ARPA ORDER NO. 189-1

MEMORANDUM
RM-5772-ARPA
APRIL 1969

EXDAMS – EXTENDABLE DEBUGGING AND MONITORING SYSTEM

R. M. Balzer

PREPARED FOR:
ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

THIS DOCUMENT HAS BEEN APPROVED FOR PUBLIC RELEASE AND SALE; ITS DISTRIBUTION IS UNLIMITED

ARPA ORDER NO. 189-1

MEMORANDUM

RM-5772-ARPA

APRIL 1969

EXDAMS – EXTENDABLE DEBUGGING AND MONITORING SYSTEM

R. M. Balzer

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of ARPA.

DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

The RAND Corporation

1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

This study is presented as a competent treatment of the subject, worthy of publication. The Rand Corporation vouches for the quality of the research, without necessarily endorsing the opinions and conclusions of the authors.

Published by The RAND Corporation

PREFACE

This Memorandum describes an on-line debugging and monitoring system designed to facilitate experimentation with new on-line debugging and monitoring aids, and to provide flexibility in alternating among these aids as execution time is controlled in either the forward or backward direction. This study is part of the ARPA-sponsored research to improve man-computer interaction. The paper should be of interest to those concerned with a proper programming environment for research and development applications.

SUMMARY

This Memorandum describes EXDAMS (EXtendable Debugging and Monitoring System), a powerful set of source-level debugging and monitoring aids for higher-level computer languages. These facilities are of two types: *static*, which refer to a specific point in execution time; and *motion-picture*, which vary with execution time and can be viewed with execution time either advancing or reversing at variable speed (e.g., the user can watch his program executing, in reverse, backing up to some earlier state).

In addition to these facilities, the EXDAMS environment features: 1) the ability to alternate, at any point in execution time, between the data space (*what* happened) and the control space (*how* it happened), thus associating a program action with the exact statement(s) causing that action; and 2) easy extendability for new user-defined debugging and monitoring aids.

This Memorandum details the implementation of this environment, through a model of the user's program and a history tape of its behavior, and gives a short example.

CONTENTS

PREFACE iii

SUMMARY v

Section
I. INTRODUCTION 1

II. DESIGN GOALS 3

III. DEBUGGING AND MONITORING AIDS WITHIN EXDAMS 5
Static Displays 5
Error Analysis 5
Source Code 6
Flowback Analysis 7
Motion-Picture Displays 8
Values 8
Source Code 9
Map 9
Windows 11
Windows with Transitions 12

IV. THE EXDAMS ENVIRONMENT 13
Program Analysis 13
Compilation 14
Run-Time History-Gathering 14
Debug-Time History-Playback 15

V. THE ADDITION OF NEW DEBUGGING AND/OR MONITORING FACILITIES 17

VI. EXAMPLE 18
Original Source Program 18
Symbol Table 19
Model 20
Augmented Source Program 22
History File 24

Appendix
A. USE OF INDEX FIELD IN MODEL ENTRIES..... 26
B. STATEMENTS ADDED TO USER PROGRAM TO PRODUCE EXDAMS HISTORY..... 27
C. INFORMATION RECORDED IN HISTORY FOR EACH TYPE OF STATEMENT..... 29

REFERENCES 30

I. INTRODUCTION

With the advent of the higher-level algebraic languages, the computer industry expected to be relieved of the detailed programming required at the assembly-language level. This expectation has largely been realized. Many systems are now being built in higher-level languages (most notably MULTICS [1]).

However, the ability to debug programs has advanced but little with the increased use of these higher-level languages. As Evans and Darlay point out:

> We find that, broadly speaking, a close analog of almost every principal assembly-language debugging technique exists in at least one debugging system pertaining to some higher-level language. However, on-line debugging facilities for higher-level languages are in general less well-developed and less widely used (relative to the use of the languages) than their assembly-language counterparts.*

In general, system builders have merely copied the on-line assembly-language debugging aids, rather than designed totally new facilities for higher-level languages. We have neither created new graphical formats in which to present the debugging information, nor provided a reasonable means by which users can specify the processing required on any available debugging data.

These features have been largely ignored because of the difficulty of their implementation. The debugging systems for higher-level languages are much more complex than those for assembly code. They must locate the symbol table, find the beginning and end of source-level statements, and determine some way to extract the dynamic information--needed for debugging--about the program's behavior, which is now hidden in a sequence of machine instructions rather than

---

*Ref. 2, p. 41.

being the obvious result of one machine instruction. Is it any wonder that, after all this effort merely to create a minimal environment in which to perform on-line higher-level languages debugging, little energy remained for creating new debugging aids that would probably require an increased dynamic information-gathering capability?

EXDAMS (EXtendable Debugging And Monitoring System) is an attempt to break this impasse by providing a single environment in which users can easily add new on-line debugging aids to a system one-at-a-time without further modifying the source-level compilers, EXDAMS, or their programs to be debugged. It is hoped that EXDAMS will encourage the creation of new methods of debugging by reducing the cost of an attempt sufficiently to make experimentation practical. At the same time, it is similarly hoped that EXDAMS will stimulate interest in the closely related but largely neglected problem of monitoring a program by providing new ways of processing the program's behavioral information and presenting it to a user. Or, as a famous philosopher once almost said, "Give me a suitable debugging environment and a tool-building facility powerful (and simple) enough, and I will debug the world."

Since EXDAMS is currently being debugged and is not operational, no performance statistics are available.

II. DESIGN GOALS

EXDAMS was designed to satisfy three needs: first, as a vehicle to test some proposed, but unimplemented, on-line debugging and monitoring facilities; second, as an extendable facility to which new debugging and monitoring aids could be added easily, then tested; and, third, as a system providing some measure of independence of not only the particular machine on which it is being run and the particular implementation of the language being debugged and/or monitored, but also of several source languages in which users' programs could be written and debugged and/or monitored.

The normal techniques for on-line debugging, involving dynamic manipulation of a running program, were inappropriate for these three ambitious design goals for two reasons: first, because these techniques were both implementation-dependent and difficult to control; second, certain important facilities, such as the ability to run the programs backwards, are impossible with these techniques.

Therefore, the program to be debugged will run with an EXDAMS routine that will monitor it, collect necessary information about the program's actions, and store this information on a *history tape*. Subsequently, EXDAMS debugging routines can retrieve any information from the history tape, format it, and present it to the user. Thus, assuming the history tape is complete (i.e., contains all relevant data), *any* debugging and/or monitoring tool involves only retrieving, then formatting, data from this static file.

The parts of EXDAMS that analyze the program and collect its history (the program-analysis and history-gathering phases discussed in Sec. IV) are language dependent. However, the major portion of EXDAMS, and the portion chosen for experimentation--the debugging and monitoring routines--interact with only the history file.

They are therefore independent of both the implementation of the source language and the source language itself--to the extent the history file is independent of the differences between source languages, as it is for the common algebraic languages (PL/I, ALGOL, FORTRAN, etc.).

With this approach, the three design goals have been achieved. Any debugging and monitoring aids can be added to EXDAMS easily by writing the appropriate file-search and formatting routines. Moreover, these aids are independent of the implementation of the source language and, to a certain extent, of the source language itself.

However, efficiency has been sacrificed. The EXDAMS approach is based on the insulation from the running program that results from the production of a history tape of the program's behavior. The production and replaying of this history involve large amounts of I/O. However, the flexibility gained far outweighs the inefficiency introduced, especially when studying alternative debugging and monitoring aids.

The EXDAMS system output device is a cathode ray tube (CRT) display, and all the debugging and monitoring aids utilize its two-dimensional and high-data-rate capabilities. Some aids, in addition, use the CRT's true graphic (point and vector) and dynamic (time-variant) capabilities. The input devices are a keyboard and some type of graphical pointing device, e.g., a light-pen, RAND Tablet, joy-stick, mouse, or keyboard cursor.

Before describing how EXDAMS works and how new debugging and monitoring aids are added to the system, we present in the following section some of the aids currently being added to the basic EXDAMS system (described in Sec. IV) to give the reader a better understanding of the types of debugging and monitoring aids possible.

III. DEBUGGING AND MONITORING AIDS WITHIN EXDAMS

EXDAMS contains two types of debugging and monitoring aids--static and motion-picture. The *static aids* display information that is invariant with execution time (a time value incremented as each source statement is executed and used to refer to particular points in the execution of a program), such as the values of variables at the time an error occurred, a list of all values of a variable up to a given execution time, or a display of a portion of the source code.

The *motion-picture aids*, on the other hand, are execution-time sensitive; that is, the data they display may vary with execution time. These motion-picture aids include the last n values of a set of variables, the current instruction and subroutine, and the current values of a set of variables. The user can run motion-picture aids either forwards or backwards at variable speeds, by controlling execution time.

EXDAMS' most attractive features, from the user's standpoint, are a) his ability to control his program's execution time, moving at variable speed either forwards or backwards, while a debugging and/or monitoring aid constantly updates its display of information; and b) his ability to stop execution time at any point, switch to another aid, and continue perusing the behavior of his program.

STATIC DISPLAYS

Error Analysis

The user requests the value of certain variables at the time an error occurred. The system displays the value of these variables and all other variables in the error-causing source language instruction, the type of error, and the source instruction in error.

Source Code

A portion of the user's source code is displayed in optional formats that may include indications of the number of executions per statement and the removal of levels in the source code (such as the bodies of do-groups or the code in the THEN or ELSE clauses) below a certain depth, to afford the user a broader view of his program.

The user may request this display in two manners. He may call for the code around a certain label by requesting SOURCE AT and specifying a label, or a label plus or minus some number of source statements. He also may call for, at any time, the source code around the exact statement that caused a particular value of a variable by requesting SOURCE FOR and specifying the desired value (the source statement causing that value will be marked by its brightness). That is, EXDAMS can associate any value with the exact source statement that produced it.

This ability, and its inverse of associating any source statement with the values it produces, is fundamental to the EXDAMS philosophy of debugging and monitoring that the activity of a program may be viewed in either the data or the control spaces. The data space shows *which* manipulations a program performs, which values change, and the sequence in which they change. The control space demonstrates *how* a program performs its manipulation.

In a canonic debugging situation, according to the EXDAMS philosophy, the user first ascertains *what* is happening, then decides whether this behavior is correct, and finally, if it is not correct, determines *how* the program performed these operations, at the same time seeking the error in the program and/or data. Thus, any comprehensive debugging and monitoring system must include powerful facilities in both the data and the control spaces and provide a simple means of alternating between corresponding points in either space, as the user's needs or personal preferences dictate.

Flowback Analysis

By calling for FLOWBACK FOR and specifying a particular value, the user requests EXDAMS to analyze how information flowed through his program to produce the specified value. This analysis appears in the form of an inverted tree, with the bottom node corresponding to the value for which the flowback analysis was desired. Each node consists of the source-language assignment statement that produced the value, the value itself, and links to nodes at the next level. These nodes correspond to the non-constant values in the assignment statement displayed in the node that links with these nodes. These nodes have the same format as the original and are linked to nodes for all non-constant values used in the particular assignment statement producing their value. Thus, Fig. 1 shows a flowback analysis for a particular value of A.

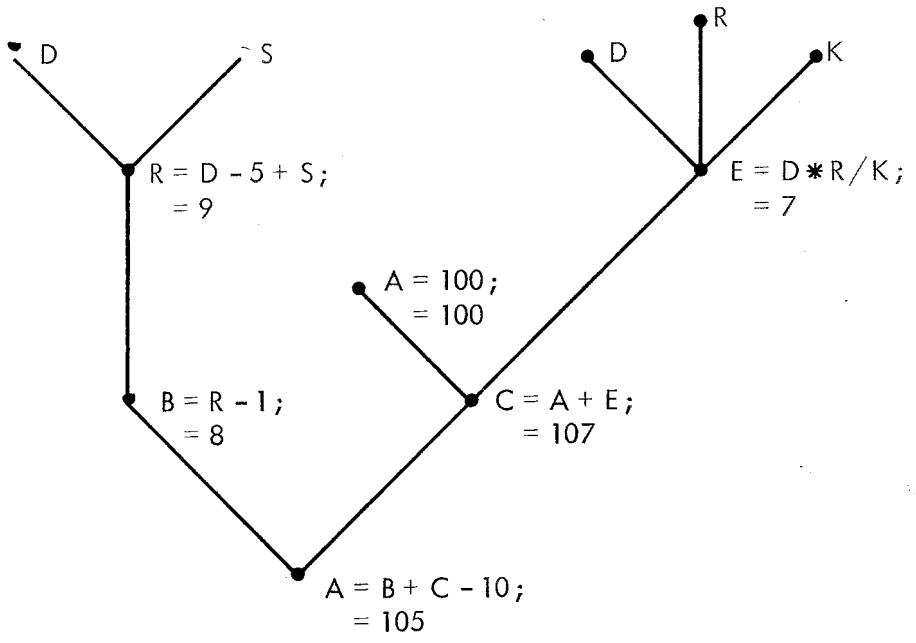

Fig. 1--Flowback Analysis

This display shows that the assignment statement "A=B+C-10;" produced the specified value of A, and its value here was 105. The values of B and C used in this assignment to A were 8 and 107, respectively, and were produced by the assignment statements "B=R-1;" and "C=A+E;", respectively. Each of the other nodes is explained in the same manner.

As many levels as will conveniently fit on the screen will be displayed. The user can request a similar flowback analysis along any particular branch. He can also call for the source code around any assignment statement in the flowback analysis and, as described in the section *Motion-Picture Displays* below, watch the execution either forwards or backwards from any point.

A similar type of flowback analysis is possible for the control space, which displays the flow of control through the program between any two points in execution time (i.e., between two nodes in the flowback analysis). In a non-parallel processing environment, this is simply a linear sequence, unless one wishes to indicate control sequences at a lower level (within a subroutine or do-group) as a closed loop out of the main flow of control.

MOTION-PICTURE DISPLAYS

In all the following examples, the information displayed is a function of execution time, whose rate of change the user may increase or decrease, stop, or reverse. Such control, together with the ability to alternate between different debugging and monitoring aids, enables him to discover and pinpoint the bugs in his program.

Values

This facility displays the values of the variables or labels specified by the user. Each specified variable

or label is assigned a contiguous set of columns on the display in which their values will appear. (The label values will be a checkmark indicating at what point in the execution the label was reached.) These values will be ordered according to execution time, so that a value produced earlier than another will appear higher on the screen (Fig. 2). This display can be scrolled up or down to show other values that can not fit on the screen at the same time. This scrolling alters execution time appropriately. The user can change the direction of scrolling (and execution time) or stop at any point. Once stopped, he may alter the list of variables on the screen and restart, or he may request the source code for a particular value displayed.

Source Code

This facility allows the user to watch his program statements execute either forwards or backwards. The statement being executed will appear brightened on the screen. If it is an assignment statement, the value of the assignment will also be displayed. If the instruction being executed is not on the screen, the portion of the program containing this instruction will be displayed. The user can command the system to follow subroutine calls and, as in the static display of source code, to display all levels.

Map

This facility is an extension of the source-code facility and is an adaptation of Stockham's work on flow analysis [3]. The user specifies nodes (labels) to be displayed. All code between these nodes may be considered a single macro statement, for the purposes of execution-time advancement. Thus, as the user varies the execution

```
ABC       R1          S          FILE       LABEL2

12

                      0

                                 FILE1

-10

                                 JOE

          '101'B
          '0'B

                      3001

1000

                                            ✓

                                 HAL

          '1'B

                                            ✓

1000
```

Fig. 2--Values Display

time, the node corresponding to the code being executed brightens and, as execution moves from one node to another, a displayed arrow indicates this shift. The length of time a node brightens is determined by either a common execution-time rate for *each* macro statement or by the execution-time rate for *all* statements executed within the macro statement.

The former display is most useful for following program execution while searching for a bug, while the latter is well-suited to monitoring applications in which the user is trying to determine how the program operates and where it spends most of its time. The EXDAMS environment--allowing the user to dynamically stop the display, expand some nodes into several separate nodes, collapse other nodes into a single node, and then continue or reverse direction--should greatly improve the usefulness of this display.

Windows

The current values of the variables specified appear in "windows" (i.e., areas on the display screen) as execution advances or reverses. If the value exceeds the size of the window, as much will be displayed as possible. In the case of arrays, the system will display in the window the value being changed and as many array elements, and their indices, around it as can fit. In addition, for either arrays or strings, certain variables can be specified as pointers or indices into these data representations.

The values of these variables appear in graphic, rather than numeric or alphanumeric, form according to the position of an arrow directed at the character or element at which the pointer or index is also directed. Thus, in a buffer application, where many buffers are scanned and processed and new buffers created, the user can watch the data in the buffers change dynamically, and see the pointers and indices move back and forth through the buffers.

Windows with Transitions

This facility performs the same operations as the preceding *Windows* facility except that, in addition, it indicates the interrelationships between the displayed variables. As each new value is displayed, a flowback analysis determines whether the current value of any displayed variable was used in the creation of the new value. If so, an arrow indicating this dependence appears, linking the windows of these variables to the window of the variable being changed. To obtain more detail for a particular transition indicated by the arrows, the user may define a new display relevant to that transition only, then either re-advance or reverse execution time. After completing the study of this particular transition, he may return to his original display.

IV. THE EXDAMS ENVIRONMENT

EXDAMS is a four-phase system predicated on the assumption that neither an incremental compiler nor a special debugging compiler designed for EXDAMS requirements would be available for the source language being debugged and/or monitored. If either is available, considerable restructuring of these phases would be prerequisite to the full utilization of these capabilities. The four phases are program analysis, compilation, run-time history-gathering, and debug-time history-playback.

PROGRAM ANALYSIS

The first phase analyzes the user's source program as it performs four functions, the most important of which is the creation of a model of that program. This model, the heart of the debug-time history-playback, is the means by which values gathered on the history tape are interpreted and by which portions of the source code are retrieved, and is the repository of all structural information known about the program. The use of the model for these functions will be explained in the section *Debug-Time History Playback*, pp. 15-16, but the contents of the model will be discussed here.

The program analysis produces both a symbol table and a random-access file of the user's source program for the history-playback. As it analyzes the program, it also builds a model of the program, and inserts debugging statements into the program to provide the information necessary for history-gathering. In general, the history contains all the dynamic information needed to update execution time either forwards or backwards, while the model contains all necessary static information.

Each model entry consists of an indicator of the type of model entry, a pointer to the associated source statement,

and an index to an entry in either the model or the symbol table, depending on the type of entry.†

The model contains both the static control-information and the variable alteration-information of the user's program. The control-information consists of the CALL, GOTO, IF-THEN-ELSE, and DO-END structure of the program, while the variable alteration-information consists of the names of the variables on the left-hand side of assignment statements and those altered by input statements.

The debugging statements added to the program pass the relevant run-time information to the run-time history-gathering routines.††

The updated program is passed to the compilation phase, while the symbol table and model are saved for the debug-time history-playback.

Compilation

The standard source-language processor compiles the source program, as updated during program analysis.

Run-Time History-Gathering

The compiled version of the updated program is run with a set of run-time routines that it calls. These routines gather dynamic information about the program's behavior. This information is collected in a buffer that is written out when full. It is the *history tape* of the program's behavior and, together with the symbol table and model, is sufficient to recreate the program's behavior in either the forwards or backwards direction of execution time. This history contains, basically, the values of the variables on the left-hand side of assignment statements, the direction

---

†Appendix A explains the use of the index field for each type of model entry.

††Appendix B details these statements.

(THEN or ELSE) taken in IF statements, the direction (remain in or flow out) taken at the end of DO-LOOPS, and the point from which a GOTO or CALL was issued (to facilitate execution-time backup).†

Debug-Time History-Playback

This phase contains the debugging and monitoring aids which present the history information to the user in a usable form on his display screen. It also interprets the user's commands for alternative displays and/or execution-time variations, and provides an editing capability for modifying discovered bugs and for returning this modified program to the four phases for another debugging iteration.

The main function of this phase is to assemble information from the history and display it on the screen. Appropriately, the main routine in the phase is the information retriever used by all the debugging and monitoring aids to retrieve desired information from the history. It accepts a) requests from the processing routines for information on a certain variable or set of variables and b) a direction for execution-time. Using this direction, it searches the history for the next occurrence of a value change for any variable in the requested set. It returns the name of this variable, its new (or old, if executing backwards) value, and its attribute.

Special calls facilitate the next subroutine call, goto, return, assignment, iteration, or conditional statement to be retrieved, so that all information in the history is retrievable through this routine. The calling routine describes what information to retrieve, and combines, processes, and formats it for the display routines that interact with the display equipment.

---

†Appendix C presents the precise information placed in the history.

The information retriever moves a marker through the model as values are read in from the history. This movement serves three purposes:

1) To permit interpretation of the bits in the history. Since the values in the history are not of a fixed length, knowledge of the type of the next value allows the routine to correctly interpret the value and position itself at the next value.
2) To associate the values in the history with statements in the source program (through the pointer to the source statement in the model), enabling users to alternate between values in the data space and the associated source statements in the control (program) space.
3) To reduce the amount of I/O necessary. By using the model to interpret values from the history, we need store only the value of source variables and not also the identification of the variables of which it is the value. This reduces the amount of I/O by roughly one-half; since the system is I/O-bound, this obviously improves the system's response.

V. THE ADDITION OF NEW DEBUGGING AND/OR MONITORING FACILITIES

To add a new debugging and/or monitoring facility to the EXDAMS system, first, extend the command language of EXDAMS to include the new commands needed to control the new facility and to route control to the new routine for these commands. As long as these commands do not conflict with existing ones, this is an easy task.

Second, obtain the information required to respond to the new commands by requesting it from the information retrieval routine as described in the previous section. This is the essential issue in the EXDAMS philosophy: All the information required by a routine can be obtained easily, by request, without interacting with the source program, the object code, or the history, but only with the information retrieval routine.

Third, process and combine the obtained information. The ease or difficulty depends entirely on the facility being added and is independent of the information collection mechanism.

Finally, format and display the processed information. Again, the effort required depends entirely on the facility being added and is independent of the monitoring mechanisms.

Thus, the EXDAMS environment reduces the problems of collecting information for a debugging and/or monitoring facility, but provides only minimal capabilities in the processing and presentation of this information. If the collection of information is a major problem in the creation of a debugging and/or monitoring facility, then EXDAMS has met its design goals. In addition, as we gain more experience in the types of processing and formatting required, we may also be able to provide capabilities that facilitate these areas.

VI. EXAMPLE

To illustrate the EXDAMS system, we present an example source program written in PL/I [4], followed by the major transformations performed on it by EXDAMS.

ORIGINAL SOURCE PROGRAM[†]

```
 1)  example_program:  PROCEDURE OPTIONS (MAIN);
 2)       DECLARE
 3)            a (10,3) CHARACTER (8) EXTERNAL,
 4)            i BINARY FIXED,
 5)            switch BIT (1),
 6)            search_string CHARACTER (8) VARYING;
 7)
 8)
 9)       GET FILE (input) LIST (switch, search_string);
10)       IF switch THEN
11)  loop:  DO i = 1 TO 10;
12)            DO j = 1 TO 3;
13)                 IF a(i,j) = search_string THEN DO;
14)                      PUT LIST (i, i<j);
15)                      CALL abc (i, i+j*3);
16)                      GO TO end_program;
17)                      END;
18)            END loop;
19)       ELSE
20)            PUT LIST ('switch turned off');
21)  end_program:
22)       i=j*i-5;
23)       RETURN;
24)       END example_program;
```

[†]The reserved keywords [4] of the source language are in all capital letters.

SYMBOL TABLE

The data are formatted here to facilitate reading, but this format does not reflect actual internal representation. The dummy entries (12 through 17) at end of the Symbol Table represent the types of expressions being passed to a subroutine or output.

Symbol Number	Name	Attributes	Model Entry Number
1	A	ARRAY(*,*),CHARACTER(8)	
2	ABC	PROCEDURE (*,*)	
3	END_PROGRAM	LABEL	26
4	EXAMPLE_PROGRAM	PROCEDURE	1
5	I	BINARY,FIXED	
6	INPUT	FILE,STREAM	
7	J	BINARY,FIXED	
8	LOOP	LABEL	6
9	SEARCH_STRING	CHARACTER(8),VARYING	
10	SWITCH	BIT(1)	
11	SYSPRINT	FILE,STREAM	
12	DUMMY	BINARY,FIXED	
13	DUMMY	DECIMAL,FIXED	
14	DUMMY	BINARY,FLOAT	
15	DUMMY	DECIMAL,FLOAT	
16	DUMMY	CHARACTER(*),VARYING	
17	DUMMY	BIT(*),VARYING	

MODEL

To facilitate the reader's interpretation, the pointer to the source code is represented here as a line number in the original program, and an explanation of the index field of the model entry has been added.

Model Entry Number	Entry Type	Source Code Pointer	Index to Model or Symbol Table	Explanation of Index
1	PROCEDURE	1	29	Index of associated END model entry.
2	GET	9	6	Index of Symbol Table of file associated with GET.
3	GET_ASSIGNMENT	9	10	Index of symbol receiving new value.
4	GET_ASSIGNMENT	9	9	Index of symbol receiving new value.
5	IF	10	22	Index of model entry for end of THEN clause.
6	LABEL	11	8	Index of label in Symbol Table.
7	ITERATIVE_DO	11	21	Index of model entry for associated END statement.
8	ITERATIVE_ASSIGNMENT	11	5	Index in Symbol Table of iteration variable.
9	ITERATIVE_DO	12	20	Index of model entry for associated END statement.
10	ITERATIVE_ASSIGNMENT	12	7	Index in Symbol Table of iteration variable.
11	IF	13	19	Index of model entry for end of THEN clause.
(Notice there is no entry for the non-iterative DO statement in line 13 of the source code.)				
12	PUT	14	11	Index of Symbol Table entry of file receiving new value.
13	PUT_ASSIGNMENT	14	5	Index in Symbol Table of first output value.
14	PUT_ASSIGNMENT	14	17	Index in Symbol Table of second output value. (This is a dummy entry for the attributes (bit) of the output expression.)

Model Entry Number	Entry Type	Source Code Pointer	Index to Model or Symbol Table	Explanation of Index
15	CALL	15	2	Index of label in Symbol Table.
16	CALL_PARAMETER	15	5	Index in Symbol Table of value being passed as first parameter.
17	CALL_PARAMETER	15	12	Index in Symbol Table of value being passed as second parameter. (This is a dummy entry in Symbol Table that represents the attributes of the expression being passed.)
18	GOTO	16	3	Index of label in Symbol Table.
19	SHORT_IF_END	17	11	Index of model entry of associated IF statement.
20	ITERATIVE_END	18	9	Index of model entry of associated iterative_do.
21	ITERATIVE_END	18	7	Index of model entry of associated iterative_do.
22	ELSE	19	25	Index of model entry of end of ELSE clause.
23	PUT	20	11	Index in Symbol Table of file receiving new value.
24	PUT_ASSIGNMENT	20	16	Index in Symbol Table of first output value. (This is a dummy entry for the attributes (character) of the output expression.)
25	FULL_IF_END	20	5	Index of model entry of associated IF statement.
26	LABEL	21	3	Index of label in symbol table.
27	ASSIGNMENT	22	5	Index in Symbol Table of variable left of assignment statement.
28	RETURN	23	4	Index of associated procedure label in symbol table.
29	PROCEDURE-END	24	1	Index of model entry of associated procedure statement.

AUGMENTED SOURCE PROGRAM

The altered or inserted source statements are italicized to facilitate their recognition.

```
example_program:  PROCEDURE OPTIONS (MAIN);
     DECLARE
          a (10,5) CHARACTER (8) EXTERNAL,
          i BINARY FIXED,
          switch BIT (1),
          search_string CHARACTER (8) VARYING;
     DECLARE condition_tester RETURNS (bit 1));
     GET FILE (input) LIST (switch, search_string);
     CALL bit_value (switch); /*record new value*/
     CALL character_value (search_string); /*record
       new value*/
     IF condition_tester (switch) THEN DO; /*record
       value of if condition*/
          CALL goto_issued_from (5); /*record index
            of model entry from which control passed
            to label*/
loop:
          CALL outside_do_loop; /*record control outside
            of do-loop*/
          DO i=1 to 10;
               CALL inside_do_loop; /*record control
                 inside of do-loop*/
               CALL binary_fixed_value (i); /*record
                 new value*/
               CALL outside_do_loop; /*record control
                 outside of do-loop*/
               DO j=1 TO 3;
               CALL inside_do_loop; /*record control
                 inside of do-loop*/
               CALL binary_fixed_value (j); /*record
                 new value*/
                    IF condition_tester (a(i,j)=search_string)
                      THEN DO; /*record value of if condition*/
                         PUT LIST (i,i<j);
                         CALL binary_fixed_value (i); /*record
                           output value*/
                         CALL bit_value (i<j); /*record output
                           value*/
                         CALL called_from (15); /*record index
                           of model entry of call statement*/
                         CALL binary_fixed_value (i); /*record
                           value of passed parameter*/
                         CALL binary_fixed_value (i+j*3);
                           /*record value of passed parameter*/
```

```
                    CALL abc (i,i+j*3);
                    CALL goto_issued_from (18); /*record
                      index of model entry of goto
                      statement*/
                    GOTO end_program;
                    CALL end_then_clause; /*record end
                      of then clause*/
                    CALL inside_do_loop; /*record control
                      at end of do loop*/
                    END;  /*explicitly end each iterative
                      do loop*/
                CALL outside_do_loop; /*record control
                  outside of do-loop*/
                CALL inside_do_loop; /*record control at
                  end of do-loop*/
                END loop;
            CALL outside_do_loop; /*record control outside
              of do-loop*/
            CALL end_then_clause; /*record end of then
              clause*/
            END; /*end non-iterative do group inserted
              after if statement*/
        ELSE DO; /*add do to enclose added statements
          within else clause*/
            PUT LIST ('switch turned off');
            CALL character_value ('switch turned off');
              /*record output value*/
            CALL end_else_clause; /*record end of else
              clause*/
            END; /*end non-iterative do group inserted
              after else statement*/
        CALL goto_issued_from (25); /* record index of
          model entry from which control passed to label*/
end_program:
        i=j*i-5;
        CALL binary_fixed_value (i); /*record new value*/
        CALL return_issued_from (28); /*record index of model
          entry of return statement*/
        RETURN;
        END example_program;
```

HISTORY FILE

We assume the input value for switch and search_string to be TRUE and 'XYZ', respectively. We further assume that the first element of array A that matches this string is A(2,1). The format of the values in the file facilitates reading. Comments appear on the right. The reader can start either at the end of the history and work backwards to the beginning of the program, or at the beginning of the history and work forward towards the end of the program.

Value	Comments
TRUE	Input value of SWITCH.
XYZ	Input value of SEARCH_STRING.
TRUE	Value of if-condition.
5	Index of model entry from which goto was issued.
OUTSIDE_DO_LOOP	Control is outside outer do-loop.
INSIDE_DO_LOOP	Control is inside outer do-loop.
1	Value for iteration variable I.
OUTSIDE_DO_LOOP	Control is outside inner do-loop.
INSIDE_DO_LOOP	Control is inside inner do-loop.
1	Value for iteration variable J.
FALSE	Value of if-condition.
INSIDE_DO_LOOP	Control is at end of inner do-loop.
INSIDE_DO_LOOP	Control is at beginning of inner do-loop.
2	Value for iteration variable J.
FALSE	Value of if-condition.
INSIDE_DO_LOOP	Control is at end of inner do-loop.
INSIDE_DO_LOOP	Control is at beginning of inner do-loop.
3	Value for iteration variable J.
FALSE	Value of if-condition.
INSIDE_DO_LOOP	Control is at end of inner do-loop.
OUTSIDE_DO_LOOP	Control is outside inner do-loop.
INSIDE_DO_LOOP	Control is at end of outer do-loop.
INSIDE_DO_LOOP	Control is at beginning of outer do-loop.
2	Value for iteration variable I.
OUTSIDE_DO_LOOP	Control is outside inner do-loop.
INSIDE_DO_LOOP	Control is inside inner do-loop.
1	Value for iteration variable J.
TRUE	Value of if-condition.

Value	Comments
2	Output value on file SYSPRINT.
FALSE	Output value on file SYSPRINT.
15	Index of model entry from which call was issued.
2	Value of parameter being passed.
5	Value of parameter being passed.
18	Index of model entry from which goto was issued.
-3	New value for I.
28	Index of model entry from which return was issued.

Appendix A

USE OF INDEX FIELD IN MODEL ENTRIES

Model Entry	Use of Index Field
PROCEDURE, BEGIN, ITERATIVE_DO, DO_WHILE	Index of associated END model entry.
END	Index of associated PROCEDURE BEGIN, ITERATIVE_DO, or DO_WHILE model entry.
IF	Index of associated ELSE (if this is an IF-THEN-ELSE statement) or SHORT_IF_END (if this is an IF-THEN statement) model entry.
ELSE	Index of associated FULL_IF_END model entry.
SHORT_IF_END, FULL_IF_END	Index of associated IF model entry.
CALL, FUNCTION_INVOCATION, GOTO, LABEL	Index in Symbol table of associated label.
RETURN	Index in Symbol table of associated Procedure label.
GET, PUT	Index in Symbol table of associated file.
ASSIGNMENT, GET_ASSIGNMENT, PUT_ASSIGNMENT, ITERATIVE_ASSIGNMENT, CALL_ARGUMENT	Index in Symbol table of associated variable (or dummy entry if an expression).

Appendix B

STATEMENTS ADDED TO USER PROGRAM TO PRODUCE EXDAMS HISTORY

In the additions described below it is assumed that DO_END brackets are placed around statements as necessary to preserve the semantics of the user program, e.g., when the THEN clause is expanded from one statement to two or more.

1. For each variable on the lefthand side of an assignment statement, each parameter in a function or procedure call, and each variable in an input or output statement, a call to the appropriate (as determined by the item's attributes) value saving routine, passing the item as the argument, is inserted after the source statement. In addition, for each parameter in a function or procedure call these same value saving calls are also inserted before the source statement.
2. For each IF statement the condition is replaced by a function call (which saves the value of the condition) with the condition as the argument of the function call.
3. At the end of a THEN clause a call is made to the END_THEN_CLAUSE routine.
4. Similarly at the end of an ELSE clause a call is made to the END_ELSE_CLAUSE routine.
5. Before a CALL statement, a RETURN statement, or a GOTO statement or the occurrence of a label a call to the CALL_ISSUED_FROM,RETURN_ISSUED_FROM, or GOTO_ISSUED_FROM routine (passing the entry number of the associated model entry as the argument) is inserted.
6. Before an ITERATIVE_DO or DO_WHILE statement a call to OUTSIDE_DO_LOOP is inserted. After the source statement a call to INSIDE_DO_LOOP is inserted.

In addition, if the source statement is an ITERATIVE_DO statement, a call to the appropriate value saving routine (passing the control variable as argument) is inserted after the call to INSIDE_DO_LOOP.

7. Before an END statement which is an end to an ITERATIVE_DO or DO_WHILE statement a call to INSIDE_DO_LOOP is inserted, and after this END statement a call to OUTSIDE_DO_LOOP is inserted. If the END statement specifies a label, it is replaced by the appropriate number of simple END statements before the above additions are made.

Appendix C

INFORMATION RECORDED IN HISTORY FOR EACH TYPE OF STATEMENT

Statement Type	Information Recorded in History
1. Assignment	Value of each variable on left hand side of assignment statement after the assignment is made.
2. IF	Value of if-condition.
3. end of then clause	Indication of end of then clause.
4. end of else clause	Indication of end of else clause.
5. Call, function invocation goto, and return statements	Index of model entry associated with source statement, and for return statement, index of model entry to which return is being made. In addition, the value of each argument in the call or function invocation is saved both before and after the call or function invocation.
6. iterative_do, do_while	Indication of the start and end of do_loop and two indicators signalling each iteration around the loop. Also, for iterative_do loops, the value of the control variable is saved each time around the loop.
7. input, output	Value of each variable (or expression) for which a value was input or output after the input or output operation.

REFERENCES

1. Corbato, F. J., and V. A. Vyssotsky, "Introduction and Overview of the Multics System," *AFIPS Conference Proceedings* (1965 FJCC), Vol. 27, Part I, Spartan Books, Washington, D.C., 1965, pp. 185-196.

2) Evans, Thomas G., and D. Lucille Darlay, "On-Line Debugging Techniques: A Survey," *AFIPS Conference Proceedings* (1966 FJCC), Vol. 29, Spartan Books, Washington, D.C., 1966, pp. 37-50.

3) Stockham, Thomas G., Jr., "Some Methods of Graphical Debugging," Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communications, held at the Thomas J. Watson Research Center, Yorktown Heights, New York, May 3-5, 1965.

4) *IBM Operating System/360 PL/I: Language Specification*, IBM Corporation, White Plains, New York, Form C28-6571-4, 1967.

5) Ferguson, H., and Elizabeth Berner, "Debugging Systems at the Source Language Level," *Comm. ACM*, Vol. 6, No. 8, August 1963, pp. 430-432.

6) Halpern, Mark, "Computer Programming: The Debugging Epoch Opens," *Computers and Automation*, Vol. 14, No. 11, November 1965, pp. 28-31.