following tools can be used to input values for a given set of variables:

- Interactive Modification Tools (see Appendix A, Section A.19)
- Hardware Modification Tools (see Appendix A, Section A.21)

3.3.3.5 Assumptions Made

A primary assumption of this process is that a logical scope has been defined by the localization process. This implies that a software component (at either a high or lower level) has been selected as the one containing the error. A second assumption for this and all other processes is that the analyst has available and understands the application system descriptions needed in this process. High quality system descriptions are especially important if implementation type errors have been tentatively eliminated and design errors are under investigation. If an implementation error has been tentatively eliminated, the analyst will have to trace the allocation of functional requirements to the design specification. This allocation may not be clearly or consistently described in the design specification (and related documentation) or may have been misinterpreted by the designer. Even when project management ensures that documentation is available and accessible, the debugging analyst may have to interact with design personnel to clarify his understanding of their contents.

3.3.3.6 Error Sources/Problems

The formulation of assertions is one of the most difficult and error prone procedures in the debugging process. Deriving entry/exit assertions concerning the expected values of parameters and interfacing global data is a complicated mental process. The analyst analyzes the design and source language specifications and must determine expected program state values that would result from the input data defined by a specific test case in terms of time and space dimensions. The analyst, in effect, "simulate" the abstract machine to determine expected values resulting from program execution. Often he cannot simulate the performance of the abstract machine, and instead of determining exact values, he establishes an expected range of values. This range may be too wide or global, and may include values which are not correct. During the assertion testing processes, an incorrect actual value may be within the range asserted to hold true which results in the assertion being incorrectly validated. Conversely, an asserted range of values may be too narrow, and the assertion can be incorrectly invalidated. Obviously, a third problem can arise when the analyst cannot adequately establish entry/exit assertions.

In principle entry/exit assertions can be derived for a component without knowledge of its internal structure (e.g., source listing). Such assertions should be provided as part of a detailed design specification; however, this is not common in practice. Additionally, assertions provided as part of the design specification may be inadequate; i.e., they may be incorrect, overly weak, or not directly testable. As a result, a debugging analyst must often generate assertions from both his knowledge of the principles motivating the design and inspection of the source program specification for the component.

In the context of formal verification of program components, assertions are not dependent upon specific input data combinations; rather they are static statements of relationships between variables which hold for all inputs. As used in the debugging methodology, assertions may be input data dependent. While such assertions will be useful for determining exactly what went wrong for a specific test case, they generally will not provide insight into candidate solutions which prevent the introduction of new errors. The prevention of the introduction of new errors into a system while fixing old errors must be viewed as a currently unsolved problem which this study does not address.

3.3.4 Hypothesis* Formulation/Test

This process is entered after the analyst has confirmed that an error exists in the selected logical scope. In particular, the analyst has established, as a result of generating and testing assertions during the execution analysis process (see Section 3.3.3), that the entry assertions associated with the logical scope appear correct in light of the given test inputs but the exit assertions have been invalidated by run-time information. During the hypothesis formulation/test process, the analyst attempts to offer a plausible explanation for the invalidated assertions, i.e., the discrepancy between expected program

* According to Webster, a hypothesis is a tentative assumption made in order to draw out and test its logical or empirical consequences. As used here, it is a tentative explanation of the cause of an error.

values, as stipulated by assertions, and actual program values, as evidenced by run-time information. While it is hoped that this discrepancy is wholly responsible for the anomalous condition being debugged, it must be recognized that the condition may have been caused by other errors in addition to the located one, or that the located error is unrelated to the reported one.

The formulation of a hypothesis is a human thought process in which the analyst attempts to derive the specific cause of the error by examining previously accumulated, as well as specially-generated, run-time information and by examining the design and source language specification. The analyst looks for answers as to which operational conditions cause the error to manifest itself, where within the logical scope does the error occur, and, finally, what is the exact cause of the error. This implies, then, that the hypothesis points to the possible resolution of the problem. If the analyst finds that the logical scope is too broadly defined to allow him to formulate an adequate hypothesis, he reenters the localization process (see Section 3.3.2) to refine (i.e., narrow) the logical scope and thereby help isolate the error's location and cause. If the analyst is successful in formulating a hypothesis, he tests its validity by simulating, in some manner, the effect of the hypothesis' implementation. Methods that may be employed to test a hypothesis include mental modification of the source program, actual modification of the object program and its environment during or before execution by use of debugging tools, and simulation of the proposed change by use of simulation tools. If the test results indicate that the hypothesis was invalid (i.e., its implementation had no effect on the anomalous condition), the analyst returns to the localization process to refine the logical scope and thereby gain further insight into the error's location and cause. If the test results indicate the anomalous condition is only partially explained, the analyst also returns to localization to refine the logical scope or, if he is convinced there is more than one error, to define a new logical scope in search of another error. Even if the test results indicate the anomalous condition is completely explained, the analyst may not have sufficiently pinpointed the error to be able to recommend a solution and once again he returns to localization to refine the logical scope.

As can be seen from the above description, hypothesis formulation/test may be an iterative process performed at various levels of abstraction. For example, the analyst may first hypothesize that a particular subcomponent is returning an incorrect value to the calling component. If testing substantiates this hypothesis, the analyst refines both the logical scope and the subsequent hypothesis to explain more specifically where within the subcomponent the error occurs and what is its cause. Hypothesis formulation/test may also be an incremental process in which the analyst first attempts to explain part of an error and then, if he is successful, he uses the partial explanation as the basis for formulating a hypothesis that completely explains the error's cause. Only when the analyst feels confident that he fully understands where, when and why the error occurs and what must be done to correct it, does he enter the resolution/termination process (see Section 3.3.5).

Figure 3-7 depicts the steps in the hypothesis formulation/test process. This process is described in greater detail in the subsections that follow.

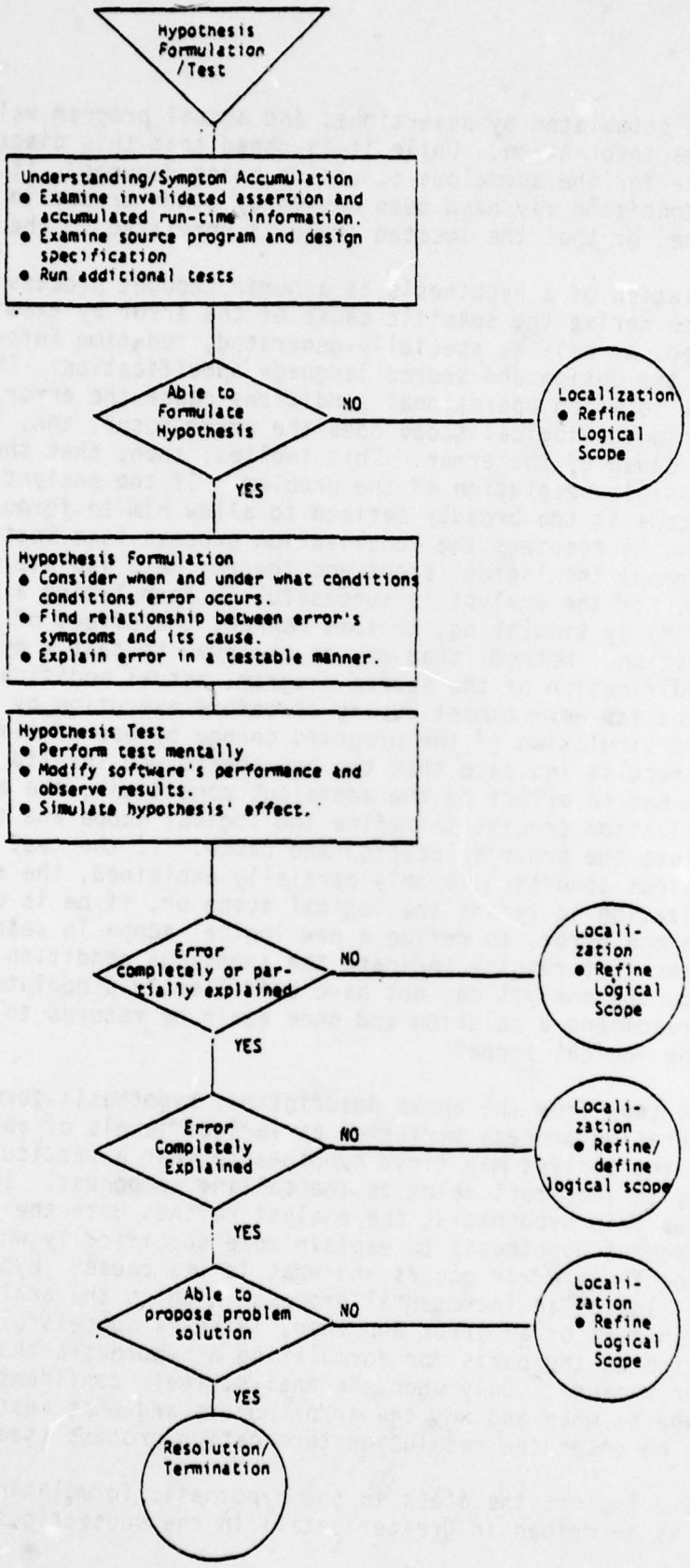



Figure 3-7. Hypothesis Formulation/Test Process

### 3.3.4.1 Process Steps

The steps in the hypothesis formulation/test process are described below.

- Understanding/Symptom Accumulation. This step is performed each time the hypothesis formulation/test process is entered. The analyst uses, as one input to this step, the invalidated assertion, i.e., the discrepancy between actual program values, as evidenced by run-time information, and the values asserted to hold true for the given test case and conditions. As an additional input, he uses the previously-accumulated run-time information. Then he examines the source program and design specification in relation to the discrepancy and run-time information to determine if he has sufficient understanding of the logical scope and software performance to attempt to formulate a hypothesis that explains the error's cause. If he is unable to do this because the logical scope is too broad, he returns to the localization process (see Section 3.3.2) to refine (i.e., narrow) the logical scope and thereby help isolate the error.

- Hypothesis Formulation. This step is initiated when the analyst feels he has sufficient understanding of the anomaly between the expected and actual software performance to attempt to formulate a hypothesis as to its cause. The analyst, in formulating the hypothesis, must consider where and under what conditions the error manifests itself, and must find the relationship between the error's symptoms and its cause. The process of formulating a hypothesis is an inductive exercise in which the analyst's reasoning goes from the particular error symptoms and supporting data to the general explanation of the cause. The hypothesis, itself, should be well formulated in that it is unambiguous, logical, testable, and indicates predictable results. It should explain the difference between required and actual program performance and thereby indicate a solution to the problem.

- Hypothesis Test. In this step the debugging analyst simulates, in some manner, the effect of the hypothesis' implementation to obtain evidence that demonstrates its validity. The method used to validate the hypothesis depends on the nature of the error and its cause, the manner in which it manifests itself, and the extent of the perturbance it creates. If the error is a simple and very localized implementation error, the analyst may be able to determine the effect of the hypothesis by mentally implementing the change in the source program specification and "stepping through" the resultant code. For more complex errors, the analyst will find it necessary to rerun the test that caused the error to manifest itself, to modify the software's performance during execution or prior to it by the use of appropriate debugging tools, and to collect run-time information, again with the use of tools, that indicates the effect of the modification. For even more complex errors, including design errors, the analyst may find it necessary to simulate the effect of the hypothesis to determine its validity.

After testing the hypothesis, the analyst may find that he has completely explained the anomalous condition being debugged, partially explained it, or completely failed to explain it. In the latter case, the analyst returns to the localization process (see Section 3.3.2) to refine the logical scope and thereby gain further insight into error. If the error is only partially explained, the analyst also returns to localization but his reason for doing so depends on the level of understanding he has achieved. If the analyst feels confident in both the correctness and completeness of the hypothesis and in his ability to recommend a solution based on it, then, most likely, more than one error has caused the anomalous condition. In this case, he reenters localization to define a new logical scope which contains another error. Otherwise, he reenters localization to refine the current logical scope so that he may gain sufficient understanding to formulate a more comprehensive hypothesis. Even if the hypothesis has completely explained the error, the analyst may lack sufficient understanding of it to recommend a solution. In this case, also, the analyst returns to localization to refine the logical scope. Only when the analyst feels confident that the hypothesis completely explains the error and that he is able to recommend a solution, does he enter the resolution/termination process (see Section 3.3.5).

3.3.4.2 Process Relationships

The relationship of the hypothesis formulation/test process to the other debugging processes is shown in Figure 3-7. This process is entered from execution analysis when the analyst has established that the entry assertions associated with the logical scope have been validated but the exit assertions have been invalidated. The successor to this process is either: (1) the localization process for the refinement or redefinition of the logical scope, or (2) the resolution/termination process when the analyst has been able to formulate a hypothesis that fully explains the error's cause and to recommend a solution for the anomalous program behavior.

3.3.4.3 Information Requirements

The following information components are used in the hypothesis formulation/test process:

- Logical Scope. Used in the definition of boundaries in the software within which the location of the error cause has been isolated.
- Assertions, specifically the validated entry assertions and invalidated exit assertions. The information verifies that the selected logical scope contains the error cause. The hypothesis is formulated in an attempt to explain the invalidated assertions.
- Application System Descriptions, specifically the requirements, design and source program specifications associated with the logical scope. This information includes the description of the specific test environment relating to the error under investigation. It is used to formulate and test the hypothesis.

- Run-Time Information, including:
  - Error Symptoms. That information which indicates an erroneous condition. The absence of error symptoms after hypothesis simulation helps to validate the hypothesis and assure that program modifications, used to test the hypothesis, will not cause new errors.
  - Program State Information. This information is available if testing the hypothesis requires running the program. It is compared with predicted results as part of testing the hypothesis.

3.3.4.4 Tool Usage

This process uses tools in each of its steps. In the understanding/symptom accumulation step, the analyst may need run-time information to determine under what conditions the error occurs. There are a number of debugging tools* that gather run-time information and help the analyst determine the program state information response to a given test case. They include the following:

- Post-Mortem Dumps (see Appendix A, Section A.1)
- Snapshot Dumps (see Appendix A, Section A.2)
- Breakpoint/Trap Dumps (see Appendix A, Section A.3)
- Monitor Dumps (see Appendix A, Section A.5)
- Dynamic Internal Trace (see Appendix A, Section A.7)
- Recorded Trace (see Appendix A, Section A.8)
- Monitor Trace (see Appendix A, Section A.9)
- Software Breakpoint/Trap (see Appendix A, Section A.16)
- Hardware Breakpoints (see Appendix A, Section A.17)
- Reversible Execution/ Backtracking (see Appendix A, Section A.18)
- Program Execution-Oriented Recording/ Reduction (see Appendix A, Section A.22)
- Input/Output-Oriented Recording/Reduction (see Appendix A, Section A.23)

* Further discussion of these tools can be found in Section 5 and Appendix A and Section 3.3.2 of Volume III.

In the hypothesis formulation step, the analyst can use a set-use matrix/cross reference analysis tool (see Appendix A, Section A.10) to help determine the variables about which the hypothesis must predict expected values. The analyst first determines which software components are contained within the selected logical scope. He then uses the set-use matrix to determine the entry and exit variables of the software components. The debugging analyst can use a number of tools in the hypothesis test step. If he needs to rerun the test case, he first must reconstruct the test environment (as he did in the verification/duplication process). There are a number of tools available for aiding the reconstruction of the exact test case, and hardware/software configuration including the data base and/or initial conditions. They include:*

- Program Support Library
- Test Input Listing Tool (see Appendix A, Section A.24)
- Script Tape (see Appendix A, Section A.24)
- System Status Summary Tool (see Appendix A, Sections A.26, A.27)
- Checkpoint/Restart (see Appendix A, Section A.28)

Tools that can be used to modify the program in order to test the hypothesis include the following:

- Interactive Modification Tools (see Appendix A, Section A.19)
- Recompilation/Correction (see Appendix A, Section A.20)
- Hardware Modification Tools (see Appendix A, Section A.21)

In this step, the analyst will need the same tools that were identified in the understanding/symptom accumulation step to collect and analyze run-time information that will be compared with predicted results. In addition, he may make use of comparator tools (see Appendix A, Section A.25) to compare program state information generated before and after a program modification was made to determine whether or not the modification caused new errors to occur. Simulators (see Volume III, Section 2.1.2) can be used to test a hypothesis that poses a design flaw as the cause of an error.

3.3.4.5 Assumptions Made

This process assumes that the localization and execution analysis processes have determined a logical scope for which entry assertions were validated and exit assertions were invalidated, and which therefore bounds the cause of the error. In addition, it is assumed that these assertions were correct and complete. If they were not, then the analyst has not, in fact, isolated the cause of error and a hypothesis can not be formulated.

* Further discussion of these tools can be found in Section 5 and Appendix A, and Section 3.4.3 of Volume III.

### 3.3.4.6 Error Sources/Problem

A primary problem in this process is that the logical scope has not been narrowed sufficiently to enable the debugging analyst to formulate a hypothesis that really explains the invalidated assertions which is assumed to be the error he is debugging. It may not be until a test of the hypothesis has taken place that the analyst will discover that he has attempted to formulate a hypothesis on the basis of insufficient information. Even if the hypothesis may theoretically explain the error cause, not all hypotheses are ammenable to complete validation by simulated implementation. For example, the hypothesis may be that a coded program component deviates vastly from its design. Inspection of the code in light of the design may validate the hypothesis but only the recoding of the component, which is beyond the scope of the debugging methodology (see Section 3.3.5), can validate that the error being debugged will be solved as a result. Similarly, the hypothesis may be that there is a discrepancy in the design in that the values output by one component are inconsistent with the values expected by a calling component. While inspection of the design of the two components may validate the hypothesis, it may be beyond the analyst's responsibility and/or capability to resolve which component's design is in error. In such cases, the analyst, after validating the hypothesis to the extent possible, enters resolution/termination (see Section 3.3.5) under the assumption that he has fully explained the error. Only after the error's solution has been implemented will it be possible to substantiate the validity of the hypothesis. If, at that time, it is found that the hypothesis does not fully explain the anomalous condition, it will be necessary to reinitiate the debugging process to complete the problem resolution.

One outcome of the hypothesis test process is that the analyst may find that he has explained an error but that it is unrelated to the error he is debugging. In this case, he reports the error in an SDR, enters the resolution/termination process to resolve the error he has explained, and restarts the debugging process in search of the cause of the original error.

## 3.3.5 Resolution/Termination

The resolution/termination process is the last activity addressed in the debugging model. When this process is entered, one of three conditions exists with respect to the error reported in the Software Discrepancy Report:

- Error has not been duplicated or has not been isolated within available/reasonable resources.
- Error has been attributed to a man/machine error.
- Error has been attributed to the software and a hypothesis has been formulated which explains the discrepancy between actual and required software performance.

The purpose of this process is to initiate the action, if any, which will resolve the reported problem.

If the analyst has been unable to duplicate or isolate the error after a reasonable expenditure of resources, as viewed in light of the seriousness of the error, a management decision is needed to resolve the problem. Intermittent software or hardware errors may be so difficult to duplicate or isolate that large amounts of resources are required. Often an error of this type is left open for some period of time in hopes that the error will not manifest itself again, or will be resolved at a later time when additional, and perhaps related, run-time information has been generated in the regular course of testing. It is to be noted that the difficulty of solving intermittent errors places greater demands on control and visibility into the testing/debugging process. Resolution of intermittent problems and, indeed all problems, is seen to be more efficiently achieved when each test is clearly identified, including an unambiguous statement of the cause-effect relationship of the test inputs and test results, and when the test process is structured so as to test increments in software functional capability.

If the analyst has determined that the observed and reported problem is due to an error in man/machine interfaces or a misinterpretation of required software performance rather than an error in the software specifications, he resolves the problem by identifying its source and offering evidence to substantiate his conclusion. If the error has been caused by inadequate/inaccurate documentation, including software specifications, test procedures, and user's instructions regarding input data preparation, initial conditions, limitations/restrictions, and hardware/software configuration, the analyst would be wise to recommend upgrading the appropriate documentation as part of the problem resolution.

If the analyst has determined that the reported problem is an error in one or more software specifications, he must document his findings, including a description of the error's cause, the conditions under which it occurs, and the proof that his hypothesis adequately expalins the difference between specified and actual software performance in terms of the given implementation. The analyst must also identify the actions that are necessary for problem resolution, as indicated by the hypothesis. The required action may entail modification to one or more of the hierarchy of software specifications (i.e., requirements, design, and source language specification); it may also entail retesting and regression testing as well as modifications to the test plan and/or procedures, user's manuals, and other supporting documentation. Whenever possible, the analyst should suggest a work-around procedure, i.e., a temporary fix which allows the tester to continue his activities, circumventing the anomalous condition in order not to impede development progress. It must be recognized, however, that such a procedure is only a temporary solution and that will be necessary for the software to conform to its functional/performance requirements as a condition for acceptance.

Neither the decision to implement the resolution indicated by the hypothesis, nor the actual implementation rests with the debugging analyst as modeled by this methodology. Unless the error is a trivial implementation error, additional analysis may be required to completely specify the solution. This is due to the fact that the analyst's hypothesis was based on an invalidated assertion derived from analysis of system performance in response to given test inputs and conditions rather than the total range of inputs

and conditions that must be handled by the software. Additional analysis may be required to ensure that the envisioned solution, as indicated by the hypothesis, will not introduce other errors. Additional analysis may also be required to evaluate the impact of the proposed solution on the cost, schedule, and performance constraints of the development effort and may require that alternative solutions be sought. None of these activities are debugging activities.

As can be seen from the above description, the resolution/termination process is a transition step between debugging and the other software development activities that proceded it. It was in those activities that the error was introduced, and it is the responsibility of thsoe activities to resolve it once it has been isolated by the debugging process.

Figure 3-8 depicts the steps in the resolution/termination process.

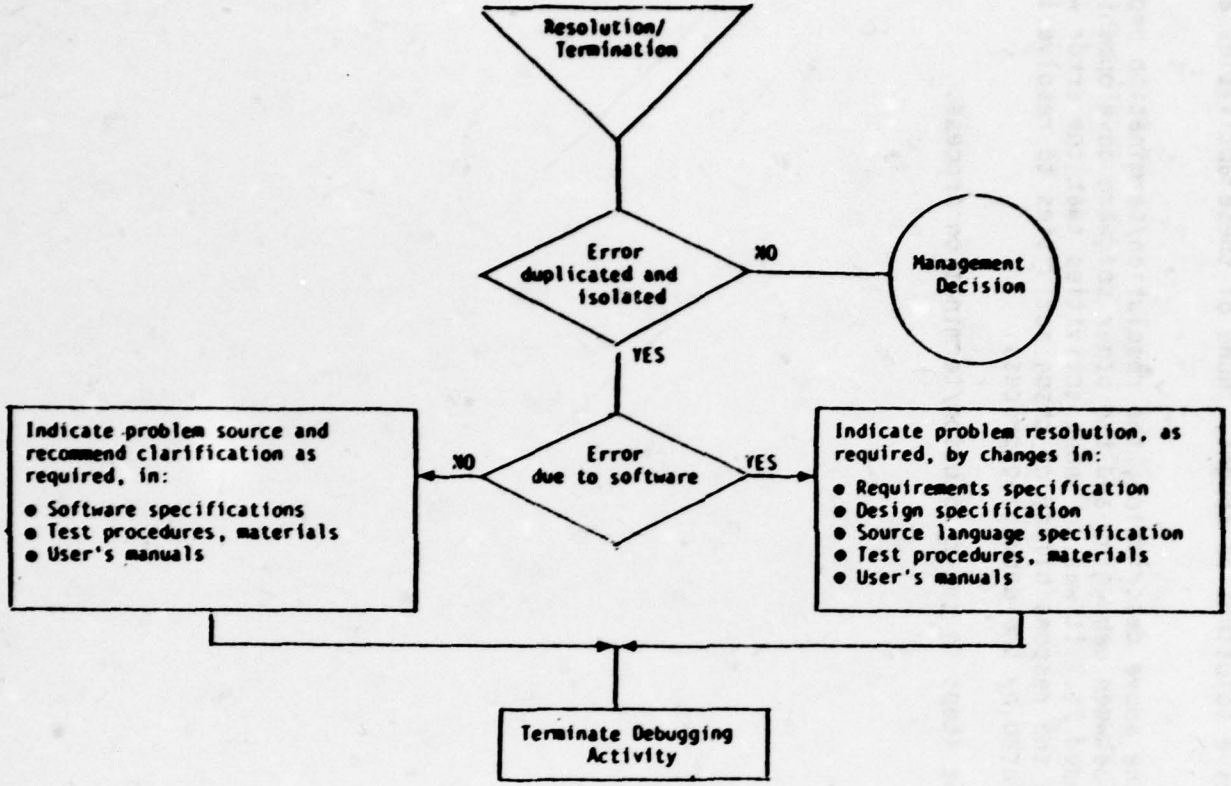



Figure 3-8. Resolution/Termination Process

## APPENDIX B: SOFTWARE DEVELOPMENT TOOLS AND TECHNIQUES SUPPORTING THE DEBUGGING PROCESS

The tools and techniques summarized in Table B-1 are used primarily for software development activities other than debugging. They are known to be of interest to RADC, and can be used to indirectly or directly support the debugging process either by themselves or as part of a software debugging methodology.

The information components included in the table are all those used in the debugging process and are amenable to support by such tools and techniques. The symbol "D" indicates that the tool/technique directly produces the desired information. The symbol "I" indicates that the tool/technique supports the acquisition of the desired information. A brief explanation of the tools/techniques follows.

### 1. CODE AUDITOR

A code auditor is a software tool used to enforce the following programming standards:

- Documentation standards - defining quantity and placement of commentary.
- Format standards - identifying physical placement and grouping of code elements on the source code listing.
- Design standards - limiting segment size and placing restrictions on the use of certain instructions with the end result of providing an optimization of code relative to execution time.
- Structural standards - requiring the use of rules for structured programming constructs, the top-down design and implementation of a system of programs, and the requirement that the components adhere to a hierarchical form.

### 2. PROGRAM FLOW ANALYZER

A program flow analyzer is a software tool which is applied to a source language coded program to perform the following functions:

- Source text structural analysis - identification and instrumentation of logical execution paths such that test case identification, module invocation and logical paths executed are recorded during test execution.
- Identification of program paths not executed.
- Provision of data to assist in the development of additional test cases appropriate to improvement of testing coverage.
- Analysis of data flow - monitoring the results of assignment and exchange statements.
- Automatic generation of computer program documentation.

Table B-1. Summary of Software Development Tools and Techniques Supporting Debugging

| Tools/ Techniques | Information Used in Debugging Process | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Information Derived Outside the Debugging Process | | | | | | Information Derived Inside the Debugging Process | | | |
| | Requirements Specification | | Design Specification | | Source Language Specification | | Logical Scope | Run-Time Information | Hypotheses | Assertions |
| | Spec. | Test Plan | Spec. | Test Procedures | Listing | Test Proc. | | | | |
| 1. Code Auditor | I | I | I | | | | | See Tables 2-1, 3-1, and 4-1 for tools that generate or analyze Run-Time Information (Program State Information) | This process is not supported by tools/ techniques; it is an intellectual process. | |
| 2. Program Flow Analyzer | D | | D | I | | I | I | | | I |
| 3. Program Design Language | | | I | | | | | | | I |
| 4. Top-Down Program Design and Implementation | | | I | | I | | | | | |
| 5. Top-Down System Design | | | I | | | | | | | |
| 6. Programming Support Library | | | | | D | | | | | |
| 7. Programmer Team | | | I | | I | | | | | |
| 8. Inspections | | | I | I | I | I | | | | |
| 9. URL/URA | D | I | I | I | | I | I | | | I |

3. PROGRAM DESIGN LANGUAGE (PDL)

A PDL is a specification or design language which provides for natural expression of procedural definitions (programs) at a level of completeness and detail appropriate to the designer's current knowledge of requirements.

4. TOP-DOWN PROGRAM DESIGN AND IMPLEMENTATION

This is a technique for design and implementation which produces a hierarchically designed and implemented program. The essential characteristics of a top-down hierarchical design is that the design procedes from the least detailed top level to the most detailed lower levels and each level of detail of the program is logically complete in itself..

5. TOP-DOWN SYSTEM DESIGN

This is the same as top-down program design, except applied to a software system, rather than to one individual program.

6. PROGRAMMING SUPPORT LIBRARY (PSL)

A PSL is a computer program and data repository established and maintained under control of a librarian in a central location. Facilities and procedures for the generation, storage and maintenance of all software developed can be implemented. The procedures established provide the following:

- The identification and delegation of responsibilities for clerical and record keeping functions associated with the programming process and the maintenance of the library.
- The delegation of responsibilities for all machine operations with regard to such items as project initiation/termination, program test philosophy, output media/frequency, etc.
- The procedures for recording, cataloging, and filing of all code generated on the project, both intermediate and final and for the retention of superceded corrected code for stated retention periods.
- A method for controlling the version(s) of the code contained in the library and the method for providing visibility into this process by configuration management personnel.
- A method for collecting and disseminating basic management data on the use of the library facilities and status of the programming activities.

7. PROGRAMMER TEAM

A team of programmers which includes, at a minimum, a chief programmer, a librarian and one other person who assists the chief programmer in the programming task.

8. DESIGN, CODE AND TEST INSPECTIONS

Formal methods for uncovering design, code and test errors by the inspection of the appropriate specifications by a team consisting of a moderator, a designer, a coder/implementer, and a tester.

9. URL/URA

A stand-alone tool to support requirements analysis and top-level design. It is discussed in Section 3.21.

# *MISSION*
# *of*
# *Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence ($C^3I$) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*