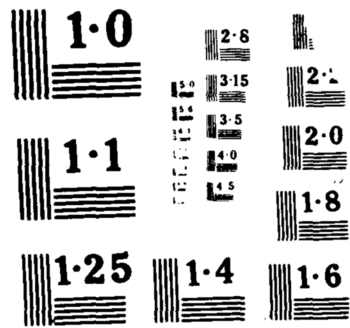
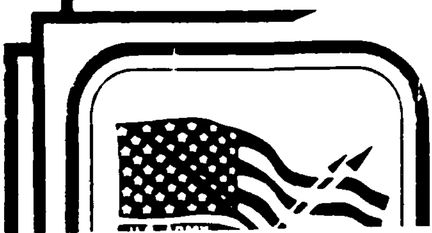
AD-A191 461 A GENERAL APPROACH FOR AUTOMATING SOURCE CODE VERIFICATION(U) ARMY STRATEGIC DEFENSE COMMAND HUNTSVILLE AL ADVANCED TECHNOLOGY DIRECTORATE 1/1

UNCLASSIFIED J H CORDLE 02 DEC 87 ATD-88-1 F/G 12/5 NL



DTIC FILE COPY

(2)



U. S. ARMY STRATEGIC DEFENSE COMMAND

AD-A191 461

P. O. BOX 1500
HUNTSVILLE, AL 35807-3801

SPECIAL REPORT ATD-88-1

A GENERAL APPROACH FOR AUTOMATING SOURCE CODE VERIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DTIC
ELECTE
APR 0 8 1988
S D
D

JAMES H. CORDLE
ADVANCED TECHNOLOGY DIRECTORATE
USA STRATEGIC DEFENSE COMMAND

DECEMBER 1987

"The views, opinions and/or findings contained in this report are those of the Author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other official documentation."

88 4 6 117

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT A Unclassified/Unlimited - For Public Release
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ATD-88-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION
Advanced Technology Directorate	CSSD-H-VP	

6c. ADDRESS *(City, State, and ZIP Code)* USASDC P.O. Box 1500 Huntsville, AL 35807-3801	7b. ADDRESS *(City, State, and ZIP Code)*

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
	X	

8c. ADDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

A General Approach for Automating Source Code Verification

12. PERSONAL AUTHOR(S)

Cordle, James Henry

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Summary-In House	FROM 3 Jun 87 TO 2 Dec 87	87 Dec 02	73

16. SUPPLEMENTARY NOTATION

X

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	08	
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

Flow Analysis, Dependence Graph, Source Code Verification, Unix, 'C' Language, Error Localization, Debugger

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Conventional debugger systems rely on user guidance to conduct error localization procedures System procedures are typically time consuming and inefficient due in part to the human factor. In addition, compatibility is limited among different hardware and software systems. These factors inhibit program design and development while increasing system cost.

A prototype software system has been developed that automates the debugging process. The system uses automation as an alternative technique for improving current debugger designs. Problem areas associated with typical debugger systems are presented. Concepts employing automation are supplied to remedy these deficiencies. Validation of these concepts are demonstrated by the prototype system. The prototype system exploits automation concepts through test case verification and provides the framework for an automated debugger system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL James H. Cordle	22b. TELEPHONE *(Include Area Code)* (205) 895-4392	22c. OFFICE SYMBOL CSSD-H-VP

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

SPECIAL REPORT ATD-88-1

A GENERAL APPROACH FOR AUTOMATING SOURCE CODE VERIFICATION

James H. Cordle
Advanced Technology Directorate
USA Strategic Defense Command

DECEMBER 1987

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

ABSTRACT

Conventional debugger systems rely on user guidance to conduct error localization procedures. System procedures are typically time consuming and inefficient due in part to the human factor. In addition, compatibility is limited among different hardware and software systems. These factors inhibit program design and development while increasing system cost.

A prototype software system has been developed that automates the debugging process. The system uses automation as an alternative technique for improving current debugger designs. Problem areas associated with typical debugger systems are presented. Concepts employing automation are supplied to remedy these deficiencies. Validation of these concepts are demonstrated by the prototype system. The prototype system exploits automation concepts through test case verification and provides the framework for an automated debugger system.

TABLE OF CONTENTS

		Page
List of Figures		vi
Chapter		
I.	INTRODUCTION	1
	A. Program Error Localization	1
	B. The Research Problem	2
II.	BACKGROUND	4
	A. Control Flow Analysis	4
	B. Data Flow Analysis	7
	C. Summary of Flow Analysis	10
	D. Dependence Graph Construction	11
III.	ERROR LOCALIZATION	15
	A. System Design Issues	15
	B. Enhanced Methods for Traditional Designs	22
	C. Summary	25
IV.	SYSTEM DEVELOPMENT AND RESULTS	27
	A. Prototype System Development	27
	B. Initialization Phase	30
	C. Error Localization Processing Phase	35
	D. Termination Phase	42
	E. Prototype System Verification	44
V.	CONCLUSIONS AND RECOMMENDATIONS	48
	A. Conclusions	48
	B. Recommendations	50
Appendix A.	TEST CASES	51
Appendix B.	PROTOTYPE SYSTEM SOURCE CODE	54
Bibliography	....	65

LIST OF FIGURES

Page

Figure 1. Control Flow Analysis 6

Figure 2. Data Flow Analysis 9

Figure 3. Dependence Graph Representation 13

Figure 4. Program Design Cycle 16

Figure 5. Error Localization Cycle 31

Figure 6. Initialization Phase 32

Figure 7. Error Localization Processing Phase 36

Figure 8. Termination Phase 43

Chapter I

INTRODUCTION

Recent software system designs have demonstrated the need for superior development techniques. With computer software expenses constituting about 90% of total system cost [10], alternative methods are needed to provide more efficient approaches to software development. One such method is automation. Fully automated support of software development can result in a 20-25% reduction in cost [10].

A. Program Error Localization

Program debugging is the most tedious task associated with software development. Conventional systems require user interaction at all levels due to the lack of automation in the error localization process. A source code program is typically prepared before execution in an attempt to provide the user with information necessary for localizing errors. It is the user's responsibility to apply this information correctly. In addition, the user is responsible for program maintenance, conducting debugger operations, and a host of other menial tasks. These menial tasks are tedious and may introduce additional errors not associated

with the debugging problem. These errors are normally caused by user fatigue and can go undetected throughout the debugging process. Incorrect application of the information provided can result in time spent in areas where problems are non-existent. These errors can result in a tremendous amount of wasted time and increased cost.

B. The Research Problem

Automation is a method of improving current debugger designs. Automating tasks can reduce the number of accidental errors introduced into a debugging session and reduce user frustration. In addition, automation reduces cost and time factors while increasing system performance.

This research focuses on providing an automated debugger system. A prototype system will be developed to prove concepts associated with this research. Menial tasks associated with typical debug sessions will be automated. Of fundamental importance will be the control of debugger operations by the prototype system. Using a data base constructed before program execution, the prototype automatically isolates program errors. This feature will reduce user interaction to a minimum. The prototype system will be exercised using test programs to provide a visual inspection of debugger capabilities.

The objective of this research is two-fold: to provide

a 'smart' debugger through automation and to minimize the need for user intervention. Both factors support the idea of a universal debugger concept. The following research satisfies these objectives by demonstrating a possible prototype system.

Chapter II discusses source code preparation necessary to provide the facilities for an automated debugger system. Problems with conventional debugger systems and techniques for improving these problems through automation are presented in Chapter III. The prototype system that demonstrates these concepts is described in Chapter IV. Chapter V provides concluding remarks and suggested areas for further research.

Chapter II

BACKGROUND

The effectiveness of an automated system commences with program data base design. An algorithm scans the source code acquiring pertinent details and places them in a program data base for future use. Additional algorithms extract information from this program data base to make decisions regarding automated system guidance. Increasing automated system capabilities requires a more complex algorithm and data base design.

Program data base design begins with a flow analysis of the source code program. Flow analysis consists of two distinct parts: control flow analysis and data flow analysis. The resultant structure of flow analysis is a dependence graph forming a program data base to be manipulated by the automated system.

A. Control Flow Analysis

Control Flow Analysis (CFA) depicts the potential, possible flow of control from one statement to another in a program. This flow of control is typically represented by a

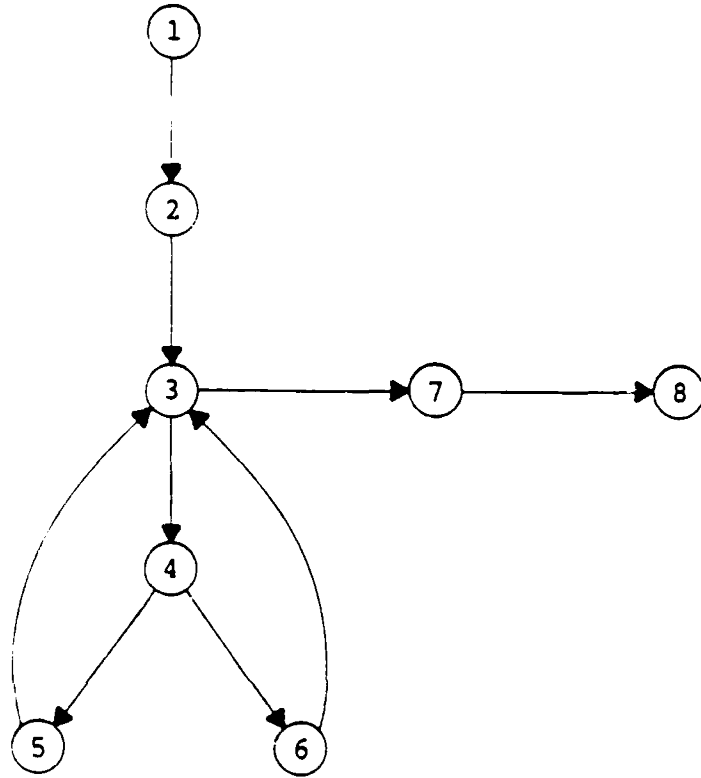
directed graph termed the 'control flow graph' or simply 'flow graph'. A flow graph must have the following two properties:

(1) A single entry node at which to begin, but may have more than one exit node.

(2) A path must exist from the entry node to every node in the flow graph.

Formally a flow graph is represented by the triple G = (N, E, S0), where (N, E) is a directed graph and N and E are node and edge sets respectively. The entry node S0 is an element of N to which there is a path to every node of the flow graph. In general, the nodes of a flow graph, N = {S0,S1,S2,...,Sn}, represent program statements and a set of edges, E, represent control paths from one statement to another [4]. The successors of node Sm are nodes Sn for which there exists a path (Sm,...,Sn). The predecessors of node Sn are all nodes Sm for which there exists a path from Sm to Sn [1]. Figure 1 depicts a pictorial representation of a typical control flow graph and corresponding information using test case two found in Appendix A.2.

The purpose of CFA is to encode the flow of control of a program for use in the data flow analysis that follows [6]. The flow graph provides the data flow analysis procedures with an appropriate structure. This structure guides the analysis from one program unit to another in a specified order [4]. Several methods exist for flow of control transformations applied to the control flow structure




```
S0 = 1

N  = { 1,2,3,4,5,6,7,8 }

E  = { (1,2),(2,3),(3,4),(3,7),(4,5),(4,6),
       (5,3),(6,3),(7,8) }
```

Figure 1: Control Flow Analysis

during CFA to simplify the ensuing data flow analysis. Node splitting and graph transformation are two such techniques available [6]. No more time than necessary will be spent in CFA except to provide a useful flow graph for data flow analysis.

B. Data Flow Analysis

Data Flow Analysis (DFA) is the pre-execution process of ascertaining and collecting information about the use of quantities in a computer program. Variables are normally selected as the quantities under observation, because they provide fundamental information from which other information can be determined [6]. In DFA, the flow graph, produced in CFA, is scanned systematically to obtain information about the use of variables. Certain inferences can be made about the use of these variables at other points of the program. A DFA must in effect examine all possible execution sequences starting at the entry node to see if the variable under consideration is ever used again in successor nodes (statements). Algorithms exist to accomplish this task [4] and the prototype system will not consider these any further. Information discovered during data flow analysis is placed into a suitable structure for future use.

The conventional method of locating anomalies in the data flow consist of three steps as stated by Fosdick and Osterweil [4]:

(1) Determine whether an anomaly exist.

(2) Find a path containing the specific anomaly.

(3) Attempt to determine whether the path is executable.

Step 3 above can be done through techniques such as symbolic execution; however, such techniques are costly and should be avoided if possible [4].

In these terms, DFA seems to be better suited for tasks such as code optimization or syntax analysis for compiling source code. By extending these three steps, a systematic approach can be established for a prototype debugger system. The function of this system is to automate tasks normally performed manually by the user. The prototype system envisions three corresponding steps for error localization:

(1) Determine whether an anomaly exists by querying the user after program execution followed by additional executions if necessary.

(2) Find a path containing the anomaly.

(3) Localize the error by suggesting the node or statement that could be the possible or suspect source of the anomaly.

Extending the normal definitions associated with DFA, additional information can be acquired to support complex debugger facilities. These 3 steps continue to require a typical DFA before program execution and infer usage of this information during program execution. Figure 2 demonstrates

statement	source code
1	`num_of_chars = 0`
2	`num_of_lines = 1  /* error */`
3	`while..........`
4	`  if..........`
5	`    ++num_of_chars`
6	`  else.......++num_of_lines`
7	`printf.........num_of_chars`
8	`printf.........num_of_lines`

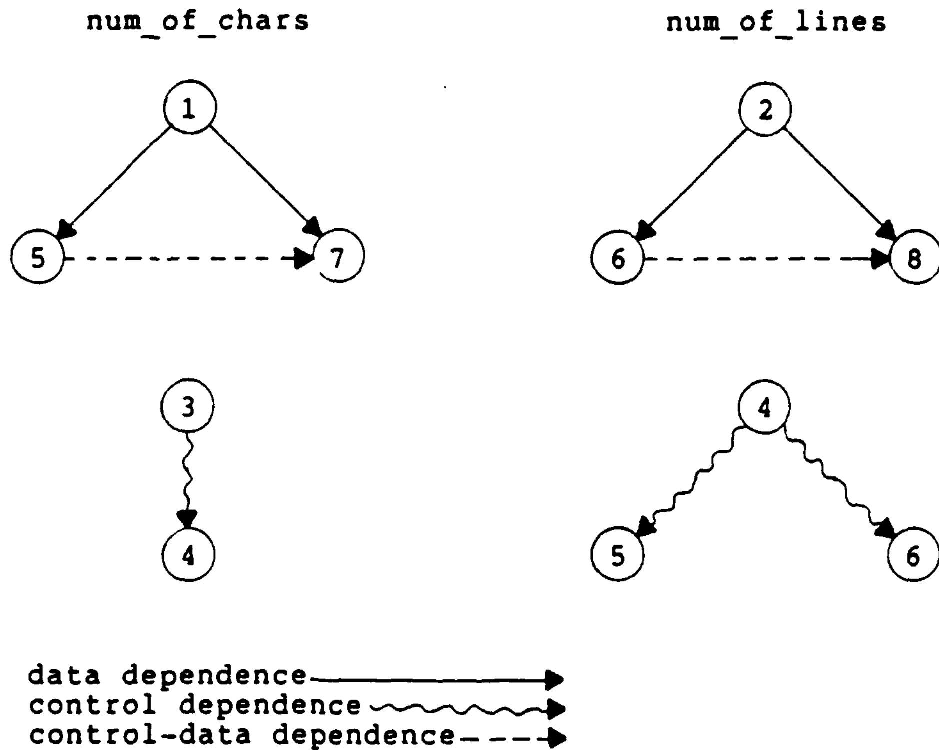


Figure 2: Data Flow Analysis

DFA using source code of test case two located in Appendix A.2. This DFA was performed using definitions proposed by Korel [7]. These definitions can extend typical data flow analysis to support the debugger facilities suggested. These definitions will be discussed further in section D.

C. Summary of Flow Analysis

In summary, DFA provides a search over the nodes of a flow graph created by CFA to determine certain relationships between the uses of data in various statements. Thus, before DFA can begin, a correspondence between statements of a program and the nodes of a flow graph, or CFA, must be established. DFA will want to associate nodes in a flow graph with sets of variables that have been either referenced or defined. Once this association has been made, flow analysis will be complete.

Specifically, the prototype system is not concerned with typical methods [1,4,6] used to detect anomalies but with detection of underlying errors requiring interactive input from the programmer. CFA and DFA occur before program execution. The prototype system assumes a suitable CFA and DFA has been performed on the program before execution and that the program source is ready for analysis.

D. Dependence Graph Construction

The end product of flow analysis results in the establishment of a structure with information about the relationship of program statements and variables. Many such structures exist, but of particular interest is the dependence graph incorporated by Korel [7]. The dependence graph resolves the relationship between program statements and variables through three types of dependences or influences. As defined by Korel [7], these entail:

(1) Data.

(2) Control.

(3) Control-data.

Data dependence occurs when one statement assigns a value to a variable of data and another statement uses the value of the data variable. The data variable must not be redefined between the points of assignment and use or the use will become dependent on the new definition of the data variable.

Control dependence is defined between a statement containing a test condition, and the succeeding statements that may be chosen to execute based on results of the test condition.

Control-data dependence is a combination of data and control dependences. Control-data dependence is the result of a data variable changing because a different control path

was chosen. A control path is selected from the results of a test condition. A data variable is changed within the control path. On leaving this control path, the data variable is referenced. The value of the data variable is dependent on modifications made to it while inside the control path. The control path affected the value of the data variable resulting in a new data dependence, thus control-data dependence [7].

The result of CFA and DFA is a data base structure (dependence graph) containing pertinent details of the source code program. DFA incorporates dependence definitions described by Korel [7] and scans the CFA structure resulting in a dependence graph of the source code program. This dependence graph forms the basis from which all operations of the prototype system revolve. Using the CFA of Figure 1 and the above definitions to provide the DFA of Figure 2, an appropriate dependence graph of the source code program is shown in Figure 3.

The dependence graph of Figure 3 is shown as an adjacency list. Each node contains information identifying the relationship between program variables and statements. This information assists the prototype system in making decisions about debugger operations. The prototype system obtains this information by backtracking through the adjacency list. The data base can be represented with other structures and optional traversal techniques exist. These methods were

```
n1 --> null

n2 --> null

n3 --> null

n4 --> n3(c) --> null

n5 --> n4(c) --> n1(d:num_of_chars) --> null

n6 --> n4(c) --> n2(d:num_of_lines) --> null

n7 --> n5(cd:num_of_chars) -->
       n1(d:num_of_chars) --> null

n8 --> n6(cd:num_of_lines) -->
       n2(d:num_of_lines) --> null
```

```
n# == source code statement or node

d  == data dependence

c  == control dependence

cd == control-data dependence
```

Figure 3: Dependence Graph Representation

chosen merely to demonstrate the data base design and the prototype system. However, these methods seem to be natural choices for the proposed system.

Chapter III

ERROR LOCALIZATION

Traditional debugger systems typically possess numerous attributes making them complex and difficult. These attributes impose additional hardships on the operator. Current trends are toward a more automated debugging environment. Emphasis is placed on relieving operator responsibility and automating many menial processes. In addition, current technology provides resources to enhance these systems beyond traditional approaches.

A. System Design Issues

A representative program design cycle is shown in Figure 4. The following list comprises issues associated with conventional debugger systems. These entail:

(1) System operational concept and interface language.

(2) Information management.

(3) Information structural arrangement.

(4) Adaptability.

(5) Machine independence and flexibility.

(6) Error isolation techniques.

(7) Data examination.

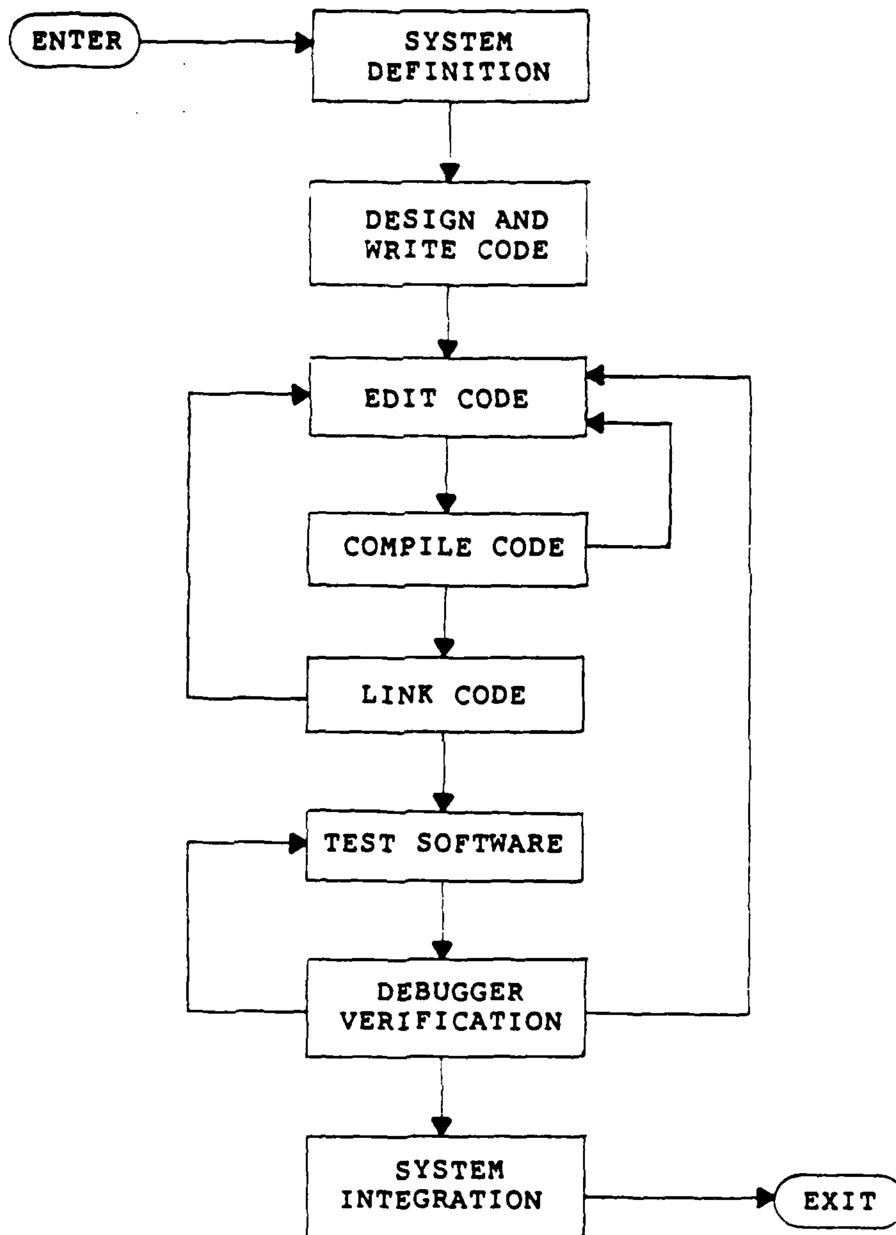


Figure 4: Program Design Cycle

Users must be familiar with the following breakpoint debugging concepts: a working knowledge of system operation and the interface language, identification of the problem area, and breakpoint placement.

Interface languages are typically complex and cryptic in nature with a minimum amount of suitable documentation available. A thorough knowledge of the debugger command language is required to efficiently determine information necessary for error isolation. Typically, errors must be corrected on localization within the system debuggers host operating system. Therefore, several editors may be used before completing the final product. Knowledge of editors and operating systems are highly desired. Examining the incorrect area could result in tremendous amounts of wasted time [7]. Placement of breakpoints are critical to efficiently guide a search for error isolation.

System or user queries require a language interface. Parsing an input response is a tedious task dependent on the complexity of the response admitted. Languages parsed are described by a grammar. The grammar is a set of definitions consisting of tokens. These definitions determine legal combinations of tokens. A method of describing the syntax description associated with language development is the Backus-Naur form (BNF). BNF is a shorthand for describing language syntax providing a straightforward method for parser design [13]. The complexity of the BNF description

or language definition is determined by the amount and sophistication of responses permitted. The amount of storage allocated for prompts and responses is determined by language complexity. A linked-list organization is suitable for preserving prompts and responses.

Directly related to operational concepts and interface languages are problems of information management. Vast amounts of data resulting from these problems about program execution and debugger status are maintained by the operator. This is especially difficult for source code of ample size. The operator must always be aware of program execution and system status in current debugging environments. Normally, program listings in the form of memory maps and list files are kept close for examination during typical error localization sessions. These must be kept current to preserve proper symbol addresses and statement locations within the program segment. Minor changes to the program during a debug session can easily become a tiresome task if program listings and memory maps are not kept current. Terminating the debugger is usually required to modify the suspect program and maintain a current set of program listings and memory maps.

Responding to system or user queries requires an information data base. Depending on system capabilities, this data base can become extensive. Typical systems allow symbolic debugging capabilities. Users need not reference

program variables using memory addresses. This requires construction and maintenance of symbol tables or similar constructs. The compiler commonly places symbolic information into the object modules it generates. The linker is designed to recognize this information and carries it throughout the run-time phase. This permits referencing program information during the debug session.

Increasing system capabilities requires more complex structures. ACES provides a convenient means to retrieve program information. The concept requires a data base comprised of four tables, each table performing a specific function [11]. Korel describes a system requiring a dependence graph. This structure defines the relationship between program instructions by categorizing them into specific groups [7]. Additional systems such as FACES [10], FAST [2], and DAVE [9] require similar constructs.

These systems normally require complex links and pointers for maintenance. Additional mechanisms such as backtracking are required for information extraction. Dependent on the number and complexity of structures, this technique might be inadequate, requiring more complex traversal techniques.

A principal difficulty of debugger systems is adaptability. Conventional systems are designed for specific hardware and languages and are restrictive in the sense that a separate debugger is required for each language. These

restrictions inhibit programmers by limiting language selection associated with debugger support. Many designs are inefficient because of this limitation.

Limiting debugger capabilities are machine independence and flexibility [10]. Designing debuggers for specific systems stifles portability among machines. Consideration to the host debugger language and choice of operating system are of utmost importance. Selection of those promoting portability and machine-independence are desirable. Languages and operating systems currently exist supporting portability [5,12]. Typical systems restrict the operational environment.

Location and correction of program errors is a time consuming and complex process. Several techniques exist to do this function. Programs may initially contain an enormous amount of problem areas that require use of the editing-compiling-linking (ECL) cycle to be extremely active. To alleviate this cycle, programmers attempt program patching of the questionable source. This permits continuation of the debugging cycle. The difficulty with patching is the level at which the patch must be made, using hex code or mnemonics. Even the most experienced programmer can have difficulty grasping this concept. Patches are difficult for a variety of reasons; high-level language compiler conventions, size of patch, and patch incorporation may generate entirely different code [15].

The most common method is breakpoint placement at the suspect problem area. Once the system breakpoint is executed, variables or test condition outcomes can be modified. However, difficulty arises in keeping track of several modifications. Programmers resort to extensive ECL cycles to update and keep programs current [15]. Once a breakpoint is executed, trace buffers are provided to record past instructions before breakpoint termination. This gives an accurate account of the path executed up to this point in time. Many times these provide little or no information. For example, if an infinite loop is encountered, the trace buffer will contain isolated addresses with no information of past instructions executed. The actual area of incorrectness is lost. On reaching these breakpoint areas, debugging procedures must be available to the programmer for immediate action. Typical systems provide menu driven prompts to guide the user. These procedures are complex and operator familiarity is a required prerequisite.

Although most applications are developed in high-level languages, many debuggers reduce high-level language programs to assembly-language equivalents. Once a breakpoint is reached, the programmer is normally forced to translate between high-level source terms and low-level processor specific terms [8]. This increases frustration and injects confusion into the debugging environment. Access to program variables is difficult. Typical debuggers provide limited symbolic access to variables. This access is normally to

variables of the most basic program types. More complex structures such as user-defined data types, stack-based dynamic variables, and structured arrays are examples of data acquisition requiring more complex handling methods [15]. Observing program data within these complicated constructs requires a more complex command language to be learned and designed.

B. Enhanced Methods for Traditional Designs

Techniques for improving the quality of debugger systems software are presented in this section. The trend is toward automated processes with emphasis on minimizing operator involvement and responsibility. Automating processes and minimizing user involvement can reduce the time spent inside the ECL cycle of Figure 4. As the development phase proceeds, this cycle of design can become costly if the appropriate tools are incorrectly incorporated or inefficiently operated. Reducing time spent inside this cycle can increase productivity.

Automating processes minimizes the amount and complexity of system debugger concepts and the interface language respectively. Automating the systematic approach of error isolation and placing most debugger decisions on the host can relax these requirements. A more complex system design is required resulting in an inverse reaction to operator involvement. Time normally spent learning

complicated interface languages and operational concepts can be spent in the design stage of software development producing more efficient and compact code. The key to software productivity focuses on minimizing human interfaces [8].

Parser complexity can be considerably reduced by simplifying queries and curtailing responses requiring translation. A reduction in parser sophistication can simplify a language's BNF description. Storage required and structure maintenance can be decreased. Assuming the system is guiding the debugger session, a minimum amount of information is required from the operator. The result of automating the error localization approach results in a simplified parser via limited responses requiring translation.

Hindering the user with maintenance of source listings and memory maps can be alleviated. The system can provide the user with information pertinent to the current breakpoint setting under investigation. Sustaining source code information can provide the user with an instantaneous view at any time. Modifications to the source code can be achieved from within the debugger and the compile-link portion automated. Standardizing or providing editor selection familiar to the user can increase performance associated with error correction capabilities. The current trend is toward a more automated approach for eliminating or relaxing these requirements.

Providing structures for sustenance of symbolic information is imperative. System queries require familiar terms to simplify user interfaces. By automating the debugger process, notably the power to make simple decisions, the amount and complexity of symbolic information stored can be minimized. Requirements of relaying vast amounts of information to the user are no longer essential. Reducing storage requirements decreases the complexity of structural maintenance. The user is nevertheless provided with ample information about source code modifications and system queries when necessary.

The data base created for interrogation should be independent of the source language. Intel Corporation's PSCOPE debugger uses this technique by providing a variety of compilers for a single machine [8]. This allows the user to choose the optimum language for a particular task. Debug information is integrated into the language compiled, independent of language selection, to be used on a single machine. Data base designs promote language independence by expanding the number of languages a system can support.

The process of error isolation can be automated. Performing a suitable analysis on input programs before debugging provides valuable information necessary in error localization. This information can be stored with symbolic information determined at compile-time. Korel [7] uses the dependence graph to provide information pertinent to

conducting error isolation through breakpoint placement. User responsibility for breakpoint placement is no longer required. In eliminating breakpoint placement, the user need not be concerned with system operational concepts, learning complex interface languages, or maintaining and decoding debug information.

C. Summary

The efficiency of the above issues affect the following consequences. They are as follows:

(1) Time.

(2) Cost.

(3) Performance.

Time is directly proportional to cost and inversely related to performance. Those requirements not executed efficiently are time consuming. As a result, production costs are increased. Increasing system features and complexity will result in high performance. However, this performance will not be realized until familiarity with debugger operations and interface languages are fluently executed. This orientation process is lengthy in nature. Minimizing user involvement reduces overhead time commonly required for system familiarization. Replacing human intervention with automated processes is desirable resulting in increased system performance inversely affecting time and cost issues.

In summary, those issues associated with debugger usage

and design have side-effects directly related to time, cost, and performance. Efficient, automated operation is desired to minimize system dependence on user involvement.

Chapter IV

SYSTEM DEVELOPMENT AND RESULTS

Design issues associated with typical debugger systems were presented in the previous chapter. With these issues were techniques for improving conventional designs through process automation. Verification of these techniques dictate the development of a prototype system design. The objective of this design is to substantiate these concepts through test case evaluation of the prototype.

The Automatic Visual Debugger (AVD) is a prototype system designed to comply with these concepts. The complete process of error localization and key to system success revolves around automation of debugger functions. Equally important is the communication of pertinent information to the user. This is performed visually by displaying system status and program information continually throughout the automated debugging process.

A. Prototype System Development

Automated systems can be categorized by three distinct characteristics:

(1) Function.

(2) Application.

(3) Operational mode.

The functional classification describes what capacity AVD will perform. The specific function is source program dynamic analysis, an automated tool for monitoring program run time behavior. During the execution of a program, certain information is gathered about source code behavior. This information can be either used by the debugging system or communicated to the user.

The application classification determines that phase of program development the AVD system will encompass. From the beginning, AVD has been designed to aid in the testing and debugging phase of program development.

The operational mode determines if the source code will be analyzed before execution, static, or during execution, dynamic. Static analysis does not require program execution to determine information [10]. AVD assumes a suitable static analysis has been performed before system execution. This implies that the suspect source code is free of compilation errors and a suitable program data base is generated for future manipulation. This data base can be constructed as a dependence graph, symbol table, or combination of structures for information extraction. Dynamic analysis requires source code execution to do an adequate investigation of program characteristics [10]. As stated by the functional

classification and reinforced here, AVD is a dynamic analysis system.

The prototype system was designed and written in the C programming language. In addition, the prototype system was developed on the Unix operating system. Both C and Unix are highly portable and flexible producing compact and efficient code.

The Unix operating system is available on many machines, from smaller microcomputers and minicomputers to the largest mainframes. Unix is available on many different machines and is therefore termed portable by the scientific community. This criteria is important for AVD to meet the specific requirement of flexibility presented in Chapter III. Portability is largely because the C programming language with about 95% of the Unix operating system written in C [12]. A novel set of features, utilities, and system portability are reasons for Unix popularity and success. Multitasking and shell functions allow more productivity permitting AVD to do functions (nested processes) not normally allowed.

The C programming language is a modern systems language. The C language has high-level constructs for efficient, modular programming in addition to low-level functions for manipulating data at the machine level [12]. Low-level abilities allow specification of details in user programs to achieve maximum computer efficiency. High-level

abilities promote programming efficiency by concealing details of the computer's architecture [5]. High and low level traits make C an excellent language for AVD processing. C can be used to write machine-independent programs, has high level features for flexibility, and can still be used for systems programming. These features make C portable, standard, and powerful. Programs written in C are fast and require small amounts of storage. Powerful tools are built into the C language for many functions, specifically string processing, which is much needed by AVD. Teamed with Unix, requirements relating to flexibility and machine independence are easily done by the AVD system. Appendix B contains the prototype system software.

The AVD system consist of a three phase process. These phases constitute:

(1) Initialization.

(2) Error localization processing.

(3) Termination.

Each phase of operation will be comprehensively covered. A condensed version of the error localization cycle is shown in Figure 5.

B. Initialization Phase

The initialization phase of AVD is graphically presented in Figure 6.

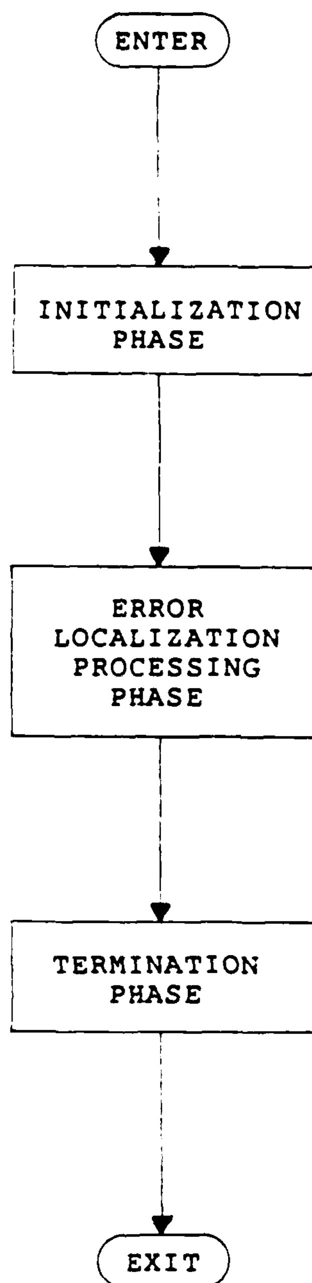


Figure 5: Error Localization Cycle

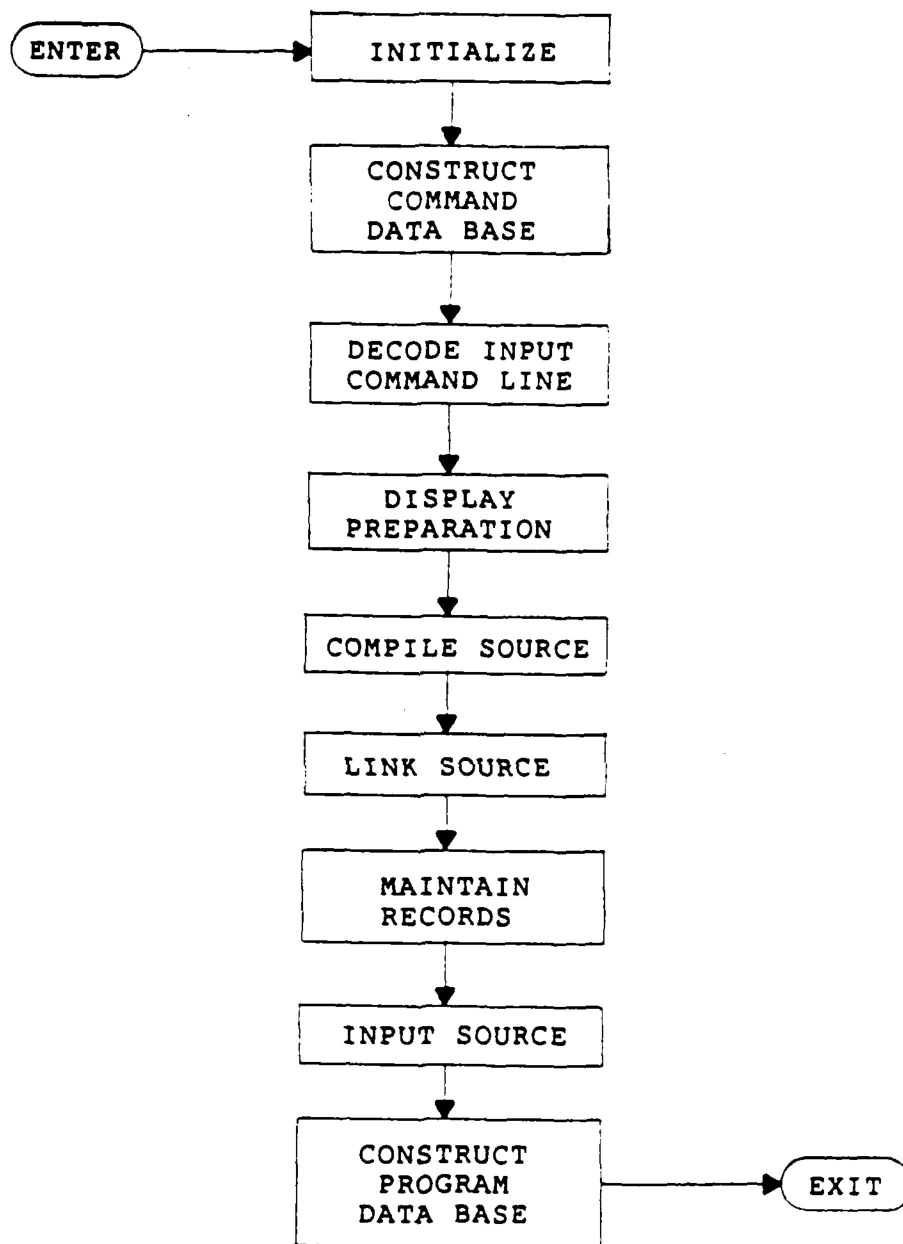


Figure 6: Initialization Phase

The command data base is comprised of Unix commands. These commands are principally used for sustaining current object files after source file modifications. Modifications can be executed by the user or AVD. User modifications to source code occur normally after errors are located. AVD modifications are because breakpoint placement requires continuous monitoring. Additional commands are provided for display, editing, and other facilities. The command data base is easily expanded to include any commands desired. Of greater significance is the ability to automatically execute these commands from within the debugging process. The C library function SYSTEM provides for executing Unix commands within the AVD debugging phase. These two factors favorably influence productivity.

The input command line is decoded to obtain information necessary for maintaining and modifying the correct source file. The command line is scanned and relevant information extracted. This information, normally source code filename, is stored and concatenated with the appropriate Unix commands for future applications. The command line must be scanned if multiple module names exist for a single program design. This eases the programmer restriction of single module program designs by allowing independent compilations of separate modules.

The display incorporates CURSES [14], a screen oriented program package for interfacing AVD to the terminal display.

During the initialization phase, arrangements are made to incorporate CURSES. Library routines are called to initialize the screen and configure keyboards dependent on program specifics and hardware used. Display functions will be discussed thoroughly in Section C. For now it is only necessary to know that some initialization occurs to incorporate displays to provide a better user interface and that this is the logical place for that initialization to occur.

Continuing through the initialization phase, efforts are now focused for final preparations of the source code. As stated earlier, it is assumed that a suitable static analysis has been completed on the source code. However, for program usage the source code is compiled and linked with the resultant object code placed in the default file, A.OUT. Following these commands, AVD issues the appropriate Unix commands for insuring all object files are current. These filenames are determined from the command line before execution. It is important to remember that these functions are being executed automatically without user intervention.

Equally important is the input of program source code for future display. This eliminates the lengthy listings and memory maps that are ordinarily used in debugging sessions. The procedure LEXICON completes the task of inputting the program source code. LEXICON then calls the C library function MALLOC to acquire this code. MALLOC dynamically allocates memory for each line of source code. An array of

pointers independently references each line of code.

The final stage of initialization is program data base construction. The program data base is built for future information extraction. Algorithms obtain information by scanning the source and object code files. These algorithms should be capable of examining any type source and producing a common data base. The program data base is a dependence graph or similar structure and provides information to AVD necessary to localize a program error. AVD assumes an appropriate data base exist before program execution.

The initialization phase automates many tasks normally considered as the user's responsibility. The task of compiling and linking is controlled by AVD. The task of maintaining current listings is automated and source code information is provided to the user on request. The automation of these two tasks eliminates the need for procedures that are frequently used throughout typical debugging sessions. Automation of these processes reduces wasted time and increases system performance.

C. Error Localization Processing Phase

Figure 7 graphically represents the error localization processing phase.

The error localization processing phase is an infinite loop. The loop is terminated by issuing an EXIT command.

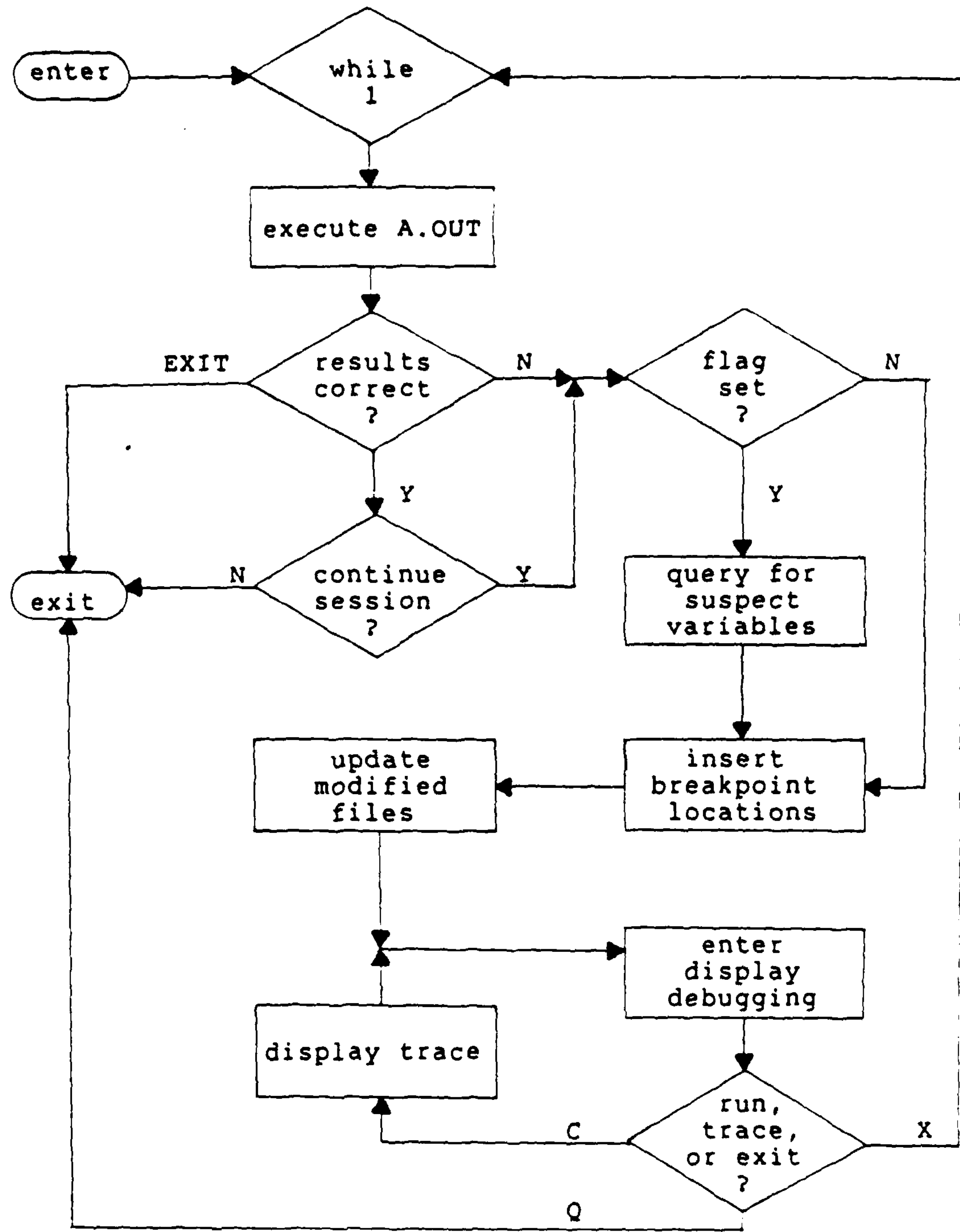


Figure 7: Error Localization Processing Phase

In addition, system prompts provide an alternate means of terminating the debug session.

Initially the program is executed and results displayed to the user. AVD (Automatic Visual Debugger) queries the user to determine if these results are correct. Dependent on the user's response one of two paths may be taken at this time. The user may immediately exit AVD or enter a debug session for error localization. Exit may be desired if program results are correct. The user may also specify a different variable for inspection if so desired. System prompts are provided for supporting these options.

The second path leads to a debugging session with AVD. Assuming the results are incorrect, the first stage of automatic visual debugging occurs. AVD queries the user for the variable suspected of being in error. A flag setting prevents the system from continually querying the user for suspect variables. This flag permits a query only when needed. All locations within the source for each program variable are recorded in a data base. In addition to variable locations, information about variable usage may be recorded. The sum of this information, location and usage, assist AVD in the critical placement of breakpoints. For example, a suspect variable may be inside a loop. Should the breakpoint be placed to observe the suspect variable with each loop iteration or at the loop conclusion? Algorithms exist to resolve these issues [7]. Traversal methods

scan the program data base to extract information about the suspect variable. Backtracking is an efficient method for traversing the program data base and acquiring information to assist these algorithms in determining breakpoint location.

To accurately place a breakpoint requires three steps. Step one scans the program data base for desired breakpoint location and suspect variable usage. Algorithms incorporate this information to resolve issues and determine exact breakpoint location. Step two physically places the breakpoint by arming system hardware to recognize certain conditions. Step three updates all files if necessary and executes the desired breakpoint setting.

AVD simulates these three steps associated with breakpoint placement by executing the following operations. These operations occurred at points in time corresponding to a genuine debugging process. When breakpoint placement was necessary and at the appropriate time, the current process of the debugging system was suspended. A subprocess was spawned to emulate typical breakpoint placement. The subprocess can be visualized as time spent to normally place a breakpoint in a conventional system. The spawning process was begun by the C library function SYSTEM. Corresponding to step one above, the program data base is traversed to determine information about breakpoint placement. Corresponding to step two, PRINTW statements are incorporated to simulate

arming hardware to recognize breakpoint settings. A PRINTW statement represents a typical breakpoint and is placed at locations determined by traversing the program data base. The EMACS editor assists in providing a means of inserting these statements into the program source. Corresponding to step three, the source file is updated by executing a Unix MAKE command. MAKE is capable of compiling and linking files based on their prior history. MAKE operates on a default file called MAKEFILE. This file can easily be input, modified to include additional modules detected in the command line, and output for continued usage. On executing the MAKE command, the subprocess is terminated and the suspended process returned to active status. The second half of step three involves executing the program up to the breakpoint location. This portion is provided as an option inside the display trace.

The subprocess succeeds in breakpoint placement by scanning the program data base, physically placing the breakpoint, and performing file maintenance. Automating these three functions spares the user of tedious tasks and provides a simplistic approach toward the breakpoint placement process. These functions would be combined into a single procedure for a genuine debugging system as demonstrated by Korel [7].

At this stage of operation a breakpoint has been successfully placed and AVD enters the trace display procedures

shown in Figure 7. Now the AVD system incorporates CURSES [14]. CURSES is a powerful programming package capable of enhancing the display interface. This is done through multiple windows, text functions such as highlighting and bold type, and controlling cursor movements. AVD uses CURSES to create a two window concept. These windows provide trace display information and expert system guidance. The first, TRACE_WINDOW, provides the user with information about breakpoint placement, suspect variable identification, and next choice of system command. The second, STDSCR, is an expert system guidance window. STDSCR provides information transfer between the user and AVD through system queries and user responses. This is the window for normal operations outside the trace display procedures.

Display operations are entirely controlled by procedure WINDOWS. All situations involving command response are highlighted for emphasis and furnished to the user in menu fashion. The menu offers command choices corresponding to a specific alphanumeric key and is displayed to the user for selection. These features assist the user in the decision making process by providing a list of commands allowed at specific points in the debugging phase. This eliminates the necessity to learn system commands and operations.

On entering the trace display, WINDOWS provides a trace display menu. Three options are provided: exit, observe trace settings, or execute the current breakpoint setting.

The trace display menu can be easily expanded to include additional commands as desired. The exit command is self-explanatory. Pressing the alphanumeric key Q will end the AVD debug session and return to the host operating system.

The execute command allows the user to run the program with the current breakpoint setting. Pressing the alphanumeric key X provides this function. Starting this function terminates the trace display window, re-enters the standard screen, and executes current program settings. The error localization phase is repeated.

The third option allows observance of trace settings. The trace settings include display of breakpoint location and suspect variable. The current breakpoint location is highlighted and displayed to the user. Source statements surrounding the breakpoint are displayed in conventional video. This emphasizes the breakpoint setting and orients the user to breakpoint location within the source code. The variable suspected of being in error is highlighted and displayed. Pressing the alphanumeric key C executes this option. Terminating this option returns the user to the trace display menu. The user is therefore allowed to observe trace settings before executing the program.

Each window is provided with a message library. These libraries contain queries executed within their respective windows. Each window contains a set of specific queries and messages transmitted to user. Procedure PROCESS_MESSAGE

consists of messages in the STDSCR window and include those used in expert system guidance. These entail queries about suspect variable identification, correctness of results, and others to determine needed information. Procedure HIGHLIGHT consists of messages in the window TRACE_WINDOW and include those used during the trace display portion of AVD. These entail options about program execution and trace display viewing. The messages in this library are highlighted for emphasis. Each library can be easily expanded to accommodate several messages. However, crowding the display with numerous messages tends to increase user frustration and confusion.

The error localization processing phase is complete. Guidance through this phase was described above. Additional techniques for automation are provided. Selecting the breakpoint location, placing the breakpoint, and executing the program to provide breakpoint information was described. Providing command selection during the error localization phase eliminates the need for learning system commands and concepts. Automating these processes increases productivity by minimizing user involvement.

D. Termination Phase

Figure 8 graphically represents the termination phase. In the WINDOWS procedure, mechanisms exist to detect

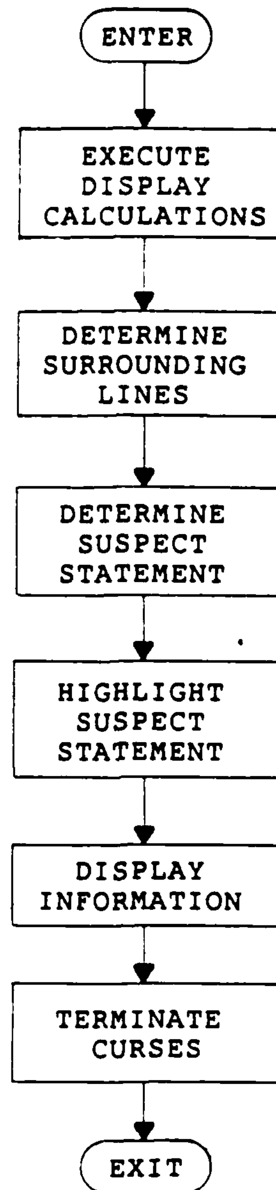


Figure 8: Termination Phase

and conclude the error localization process. These algorithms base their decisions on information acquired from correlating user queries and data base data. Error isolation leads to procedures for terminating the AVD session and displaying final results. Procedure LOCALIZED is responsible for displaying final results.

Before displaying the source statement containing the error, LOCALIZED executes calculations to display surrounding lines. This orients the user to the error location within the source code. The line number, source code statement, and surrounding lines are displayed. The suspect statement is highlighted for emphasis.

Transmitting the information to the user completes the error localization cycle. The final task is to reset terminal hardware and displays to their original settings. This is done by executing the appropriate commands to exit the CURSES package. The termination of CURSES results in the exit of AVD and return to the host operating system.

E. Prototype System Verification

The prototype system was verified by designing three typical program test cases. Each test case had specific errors deposited in the source code to be detected. These errors were different for each test case. Source code for test case programs are located in Appendix A.

Test case one is a sequential program without control constructs or procedure calls. This suggest a data dependence relationship between instructions [7]. The program function is to calculate an employees net pay. The error to be detected is a numerical computation; addition is shown but multiplication is required. The statement, RP = RH + PR, should have been RP = RH * PR. This was determined by scanning the program data base and determining that input values to RP should be verified. AVD places breakpoints in the program to determine RH and PR correctness. On reaching these breakpoints queries are made to the user to determine variable status. Both variables are found to be correct suggesting RP was calculated improperly. AVD successfully isolates the error to a single statement and transmits this information to the user. This test case is similar to one use by Korel [7].

Test case two contains control constructs and is more complex in nature. All data dependences are possible. Data is admitted from the standard input (monitor) and a record kept of the number of lines and characters transmitted. The error to be detected is an initialization anomaly. The statement, NUM_OF_LINES = 1, should have been NUM_OF_LINES = 0. The error is successfully isolated by placing breakpoints at variable NUM_OF_LINES locations. The first breakpoint location is in a control construct. AVD algorithms using information from the program data base resolve issues of breakpoint placement. It is determined that the breakpoint

should be placed within the construct. This will break and report the NUM_OF_LINES value with each iteration. Queries to the user verify that the variable is initially incorrect after one iteration. AVD traverses the program data base for checkpoints and determines usage at one additional position. AVD isolates this location and through user queries determines the variable was initialized incorrectly. These results are transmitted to the user.

Test case three is a bubble sort routine. The most complicated of the three, this test case contains both control constructs and procedure calls. Again, all three data dependences are possible. The error detected is caused from improper variable usage. The statement, TEMP = ARRAY [INDEX], should have been TEMP = ARRAY [INDEX_1]. Through a series of placing breakpoints and issuing user queries, information is collated and a conclusion determined. AVD successfully localizes the error location and transmits the results to the user.

Test case three demonstrates the ability of AVD to operate on multiple module programs. Case three consists of two modules. Modifications to either module results in successful error localization. AVD permits the observance of module listings, breakpoint placement within either module, and insures that the appropriate files were correctly maintained. As stated earlier MAKEFILE can be modified to include several modules.

The AVD system successfully isolates each error for all test cases. Computational syntax (operator), variable initialization, and variable usage errors are detected. Location of each error is then transmitted to the user. AVD successes are from incorporating methods developed in Chapter III into the prototype system of Chapter IV. Automating processes is the key to a more productive debugging environment.

Chapter V

CONCLUSIONS AND RECOMMENDATIONS

A. Conclusions

This research has concentrated on process automation to create a simplified debugging environment. Methods were presented to accomplish automation. These methods focused on automating tasks normally performed by the user. Techniques suggested increase system performance by deceasing cost and time factors associated with conventional systems.

Each phase of the error localization processing cycle uses automation. The initialization phase exploited automation in program preparation and source code management. Compiling and linking operations merged to form a single task to prepare source code for debugger investigation. The aforementioned task is applied throughout the error localization process. The source file was input for display purposes and a program data base built for future interrogation. Each task supported automated processes of succeeding phases. The initialization phase primary responsibility is to prepare source code for future manipulation and provide support for automated processes. This responsibility was met by the initialization phase.

The error localization processing phase is completely automated. The phase itself is driven by automation through system queries. Responding to these queries assist the prototype system in the decision making process. Within the cycle are additional automated processes for breakpoint placement and display functions.

The process of breakpoint placement was automated. A program data base was constructed by performing a flow analysis of the source code during the initialization phase. The program data base was interrogated and information extracted. This information assists algorithms in drawing conclusions about breakpoint placement. The breakpoint was then automatically placed within the source code and executed on the users request.

A display provided observance of different debug data. The display section was completely automated and provided status information to the user.

The termination phase displays concluding data and terminates the debug session, returning control to the host operating system. All hardware, including displays and keyboards, are reset to their original settings. The termination phase is completely automated and internal operations concealed from the user.

The keyword is automation. From the above summations, the prototype system met all research objectives. A

framework was provided for future debugger designs and process automation used to create a simplified debugging environment.

B. Recommendations

The prototype system is a model for future debugger designs. Many features demonstrated could easily be incorporated into current debugger systems. Inclusion of these facilities could provide a significant increase in current system performance.

Many areas of the prototype system have potential for future research. The system provided was a prototype for demonstrating new and innovative ideas. The use of a program data base was demonstrated. A universal program data base is required to make the prototype system flexible. Research into the area of a universal program data base could prove challenging. Automatic breakpoint placement was demonstrated. This area could be further developed with emphasis place on algorithm design. Each of these areas could be a basis for further investigation. Completion of these tasks could help make the Automatic Visual Debugger (AVD) a reality.

APPENDICES

APPENDIX A

TEST CASES

```
/*
** case1: financial program
*/

#include <stdio.h>

main()
 {
 FILE *file;

 float RH, OH, PR, FTR, NE, RP, OP, GP, FTW, STW, SSTW,
       NP, input_data[5];
 int count;

 file = fopen("case1.dat", "r");

 while (fscanf(file, "%f",  &input_data[count]) != EOF)
  ++count;

 RH  = input_data[0];
 OH  = input_data[1];
 PR  = input_data[2];
 FTR = input_data[3];
 NE  = input_data[4];

 RP = RH + PR; /*error: RP = RH * PR*/
 OP = 2 * OH * PR;
 GP = RP + OP;

 FTW = (GP - 10 * NE) * FTR;
 STW = GP * 0.05;
 SSTW = GP * 0.06;

 NP = GP - FTW - STW - SSTW;

 printf ("net pay = %f0, NP);
 }
```

```
/*
** case2: file line and character count
*/

#include <stdio.h>
#define CTRL(c) ('c' & 037)

main ()
 {
 int num_of_chars, num_of_lines;
 char in_char;

 num_of_chars = 0;
 num_of_lines = 1; /*error: = 0*/

 while ((in_char = getchar()) != CTRL(d))
  {
  if (in_char != '0)
   ++num_of_chars;
  else
   ++num_of_lines;
  }

 printf ("0um_of_chars = %d0, num_of_chars);
 printf ("num_of_lines = %d0, num_of_lines);

 }
```

```
/*
** case3: bubble sort routine
*/

#include <stdio.h>
#define amount 10

main()
 {
 FILE *file;

 int array[amount], index, index_1, temp,
     number, temp_index;

 file = fopen("case3.dat",  "r");

 index, temp = 0;

 while (fscanf(file, "%d", &array[index]) != EOF)
  ++index;

 temp_index = index;
 --index;

 while (index > 1)
  {
  for (index_1 = 0; index_1 < index; ++index_1)
   {
   if (array[index] < array[index_1])
    {
    temp = array[index]; /*error: index_1*/
    array[index_1] = array[index];
    array[index] = temp;
    }
   }
  --index;
  }

 for (index_1 = 0; index_1 < temp_index; ++index_1)
  printf ("==> %d 0, array[index_1]);

 message(); /*to prove 'make' works in all cases*/
 } /*main*/
```

```
/*
** Separate module used to prove that all
** facets of the 'make' command are operational.
** Used in coordination with case3.
*/

message()

 {
 printf("** Make test completed **0);
 }
```

APPENDIX B

PROTOTYPE SYSTEM SOURCE CODE

```
#include <curses.h>

#define LINE_LENGTH 80
#define FILE_LENGTH 50

static int case_1[] = {28, 22, 20, 16, 14, 0};
static int case_2[] = {16, 9, 0};
static int case_3[] = {27, 25, 13, 0};

static char str1[20] = "cat ";
static char str2[20] = "emacs ";
static char str3[20] = "vi ";
static char str4[20] = "make ";
static char str5[35] = "cc -c ";
static char str6[35] = "cc ";

WINDOW *trace_window;

char *line_ptr[FILE_LENGTH];
char variable[15];
int flag, *selection, msg_no;
int beg_line, end_line, y_coord;
```

```
main(argc, argv)
int argc;
char *argv[];

 {
 char response[15];

 initialize(&argc, argv);

 mvprintw(20, 0, "Debugging program %s", argv[1]);

 mvprintw(21, 0, "Preparing program %s", argv[1]);
 system(str5);
 system(str6);

 lexicon(argv);
 refresh();

 sleep(5);

 while(1)
  {
  msg_no = 7; process_message(&msg_no, argv);
  system("a.out");

  msg_no = 8; process_message(&msg_no, argv);

  gets(response);
  keyboard_interface(response);
  }
 }
```

```
keyboard_interface(string)
char *string[];

 {
 if (!(strcmp("no", string)))
  {
  if (flag)
   {
   msg_no = 1; process_message(&msg_no);
   gets(variable);
   flag = FALSE;
   }
  msg_no = 2; process_message(&msg_no);
  system(str2);
  system("make -f case");

  clear(stdscr);

  windows();
  }
 else if (!(strcmp("yes", string)))
  {
  msg_no = 3; process_message(&msg_no);
  gets(string);
  if (!(strcmp("yes", string)))
   {
   msg_no = 4; process_message(&msg_no);
   exit();
   }
  else
   {
   msg_no = 5; process_message(&msg_no);
   flag = TRUE;

   return(keyboard_interface(string));
   }
  }
 else if (!(strcmp("exit", string)))
  {
  msg_no = 6; process_message(&msg_no);
  exit();
  }
 }
```

```
lexicon(source_file)
char *source_file[];

 {
 FILE *file;
 char line[LINE_LENGTH];
 char *install();

 file = fopen(source_file[1], "r");

 while (fgets(line, LINE_LENGTH, file))
  install (line);

 mvprintw(22, 0, "Installed program %s0,
          source_file[1]);
 }
```

```
char *install(line)
char *line;

 {
 static int x=0;
 char *malloc();
 char *string_ptr;

 if (string_ptr = malloc(strlen(line) + 1))
  {
  strcpy(string_ptr, line);
  line_ptr[x] = string_ptr;
  ++x;
  return;
  }

 mvprintw (22, 0, "out of memory0);
 refresh();
 exit();
 }
```

```
initialize(num_of_args, source_file)
int *num_of_args;
char *source_file[];

 {
 char *object_file;

 strcat(str1,  source_file[1]);
 strcat(str2,  source_file[1]);
 strcat(str3,  source_file[1]);
 strcat(str4,  source_file[1]);

 flag = TRUE;

 if (!(strcmp("case1.c",  source_file[1])))
  { selection = case_1; object_file = "case1.o"; }
 else if (!(strcmp("case2.c",  source_file[1])))
  { selection = case_2; object_file = "case2.o"; }
 else
  { selection = case_3; object_file
    = "case3.o case3.1.o"; }
 strcat(str6, object_file);

 if (*num_of_args != 3)
  strcat(str5, source_file[1]);
 else
  {
  strcat(str5, source_file[1]);
  strcat(str5, " ");
  strcat(str5, source_file[2]);
  }

 initscr();
 nl();
 echo();
 crmode();

 clear(stdscr);
 }
```

```
process_message(msg, source_file)
int *msg;
char *source_file[];

 {
 if ((*msg != 1) && (*msg != 8) && (*msg != 9))
  clear(stdscr);

 move(22, 0);

 switch(*msg)
  {
  case 1:
   printw("Enter variable  suspect of being
          in error: ");
   break;
  case 2:
   printw("Inserting breakpoints0);
   break;
  case 3:
   printw("Terminate JEANS session? ");
   break;
  case 4:
   printw("JEANS session concluded!0);
   break;
  case 5:
   mvprintw(21, 0, "Assume you wish to
            look at another variable!");
   break;
  case 6:
   printw("JEANS session concluded!0);
   endwin();
   break;
  case 7:
   printw("Executing program %s0, source_file[1]);
   break;
  case 8:
   printw("Were program results correct? ");
   break;
  case 9:
   standout();
   mvprintw(y_coord, 0, "Line %d: %s",
            *selection, line_ptr[*selection]);
   standend();
  }

 refresh();
 }
```

```
windows()
 {
 int count, action, display_code;

 noecho();
 trace_window = newwin(LINES, COLS, 0, 0);

 for(;;)
  {
  refresh();

  wmove(trace_window, 12, 13);
  display_code = 1; highlight(&display_code);

  wmove(trace_window, 23, 0);
  wrefresh(trace_window);

  action = getch();
  switch(action)
   {
   case 'c':
    werase(trace_window);

    wmove(trace_window, 20, 20);
    display_code = 2; highlight(&display_code);

    beg_line = *selection - 5;
    end_line = *selection + 3;

    y_coord = 3;
    for (; beg_line <= end_line; ++beg_line)
     {
     if (beg_line != *selection)
      mvwprintw(trace_window, y_coord, 5,
                "%s", line_ptr[beg_line]);
     else
      {
      display_code = 3; highlight(&display_code);
      }
     ++y_coord;
     }
    display_code = 4; highlight(&display_code);

    wmove(trace_window, 15, 0);
    for (count = 0; count < COLS; count++)
     waddch(trace_window, '=');

    wmove(trace_window, 23, 0);
    touchwin(trace_window);
    wrefresh(trace_window);
```

```
          while ((action = getch()) != ' ');

          touchwin(stdscr);
          break;

         case 'x':
          ++selection;
          if (*selection == 0)
           localized();
          move(23, 0);
          clrtoeol();
          touchwin(stdscr);
          echo();
          refresh();
          return;

         case 'q':
          move(23, 0);
          clrtoeol();
          touchwin(stdscr);
          echo();
          refresh();
          endwin(trace_window);
          endwin();
          exit(0);
         }
      }
   }
```

```
localized()

 {
 static int value = 9;

 --selection;
 if (*selection == 14)
  selection = selection - 2;
 else if (*selection == 13)
  selection = selection - 1;

 beg_line = *selection - 5;
 end_line = *selection + 3;

 clear(stdscr);

 y_coord = 7;
 for (; beg_line <= end_line; ++beg_line)
  {
  if (beg_line != *selection)
   mvprintw(y_coord, 0, "%s", line_ptr[beg_line]);
  else
   process_message(&value);

  ++y_coord;
  }

 move(23, 0);
 clrtoeol();
 touchwin(stdscr);
 refresh();
 endwin(trace_window);
 endwin();
 exit();
 }
```

```
highlight(code)
int *code;

 {
 wstandout(trace_window);

 switch(*code)
  {
  case 1:
   wprintw(trace_window," PRESS 'q': quit
            'c': trace display  'x': execute ");
   break;
  case 2:
   wprintw(trace_window,
            " To continue press spacebar ");
   break;
  case 3:
   mvwprintw(trace_window, y_coord, 5, "%s",
              line_ptr[beg_line]);
   break;
  case 4:
   mvwprintw(trace_window, 0, 0,
              "Suspect variable: %s", variable);
  }

 wstandend(trace_window);
 }
```

BIBLIOGRAPHY

[1] Allen, F.E. and Cocke, J. "A Program Data Flow Analysis Procedure," Communications of the ACM, Vol. 19, No. 3, March 1976, pp. 137-147.

[2] Browne, J.C. and Johnson, David B. "FAST: A Second Generation Program Analysis System," Proceedings 3rd International Conference on Software Engineering, 1978, IEEE Cat. No. 78CH1317-7C, pp. 142-148.

[3] "EMV-51A Emulation Vehicle User's Guide," Issue 1, Intel Corporation, Santa Clara, California, December 1984.

[4] Fosdick, L.D. and Osterweil, L.J. "Data Flow Analysis in Software Reliability," ACM Computing Surveys, Vol. 8, No. 3, September 1976, pp. 305-330.

[5] Hancock, Les and Krieger, Morris The C Primer, 2nd edition, McGraw-Hill, New York, New York, 1986.

[6] Hecht, M.S. Flow Analysis of Computer Programs, Elsevier North-Holland, New York, New York, 1977, pp. 3-24.

[7] Korel, B. "A Program Error Localization Expert System," Proceedings of SPIE Vol. 635 Applications of Artificial Intelligence III, 1986, pp. 111-118.

[8] Maritz, Paul "Software Development," Intel Development Systems Handbook, Intel Corporation, Santa Clara, California, 1986, pp. 2.140-2.144.

[9] Osterweil, L.J. and Fosdick, L.D. "Some experience with DAVE -- A Fortran program analyzer," Proceedings AFIPS 1976 National Computer Conference, Vol. 45, AFIPS Press, Montvale, New Jersey, 1976, pp. 909-915.

[10] Ramamoorthy, C.V. and Ho, S.F. "Testing Large Software with Automated Software Evaluation Systems," Proceedings 1975 International Conference on Reliable Software, 1975, IEEE Cat. No. 75CH0940-7CSR, pp. 382-394.

[11] Ramamoorthy, C.V., Meeker, R.E. and Turner, J. "Design and Construction of an Automated Software Evaluation System,"Record 1973 IEEE Symposium on Computer Software Reliability, 1973, IEEE Cat. No. 73CH0741-9CSR, pp. 28-37.

[12] Sobell, Mark G. A Practical Guide to UNIX System V, Benjamin/Cummings, Menlo Park, California, 1985.

[13] Thompson, Beverly A. and Thompson, William A. "Inside an Expert System," BYTE, Vol. 10, No. 4, April 1985, pp. 315-330.

[14] "UNIX System V -- Release 2.0 Programming Guide," Issue 2, April 1984.

[15] Vannerson, Stuart "Debugging Catches up with High-Level Programming," Electronic Design, Vol. 30, No. 13, June 1982, pp. 121-126.

END
DATE
FILMED
5-88
DTIC