A DEBUGGER FOR MODULA-2 ON THE UNIX PC

by

SRIDHAR ACHARYA

B.E. (Electrical Engg.), Allahabad University, India

M.Tech (Industrial Engg.), Indian Institute of Technology, India

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1987

Approved by:

Major Professor

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my advisor, Dr. Virgil Wallentine, for his valuable help, guidance, and encouragement during this work. I am also indebted to him for the time consuming reviews of this report and many helpful suggestions.

The implementation would never have seen the light of day without the invaluable assistance of Mr. Harvard Townsend. Harvard's support of my ideas, his keen insight into my problems, and above all his willingness to answer all my questions at all times, were crucial for this work, for which I am truly thankful.

My thanks are also extended to Dr. David Gustufson and Dr. Elizabeth Unger for serving as committee members, and to Dr. Thomas Pittman for helping me out with the low level details.

Last but not the least, I wish to thank all my friends for their moral support, good humor and encouragement during my graduate study.

ABSTRACT

This report describes the design and development of a debugger for Modula-2 on the UNIX PC. During the design phase several existing deb ıggers on various machines were surveyed with a view to identify their advantages and limitations. This was helpul in the design of the debugger.

The debugger is intended to provide a "user friendly" facility for undergraduate students debugging their class programs. The user interface ıs designed with menus and windows supported by the UNIX PC. Basic features for debugging such as, source file lookup, setting breakpoints, viewing the values of the variables and changing them if desired are provided in the debugger. An on-line menu driven help facility is included to provide the student with on-line help for the commands during a debugging session.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Need for a Debugger.

The presence of bugs in programs can be regarded as a fundamental phenomenon; a bug free program as an abstract theoretical concept like the absolute zero in thermodynamics that can be envisaged but never attained. Almost all computer programs at some stage of their existence contain errors resulting in unacceptable performance.

Debugging is the process of locating the causes of an error and fixing them. The existence of errors is detected through the discovery of their effects. Error detection and its cause is the primary domain of debugging.

Software development goes through different phases such as software requirements, analysis, detailed design, coding, testing and maintenance. Debugging comes under the coding phase and forms one of the important stages of the software development life cycle. Much attention is now being paid to debugging systems and their impact on software development. As the user's programs become increasingly larger and more complex the need for more sophisticated debugging tools increases.

1.2 What is a Debugger all about.

Whenever a program aborts, the first concern of the programmer is to locate the line in the source code at which the program failed. Subsequently he/she tries to locate the cause of the failure. The debugger, as the name suggests, guides the programmer through different steps that will enable him/her to identify the cause of the error and display the line in the source code where the error occurred. With the help of the debugger the com-

plete program can be analyzed in stages and corrective action can be taken to avoid abnormal termination.

The various methods used in fault localization can be broadly divided into two categories: the ability to monitor the system behavior by way of tracing and the ability to perform experiments in a controlled execution environment. In tracing, the program is allowed to run freely and information about the program state is written onto a debugging medium. During analysis these samples may provide insight into the cause of the misbehavior. Controlled execution techniques step the program through points believed to be significant in determining the problem. When one of the specific points(breakpoint) is reached, the program is stopped so that the user may analyze the state of the computation. This may include looking at the values of the variables, the sequence of procedure calls, etc. Also parts of the program may be perturbed in a controlled fashion to view its behavior under those circumstances.

Debugging is based on some implicit assumptions on the part of the programmer as well as on the program to be debugged. The programmer is assumed to know where to put the breakpoints and what data to analyze. Also the program should be consistent in coming out with the same results for a fixed set of input, on every run.

The above discussion concerns sequential programs. Concurrent programs are always more difficult to debug than sequential programs. This is because of the inherent complexity involved with time dependency of concurrent programs. A concurrent program consists of two or more processes executing in parallel. During their execution, these processes may interact with each other in an asynchronous fashion. A classical example of processes communicating asynchronously is the producer-consumer example, where the producer produces as fast as possible until it has to wait for the consumer to catch up with it. The

major difference between concurrent and sequential programs involves synchronization. Synchronization is the constraint imposed on autonomous processes within a concurrent program to achieve coordination between them, so that they don't interfere with each other.

Concurrent language debugging becomes more complex than that for sequential programs because besides the regular debugging facility for sequential programs it should also have a facility for individual process control, a communication monitoring facility, and process testing features. Process control should include the ability to create, suspend and destroy processes as well as the ability to obtain various process related information available in the system(such as current process state). The communication monitoring facility should be able to monitor message traffic and dynamically alter the contents of these messages. Process testing features should allow a user to isolate a process by controlling the flow of messages in and out of the process.

A powerful user interface is increasingly becoming a necessary component of all debuggers. The objective of having a good graphic interface to the debugger is to provide the programmer with a graphic picture of what the code looks like. For example if it has a linked list as part of its data structures, then the linked list is shown complete with links and the value of the fields. So instead of going through the character based information he is supplied with information in the form he perceived while designing the program. This greatly improves debugging. Further all graphic interfaces are genuinely user friendly complete with menus, windows,and an on-line help facility relieving the user of remembering all the cryptic debugging commands. Instead, he/she is able to walk through the complete debugging session with ease. Also with the help of multiple windows the different parts the user can view the source code in one and program execution in other thereby having a

good picture of how the source code corresponds to the execution. Thus we see that a good user interface enhances the capabilities of the debugger by catering to the needs of the programmer.

In this report the focus is on a design and further implementation for a debugger for Modula-2[WIRTH 83] on the UNIX PC[UNIX PC 87b], meeting the basic requirements of a user-friendly debugging facility. To aid the reader, here is how the rest of this report is organized. In chapter 2, a survey of debuggers is presented. A review of current research in this area is presented followed by a discussion on five existing debuggers. Chapter 3 covers the design aspects of the debugger under consideration. Chapter 4 presents the implementation issues involved in developing the debugger. The last chapter deals with future work in enhancing the debugger. In particular how to incorporate the concept of a knowledge base in debuggers is also covered.

Chapter 2

Survey of some existing debuggers

2.1 Evolution of Debuggers.

The earliest debuggers date back to 1950's when only storage dumps and output traces were used for debugging a program. Debuggers started to emerge in the early 60's and work accelerated in the 70's. With the advent of better hardware it was possible to make use of hardware interrupts to step through a program. The user was able to set breakpoints that would stop the execution of the program enabling the analysis of the current state of the computation. Symbolic debuggers came into use presenting the user with a friendly environment for debugging programs. A symbolic debugger interprets the symbols in the program, during a debugging session. In essence the user need not handle machine addresses instead, he/she can use the variables, constants, procedure names etc. defined in the program to access information via the debugger. The symbolic debugger makes use of the symbol table to extract information about each symbol.

Initially all debuggers were developed for sequential programs. Further debuggers were developed only after the development of the high level language, this imposed some constraint on the development of the debugger. Today, there are a several debuggers available for concurrent languages and recent debuggers are developed in parallel with the language. Special features have also been included in the system to facilitate debugging. One such feature is "ptrace" a system call in Unix System V. It is used for primitive level interprocess communication meant primarily for debuggers. This will be discussed in detail in a later section of this report.

Some recent research has as its goal the development of debuggers for distributed and concurrent environments because they pose many challenges that are absent in a sequential language. The focus of today's research is also towards the development of expert debuggers with the incorporation of knowledge base to aid debugging. This will provide the user with an added edge in debugging large and complex programs.

2.2 Survey of five debuggers.

The debuggers surveyed were: sdb[UNIX PC 85b] on AT/T 3b's[UNIX 85], dbx[UNIX 86b] on vax 11/780[UNIX 86], Modula-2[MODULA-2 86] debugger on Zenith 150's[ZENITH 84], MacPascal[MAC PASCAL 84] debugger and the C-Pascal[YOUNG 81] debugger. A brief description of each of them follows:

2.3.1 sdb.

Sdb is a source level symbolic debugger on Unix system V[UNIX 85]; it is available on all AT/T 3b machines. When debugging a core image from an aborted program it reports the line in the source program that caused the error. It allows all variables to be accessed symbolically and displayed in the correct format. The user can set breakpoints at selected statements or he/she can opt for single step execution. The source text can be listed according to the file name or at the point of current program execution. Individual procedures can be called directly from the debugger. This is useful in testing individual procedures and for calling user provided routines. sdb also supports machine language debugging. It is possible to print machine language statements associated with a line in the source code, and to put breakpoints at arbitrary addresses. The sdb program can also be used to display or change the contents of the machine registers.

Sdb is a symbolic debugger with approx. 12000 lines of source code. It supports

almost everything that is useful in debugging sequential programs. The command structure of sdb is not user friendly; it is cryptic in nature. The user must be very familiar with the Unix manuals to utilize the full capabilities of the debugger.

2.3.2 dbx.

Dbx is also a source level debugger running on UNIX 4.3 BSD operating system. It also has the basic features of a symbolic debugger like tracing, breakpointing, single stepping etc.Unlike the cryptic sdb commands, dbx uses English verbs like "run" , "trace" etc., as its commands. It has an on-line help facility which the user can invoke inside the debugger; thus the programmer does not have to exit the debugger to look at the on-line manual. Dbx also permits aliasing for shortening of its command expression.

2.3.3 Mac Pascal debugger

Mac Pascal debugger is an on-line debugger for the Mac Pascal interpreter. The debugger window is active all the time and can be called at any point of execution in the program. The Mac Pascal debugger supports multiple windows. It has one for the text that shows the source code, one for the data needed for any interactive data input and one for observing the changing values of the variables. Breakpoints can be set by moving the stop icon to the source code line number. Single step execution is also available as a menu option. The current line being executed is shown by a "hand" icon with a finger pointing to that particular line number. Although the user interface is powerful in the Mac Pascal debugger, it is limited in it's capabilities. The user cannot look into multiple files that are not currently being executed. Also facilities like invoking separate procedures within the debugger and displaying the variables in different formats that are available in debuggers like sdb and dbx are absent. Mac Pascal debugger also lacks an on-line help facility. Since there is no on-line help available the user is not able to utilize the capabilities of the

debugger completely.

2.3.4 Pascal 32

Pascal 32 is the symbolic debugger for sequential and concurrent Pascal. It has most of the capabilities of sdb and dbx except for source file lookup. It also has features for debugging concurrent programs. The "SUMMARIZE" command produces a summary list of the status of all active processes. The summary consists of the process PCB address, the current process state, the current program counter, the global base, the local base values etc. Another command "TRACEBACK" provides a traceback, listing all of the active processes in active stacks starting from the main process.

2.3.5 Modula-2 debugger on Zenith 150's

Modula-2 debugger by Modula Corp. works on Zenith 150's, and is a source level symbolic debugger. As soon as the debugger is invoked, it divides the screen into six windows viz. Source window, Procedure Chain window, Module List window, Data window, Memory window and the Dialog window. These windows are static and cannot be changed or closed by the user. Whenever the program is executed inside the debugger all the windows vanish and the program runs on the whole screen and when the program ends or aborts the windows come back. Thus the user is not able to see the result of his execution and the source code simultaneously. In the absence of a mouse, the mouse emulation via the keyboard becomes cumbersome. Also since there is no on-line help facility the user needs to be heavily dependent on the manual before he/she works with the debugger.

2.3 Review of Related Literature.

Current debugging research has been concentrated towards debuggers for a distributed or a concurrent processing environment, with experimentation on more powerful

debugging interfaces. Concurrent or distributed processing pose many challenges for debugger development and has generated a lot of interest in the research community. The following section gives a brief overview of some of the recent developments in this area.

CBUG[GAIT 85] is one of the source level debuggers developed for concurrent programs. It is a portable tool running under UNIX Operating system. Some of the important features of CBUG are:

1. Conditional Breakpointing: Here breakpoints can be set depending on the value of a variable or a location dependent predicate.
2. Process Monitoring: Process invocations can be monitored from a concurrent set along with their arguments.
3. Tracing: Tracing of interprocess activities within a concurrent set can be done at synchronization points.

During debugging sessions, the user can dynamically look at synchronization points, parallel actions, and deadlock situations. The user interface is provided with the help of multiple windows, each process running in one window.

DEFENCE[WEBER 83] is a debugger for Concurrent Euclid running in a time shared uniprocessor environment. The debugger makes use of the language's concurrency feature. The code generated by the CE compiler is interfaced with calls to a synchronization kernel that supports processes, monitors and 'signal' and 'wait' statements. Process monitoring in DEFENCE is implemented by accessing information from this synchronization kernel. Besides the regular features DEFENCE has conditional breakpointing and process monitoring. Through process monitoring the user can get information on the currently active process, the queues in which the remaining processes are stored and the current execution line

in each process body. The basic features of tracing and execution in steps is provided by modifying the assembly code generated by the CE compiler. Calls to debugging routines are inserted at appropriate places in the assembly code and the debugging routines determine what action needs to be taken.

[GARCIA 85] uses Petri Nets to model event interaction in a concurrent language.The Petri Net lends itself easily to concurrent processing, and is used here to develop a monitoring facility during concurrent program execution. A 'replay' tool to ease tracing is also presented here. This is based on a premise that an activity may be too fast for real time observation. The 'replay' tool provides facilities like "slow motion" , "still frame" and "frame advance" which allow the programmer to view interesting portions of the event at a convenient pace. The model also shows that Petri Nets can be used effectively to control the flow of synchronization. For example in a debugging session, tokens could be moved manually to reassign resources among processes.

[GARCIA 84] describes the key issues involved in debugging distributed computing systems. Several debugging tools and aids are also presented. Multiple asynchronous processes communicating via shared variables with multiple loci of control are difficult to monitor and are more prone to bugs. Multiple processors make it difficult to detect erroneous program behavior and to manage the debugging program. A solution model is presented that has different debugging modules running at different nodes and communicating with a master debugger at a master node. The programmer, through the master debugger controls the debugging of the distributed system. The desired debugging facility is provided by the integration of the different debugging modules and coordinating their actions.

The [GARCIA 84] also present a tool for examining traces. The trace files form a distributed database. The data is viewed as a single relation with different attributes like event type, process id, timestamp etc. This relation can be queried using a query language such as SEQUEL and the programmer can also perform operations on the query to pull out specific information. Other tools for controlled execution and process monitoring are also discussed.

IDD[HARTER 85] is an Interactive Distributed Debugger for debugging distributed programs. It provides a assertion language for automatically monitoring the flow of the program. The program stops when an assertion fails. The user also has the facility of expanding the given set of assertions dynamically. Besides tracing and message monitoring IDD also provides assertion monitoring. In a distributed environment it can relieve the programmer of the burden of examining large traces and can also help in catching some time dependent errors by allowing automatic monitoring of states and message sequences. IDD also provides the user with a strong graphic interface complete with multiple windows and menus. It has been implemented on a Sun workstation running Unix 4.2 BSD.

[BATES 82] applies the concept of Behavioral Abstraction to debugging. Here the system is viewed in terms of its activity rather than its state. The abstractions are expressed in terms of higher level event occurrences and are used by the debugging monitor for comparison with the actual behavior of the system. While dealing with the complexities inherent in a distributed system the user needs a high level viewpoint to derive a meaningful comparison between actual system activity and the conceptualized system behavior. The above model provides that.

VIPS[ISODA 87] is a visual debugger; it attempts to bring out the full potential of a graphic interface for the benefit of the user. It uses graphics to show the static and the

dynamic behavior of program execution thus freeing the programmer of analyzing character based information. With multiple windows the programmer is able to track the interaction of different parts of the program as well as view the source code for it. This helps the programmer to detect and localize program errors more easily than the traditional debugging tools. VIPS has been implemented on a PERQ[ISODA 87] workstation with a high resolution bit mapped display.

Chapter 3

Design aspects of a debugger for Modula-2 on UNIX PC

3.1 COFF: The Common Object File Format.

COFF (Common Object File Format) is the output file produced on some UNIX systems by the assembler and the link editor. This format is also used by the other operating systems, hence the word common is both descriptive and widely recognized. Currently this object file is used for the AT/T UNIX PC, AT/T 3b family and on the Vax 11/780 and 11/750 [UNIX 86] running on UNIX operating systems. COFF is important to the debugger since the debugger derives all the needed information from the COFF. Some of the key features of COFF are listed here.

1. Applications may add system dependent information to the object file without causing access utilities to become obsolete.
2. Space is provided for symbolic information used by the debuggers and other applications.
3. Users may make some special modifications in the object file construction at compile time.

As shown in Fig 1. an object file contains:

1. a file header.
2. optional header information.
3. a table of section headers.
4. data corresponding to section headers.

5. relocation information.

6. line number information.

7. a symbol table.

8. a string table.

The last three sections, -the line number information, the symbol table and the string table section- are the important ones to the debugger. The file header section contains information to access these sections. For the line number information to be present the program has to be compiled with a "-d" option. The string table is also absent if the source file does not contain any symbol names with names longer than eight characters.

The file header contains the file pointer to the starting address of the symbol table, the number of entries in the symbol table, the number of bytes in the optional header and the flags that describe the nature of the executable file.

When invoked with a "-d" option the C compiler generates an entry into the object file for every C language source line where a breakpoint can be inserted. These line numbers can then be referenced inside the debugger. All line numbers are grouped by the function name as shown in Fig 2. The first entry in a function grouping has line number 0 and has in place of the virtual address an index into the symbol table for the entry containing the function name. Further entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries appear in the increasing order of addresses.

The symbols inside the symbol table appear in the sequence as shown in Fig 3. They are organized according to the file name. All local symbols are grouped according to their function names. The symbols defined in storage class "static" are put after all the symbols

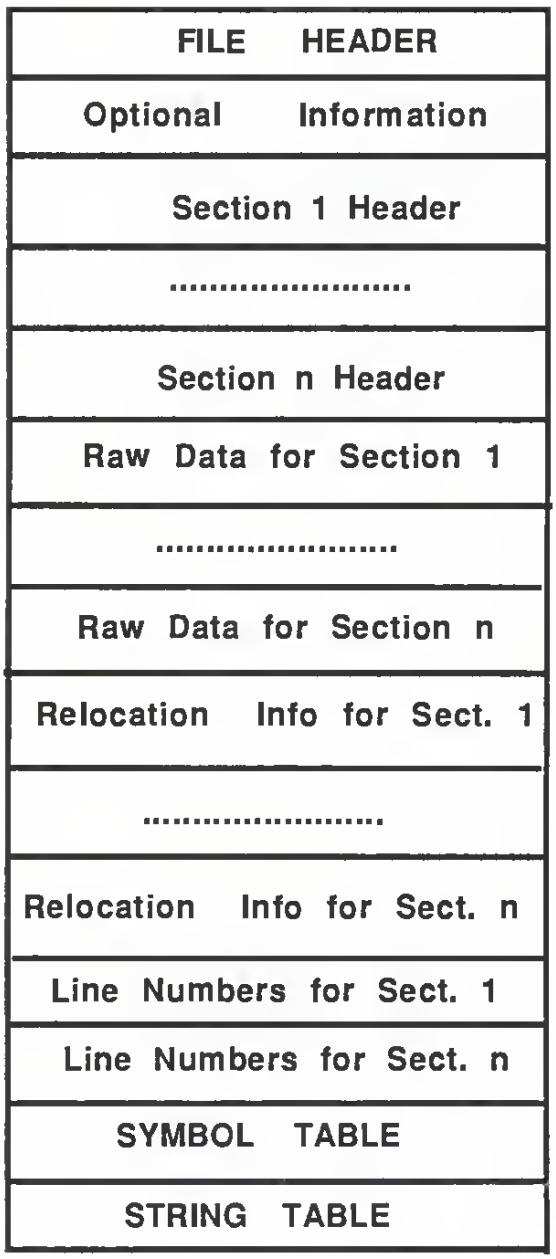


FIG 1. COMMON OBJECT FILE FORMAT

in the functions have been allocated. The symbol table consists of at least one fixed length entry per symbol with some symbols followed by the auxiliary entry of the same size. The entry for each symbol is a structure that holds the value type and other information needed by the debugger.

The first eight bytes in the symbol table entry is a union of a character array and two long integers. If the symbol name is eight characters or less the symbol name is stored in the character array. If the symbol name is longer than eight characters than the entire symbol name is stored in the string table that follows the symbol table entry in COFF (Fig 1.). In this case the eight bytes contain two long integers, the first is zero, and the second contains the offset(relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first four bytes serve to distinguish a symbol table entry with an offset into the string table, from a symbol name in the first eight bytes.

Currently there is only one auxiliary entry per symbol. The size of the auxiliary entry is the same as that of the symbol table. However unlike the symbol table entry the format of the auxiliary table entry for a symbol depends on its type and storage class.

3.2 Design of the debugger.

The primary objective of developing the debugger for Modula-2 on the UNIX PC were

1. to provide a robust and a user friendly debugging environment for the user on the UNIX PC.
2. to provide basic debugging features such as source file lookup, setting breakpoints, view the value of the variables and changing the value of the variables.

symbol Index	0
virtual address	line number
virtual address	line number
....	.....
symbol Index	0
virtual address	line number
virtual address	line number

FIG 2. LINE NUMBER GROUPING
INSIDE COFF.

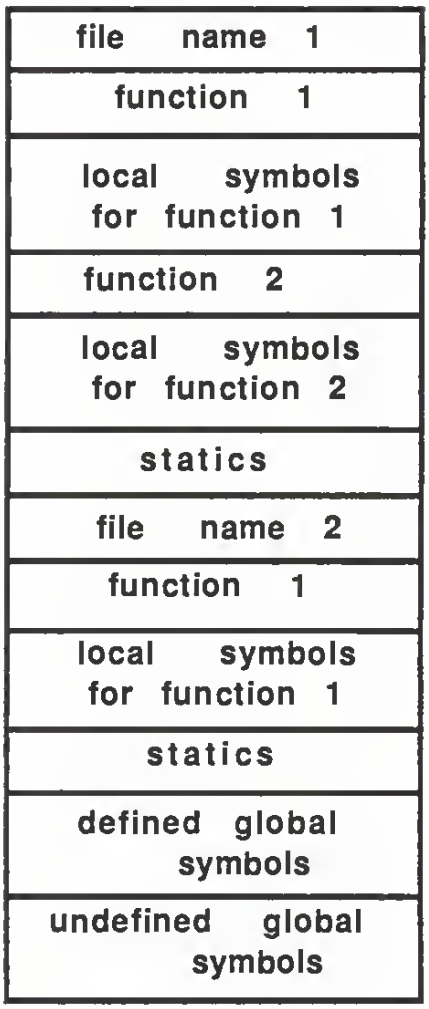


FIG 3. COFF GLOBAL SYMBOL TABLE

and

3. to provide an on-line help facility for the dehugging commands.

The need for a user friendly environment was given high priority since the facility will mostly be used by the undergraduate students dehugging their class projects. Since the UNIX PC has a mouse/window interface, it can meet the requirements of the graphic interface needed for the debugger.

The hasic debugging features mentioned ahove would enahle the user to find the location of the error, look at the source program for that error, set breakpoints, and view and change values of the variahles. These facilities are considered sufficient to debug a normal class program. Future enhancements on the debugger could be undertaken to increase the capability of the debugger.

The presence of an on-line help facility would vastly reduce the number of trips to the manual racks by the students. Furthermore it would encourage students to use the dehugger more often.

With the facility for "source file lookup" the user could view the source program along with their line numbers. The lookup facility serves two purposes. One when the program fails the user is able to look at the line numbers ir the program where it failed and by looking at the source program may be able to detect the cause of the error. The other purpose is to help the user set breakpoints by enabling him to look at the complete file with the line numbers. For setting breakpoints the user is prompted for the procedure name and the line numher in the procedure. It was decided to have the source in a separate window with a complete scrolling facility using all the three mouse buttons. The first hutton, when pressed would advance the text by three lines, the second button would

advance the text by eight lines and the third would scroll directly to the end of the file. The same would be true for reverse scrolling. In this way the user could quickly scan through the whole file.

Whenever a debugged program is executed inside the debugger the debugger process forks of a child. The child process executes the debugged program via a "exec" call. The parent(debugger) process traces the execution of the child process and waits for the child process to stop or finish. When the child process stops or finishes it wakes up the parent process that is sleeping on a "wait" system call via a signal(Fig 4.). The parent process decodes that signal and takes further action. It could continue or terminate the child process depending on what is required.

The virtual address of each line where the breakpoint can be set is given in the line number information inside COFF. They are arranged by each function. The line numbers are relative to the function with each function starting at line zero. To set breakpoints the line in the source file has to be correlated with the line number information available in COFF. Since the symbol table has information on the beginning of each function in the program, the relative line number inside the function could be matched with the line number in the source program. Breakpoint setting here has been restricted to line boundaries, meaning breakpoints cannot be set at any other address except at line numbers in the source program. This constraint has been imposed because if a breakpoint could be set at any address then the address needs to be disassembled to find out whether or not it is on an instruction boundary. Since at present there is no disassembler on the UNIX PC the address checking for instruction boundary is impossible.

Once the program stops at a breakpoint the user can look at the values of local variables inside the currently active procedure or any of the global variables. He can also

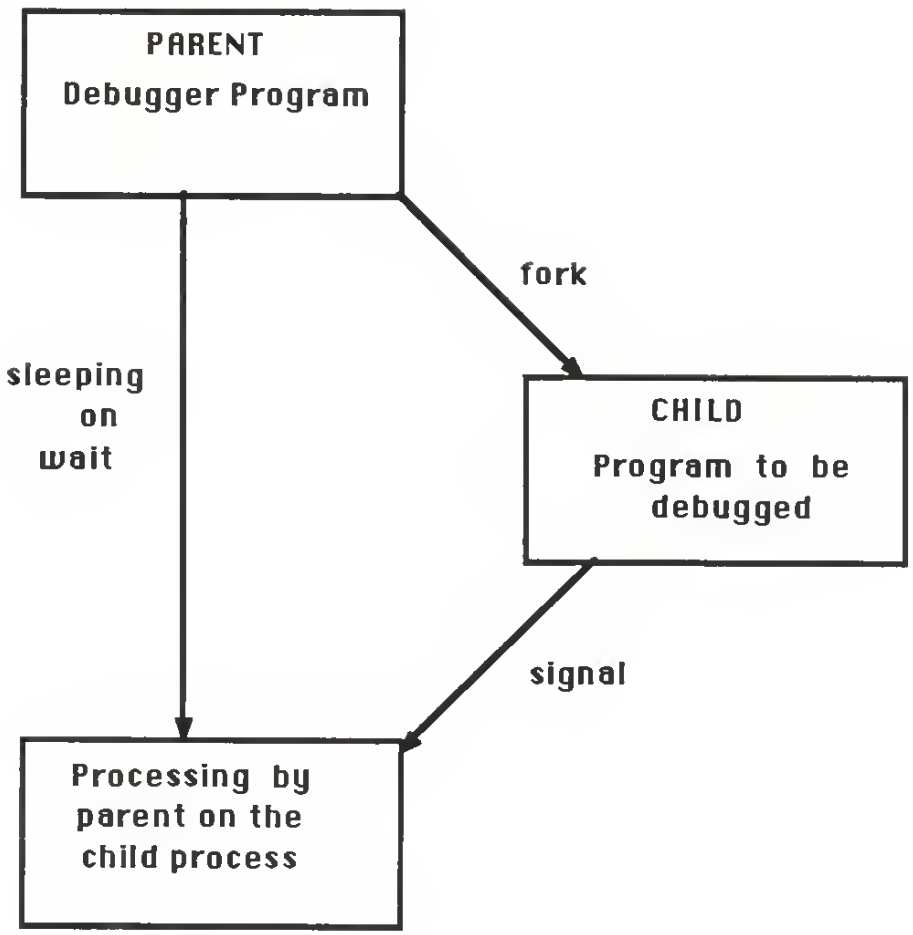


FIG 4. INTERPROCESS COMMUNICATION
BETWEEN THE PARENT AND THE CHILD PROCESS.

change the values of these variables, set more breakpoints or continue execution. To do any of the above mentioned actions the programmer needs only to select the main menu by clicking on it if it is visible or by selecting it via the window manager. The user is now ready to choose any of the actions in the main menu by moving the highlighted cursor on the choice and clicking on it.

Accessing the local variables in an active procedure when the program is at a breakpoint is done via traversing the stack frame till we arrive at the correct procedure. The type and the offset of the variable is obtained from the symbol table and the values of the variables inside the procedure is obtained from the particular stack frame. If the variable is an array or a structure then we need to get all the index values or the values of the various fields as the case may be. The "type" field in the symbol table is used to get the required information about indexes, fields etc. Once the variables are found in the stack frame, their values can be easily changed.

3.3 "Overall approach" to debugging.

The intent here is to provide the user with a global view towards debugging his program. With the help of multiple windows the user can look at the program execution on one, interrogate the source program on the other and view the value of the variables on the third one. This enables the user to get an overall picture of the program and aids in understanding it and debugging it in an efficient fashion.

The menus and windows in the user interface are similar to the Mac Pascal debugger. However, unlike it the menus are not static, the submenus disappear whenever they are not required and are redrawn whenever required. Only the main menu stays all the time and the user is able to activate it at any time by just clicking on it or by selecting it via the window manager. This debugger is a improvement over the Modula-2 debugger on the

from Modula-2 Corp. The windows here are not static, the user can change them as he wishes. Also, since the execution of the program takes place on a new window, the other windows do not disappear as they do on the Modula-2 debugger. The source file lookup has a special feature called "file by procedure name" . This feature is not available in any of the debuggers that have been surveyed so far. Here the user can view the appropriate source file by just giving the procedure name. This is specially useful when a large program is being debugged and there are many linked files.

Multiple windows allow the user to get a global view of program debugging. But if there many open windows then it effects the system performance. As each window is a process in UNIX, if there are many processes running at the same time the general system performance comes down. Hence the user is advised to keep only the ones that he requires and close all the others.

Chapter 4

Implementation Issues in developing the debugger

4.1 Development of the User Interface.

The user interface on a UNIX PC consists of menus, forms and windows accessible via the mouse or the keyboard. A menu gives the user a way to view a list of available items and allows the user to select one or more items from the list, for an action. Forms are used for command entry when a command contains a user specifiable option. A form is presented to the user for filling in the specific options. Forms are also used to display the properties of the objects in the menu. Windows are used for running any application programs. Several windows can be open simultaneously for display, but only one window the active one receives input from the keyboard. The border of an active window is highlighted while the border of all inactive ones are dark.

All open windows are manipulated by the window manager. The window manager displays a pop up menu of windows to which the user can point to and issue window shaping and movement commands. The window manager window is displayed when user presses the "Suspnd" key on the keyboard to suspend the active window. The user can suspend the current window implicitly by moving the mouse pointer outside the current window and pressing the mouse button. The window that becomes active depends on the location to which the mouse is pointing when the button is pressed.

The menus are drawn via a "menu" system call. Every time the user clicks on the menu to activate it, the menu has to be redrawn to accept any input from it. The UNIX PC supports TAM[UNIX PC 87c] (Terminal Access Method) utilities that provide an interface to the UNIX PC kernel windowing capabilities. Through TAM we can create,

manipulate and get information on the windows.

As stated earlier multiple windows can reside simultaneously on the UNIX PC screen. These windows can be owned by the same process if it created them, or by distinct processes. The covered portions of any windows are remembered on the pixel basis my the kernel. The kernel maintains an ordering based on which windows overlap which other windows. The topmost window in this ordering is always completely visible.

For an application program to respond to changes in it's window and all the windows created by it, it must catch the signal SIGWIND. SIWIND is ignored by default. Thus for the debugger process to respond intelligently to any changes in it's windows, it must catch the SIGWIND signal and issue a "wgetsel" call that returns the currently selected window. A "wgetstat" call after that determines how that window has changed. This information includes the position and dimensions of the window, whether on or off and whether or not variable width characters are allowed. Any time the mouse is clicked on the resize or the move icon, the kernel traps it and sends the signal SIGWIND to the controlling process. The pseudo code for catching the SIGWIND signal is given in Fig. 5.

A "wgetc" call gets a single character from the standard input on the specified window. If a mouse button is pressed "wgetc" returns a unique code to suggest that a mouse report follows. The mouse report itself can be read using the "wreadmouse" call. The pressing of the mouse button while the mouse pointer is on any of the border icons except "Move" and "Resize" is translated by the kernel into a corresponding keystroke. Any action on "Move" and "Resize" icons is trapped by the kernel and the signal SIGWIND is issued. For "scroll up" and "scroll down" icons the keycode returned depends on the mouse button pressed. When the user points the mouse pointer at the "help" icon and presses the button then a "message" system call is executed. The message call has, as one of its arguments the

```
#include <sys/signal.h>
#include <tam.h>

main ()
{

 void handler();

 signal (SIGWIND, handler);  /* catch SIGWIND  */

 ..............
 .. ..............

 }

handler()
{

 int  wtemp;
 struct wstat *sts;   /* pointer to wstat structure */

 signal(SIGWIND, handler);  /* reset the signal again */

 wtemp = wgetsel();       /* get the current window
                                                  selected   */

 wgetstat(wtemp,sts);   /* determine how the window
                              has changed.       */
 ...............
  ...............

}
```

FIG 5. Code to handle SIGWIND.

help file to be displayed. The contents of the "help file" are displayed on a window created by the "message" call. The message is automatically wrapped to fit within the dimensions of the window, and may contain embedded newlines. "Message" then waits for user input and returns the character read to the caller. The help file is built according to the syntax given by "uahelp" which is the user agent help process. Infact when HELP is selected, "message" executes "uahelp", passing it the help file name and the initial help display title. "Message" then waits for "uahelp" to return. "uahelp" arbitrarily limits help files to 100 distinct displays, and each display is limited to 100 lines.

4.2 Process tracing - A key for implementing debuggers in System V.

The UNIX System V provides a primitive level of interprocess communication for tracing processes; this is useful for debugging. A debugger process spawns a process to be traced with the "ptrace" system call, setting and clearing breakpoints, and reading and writing data in its virtual address space. Process tracing thus consists of synchronization of the debugger process and the traced process, and controlling the execution of the traced process. The pseudo code in Fig.6 shows the typical structure of the debugger program. The debugger spawns a child process, which invokes the "ptrace" system call and as a result the kernel sets a trace bit in the child's process table entry. The child now "execs" the program being traced. The kernel executes the "exec" call as usual but at the end finds that the trace bit is set for the child process and sends the child a SIGTRAP signal. The kernel checks for signals while returning from the "exec" system call, just as it checks for signals after any system call, finds the SIGTRAP signal it had just sent itself and executes code for process tracing as a special case for handling signals. As the trace bit is set in the child's process table entry, the child awakens the parent in its sleep in the wait system call, enters a special trace state similar to the sleep state, and does a context switch.

```
main()
{

  int  pid;

  if (( pid  =  fork() ) == 0 )
  {
    /* child - traced process   */

    ptrace(0,0,0,0)         /* set the tracing bit  */

    exec("name of the traced process here" );

  }


  /* debugger process continues here    */
  for ( ;; )
  {

     wait((int *) 0);   /* parent waiting for the
                                  child to signal */

     read(input for tracing instructions );

     ptrace(cmd,pid, ........);

     if (quitting trace)
        break;

   }

}
```

FIG 6. Structure of Debugging Process.

Typically the parent (debugger) process would have meanwhile entered a user level loop, waiting to be awakened by the traced process. When the traced process awakens the debugger, the debugger returns from "wait" , reads the user input commands, and converts them to a series of "ptrace" calls to control the child (traced) process. The syntax of the "ptrace" system call is

ptrace(request,pid,addr,data);

The request argument determines the precise action to be taken by "ptrace", and has legal values between zero to nine. The request zero sets the trace bit in the child, indicating that it wants to be traced by the parent. Requests one and two are used for reading a word from the instruction or data space, three is used to read a word form the child's USER area, four and five are used to write a word onto the child's instruction or data space, six is used to write onto the child's USER space, seven is for the resumption of the child, eight is for the termination of the child and nine sets the trace bit in the Processor Status Word of the child that causes an interrupt to occur on completion of one machine instruction. The "addr" argument is the virtual address to be read or written in the child process, and "data" is the integer value to be written.

When executing the "ptrace" system call the kernel verifies that the debugger has a child whose "ID" is "pid" and that the child is in a traced state; it then uses a global trace data structure to prevent other tracing processes from overwriting it, copies "request", "addr", and "data" into the data structure wakes up the child process and puts it into a "ready-to-run" state. It then sleeps until the child responds. When the child resumes execution (in kernel mode), it does the appropriate trace command, writes its reply into a trace data structure, then awakens the debugger. Depending on the request type the child

may reenter the trace state and wait for a new command or return from handling signals and resume execution. When the debugger resumes execution, the kernel saves the "return value" supplied by the traced process, unlocks the trace data structure, and returns to the user.

If the debugger process is not sleeping in the wait system call when the child enters the trace state; it will not discover its traced child unt l it calls "wait". At that time it returns immediately and proceeds as described above.

The use of "ptrace" for process tracing is primitive and has several drawbacks.

1. Here the kernel must do four context switches to transfer a word of data between a debugger and a traced process. It switches context in the debugger in "ptrace" call until the traced process replies to a query, switches context to and from the traced process, and switches context back to the debugger process with an answer to the "ptrace" call. This overhead is necessary because a debugger has no other way to gain access to the virtual address space of a traced process. This makes process tracing slow.

2. A debugger can only trace child processes. If the traced child process forks another process, the debugger process has no control over the grandchild, a severe handicap when debugging complex programs. If a traced process does an "exec", the later "exec ed" images are still being traced because of the original "ptrace"; but the debugger may not know the name of the "exec'ed" process, thus making symbolic debugging difficult.

3. A debugger cannot trace a process that is already executing if the debugged process had not called "ptrace" to let the kernel know that it wants to be traced. This is inconvenient because a process that needs debugging must be killed and restarted in

the trace mode.

4. It is impossible to trace "setuid" programs because users could violate security by writing their address space via "ptrace" and doing illegal operations. For example if a "setuid" program that sets the "uid" to "root" is being traced, then a clever user could overwrite the instruction space of the traced process to execute instructions with unauthorized permission. "Exec" ignores the "setuid" bit if the process is traced to prevent the user from overwriting the address space of a "setuid" program.

[Killian 84] describes a different scheme for process tracing, based on file system switching. An administrator mounts a file system "/proc" , each member of which "/proc/nnn" corresponds to the address space of a running process whose pid is "nnn". Access to these files is restricted via the normal file protection mechanism to the process owner. Users can examine the process's address space by reading the file, and can set breakpoints by writing to the file. "Stat" returns the various statistics about the process. The process's address space is read only if it is a shared segment; the data and stack segments are read/write in any case; and the "user area" is read only except for locations corresponding to the saved user registers. As with the special files there are several services available via an "ioctl" call that includes sending and receiving signals, resuming execution, setting trace bits, etc..

This method removes three disadvantages of "ptrace" . First it is faster because a debugger process can transfer more data per system call than it can with "ptrace" . Second a debugger can trace arbitrary processes, not necessarily a child process. Finally, the traced process does not have to make prior arrangement to allow tracing, a debugger can trace an existing process.

4.3 Implementing the basic debugging features.

For implementing the "Source File Lookup" feature the symbol table is read into a global data structure namely the "proctable" which has fields as shown in Fig 7. It is a structure consisting of the procedure name, its virtual address, the line number of the procedure in the source file, its offset in the symbol table, pointer to a file structure that contains the file name, a boolean flag, virtual address, offset in a.out and the file pointer if the file is open. As the symbol table is arranged by file names with all the functions in that file grouped together below it(Fig. 3), the "proctable" structure is filled in as the symbol table is read at the start of the debugging process. The display of the particular procedure in the "source lookup" window is done as follows.

The user chooses the "source by procedure name" option in the "source lookup" menu. He/she is then prompted for the procedure name. Once the user enters the procedure name he wants to see, then the "proctable" data structure is traversed for the particular procedure name. After finding the desired procedure name in the list, its line number and file_name values are extracted. The source window is then displayed with the procedure in that file. When the program stops with an abnormal termination the user might want to look at the line number in the source program where the program failed. He/she then chooses the "current file" option in the "source lookup" menu. At the point of program failure the value of the program counter is read as an offset from the register "R0" inside the "user" data structure of the child process. This program counter value is correlated with the virtual address of the line number inside COFF. As mentioned in the earlier section the line number information inside COFF is arranged by each procedure, with each procedure starting at line zero. Matching the value of the program counter with the virtual address of line numbers gives the procedure name and the relative line number inside the

```
struct proctable {

        char    pname[80];  /* procedure name  */

        long    paddress;   /* address in a.out  */

        long    stf_offset; /* offset in a.out   */

        ushort  lineno;   /*  line no in source file */

        struct  filetable  *filptr  /* pointer to the
                                       filetable structure  */

        } ;


struct filetable {

        char   lfilename[14] ;    /* name of the file */

        long   faddr;             /* address in a.out  */

        long   stf_offset ;       /* offset in a.out  */

        int    open ;      /*  whether file is already open
                               or not   */
```

FIG 7. Structure of "proctable" and "filetable".

procedure where the error occurred. Now we need to get the actual line number within the source program from this relative line number. Once again the "proctable" structure is traversed for the particular procedure name giving the line number for the procedure in the source program. This line number when added to the relative line number gives the actual line in the source program where the program failed. The file at that line is then displayed in the "source" window. Same is true when the user wants to look at the source file when the program is stopped at a breakpoint.

For inserting breakpoints the user is prompted for the procedure name and the line number to set the breakpoint. Again this line number is correlated with its relative line number within the procedure. Then the virtual address for that line number is obtained from the line number information inside COFF(Fig. 2). Next the "ptrace" system call is used with the request argument set to 3, to get the instruction at the virtual address at which the breakpoint needs to be set. This instruction is stored in the breakpoint data structure, to be used later. In the place of this instruction an illegal instruction is inserted using "ptrace" with the request argument set to "4". When the program executes, it finds an illegal instruction at the breakpoint and stops. At this point the user can look at the value of the variables, change them if desired, examine the source program or continue execution. Continuing from a breakpoint is a little tricky. The regular instruction that was extracted and stored in the breakpoint data structure, is executed as a single instruction using "ptrace" with the request argument set to "9". Then "ptrace" is again used to continue execution in a normal fashion from the next program counter. All the regular instructions that are extracted for setting breakpoints are stored in the breakpoint data structure. When the user wants to delete all the breakpoints then the regular instructions are inserted back again using "ptrace".

To look at the values of the local variables the user gives the procedure name and the name of the variable inside the procedure. The "stack frame" is then traversed for that procedure. Each stack frame has an argument pointer, a fra ne pointer, the return address of the procedure, and space for the local variables in that procedure. Once the correct stack frame for that procedure is located, the symbol table is searched for that local variable. Since it is a local variable, the value field for it in the symbol table contains the stack offset in bytes. This offset added to the start of the stack frame for that procedure gives the location of that variable inside the stack frame. That value can be read using "ptrace" with the request argument set to "4". If the local variable is an array or a structure then all the index values or the field values can be read by advancing the address in "ptrace" by the required number of bytes. Further new values can be inserted in place of old ones by superimposing the new value over the old one, using "ptrace".

To look at the value of the global variable is relatively easy, since it can be extracted from the data segment. The virtual address of the global variable is read from the symbol table. The global symbols (static) in the symbol table are located after all the procedure symbols for that file. Once the virtual address is obtained for the global variable, "ptrace" is used with request argument set to "2" to read the value of that variable. To change its value "ptrace" is used with the request argument set to "5".

4.4 Sample User Session.

A simple debugging scenario is created for a sample user session. The buggy program shown in Fig 8. has a common "range error" with array limits exceeding its normal specified bounds.

The user starts the debugger session by typing "m2db test" inside the "UNIX System" window, where "test" is the object file for the buggy program. A main "debugger

commands" menu pops up, as shown in Fig 9. The user then chooses the "run" command under the main menu to run the program inside the debugger. The program is run under a new window called "Run", and when it crashes the debugger comes out with the line number where it failed (Fig. 10). The user may then want to look at the source code where the program failed, so he/she clicks back on the main menu to make it active, chooses the "Show Source" option from it and also chooses the "Current file" option from the "Show Source" submenu. The source code appears in a separate window with the file name on top of it (Fig. 11). He/she can then use all the three mouse buttons to scroll up or down the file by clicking on the scroll icons on the right hand corner of the window. The user can also get a bigger window size, with the help of the "Resize" icon or he can move the window to a different place with the help of the "Move" icon. Once the user looks at the source code, he/she might want to look at the value of the looping variable "i" in this case. He/she then clicks on the main menu to make it active again, chooses the "Show variable" option from it and on a prompt gives the name of the variable for which the value is being sought. The current value of the variable is then displayed in a separate window (Fig. 12).

Although the program was simple and the bug trivial, it shows what a user would typically go through in a debugging session. Also if at any time the user wants to look at the on-line help facility for that command, he/she just needs to click on the "Help" icon on the right hand corner of the menu/window and the help description pops up.

```
MODULE test;

FROM  InOut  IMPORT WriteLn, WriteInt;

VAR

   array :  ARRAY[1..10] OF INTEGER;
   i       : INTEGER;


BEGIN

     FOR  i:= 1 TO 20 DO
      array[i] := i;
     END;

END test.
```

FIG 8. A SIMPLE BUGGY PROGRAM

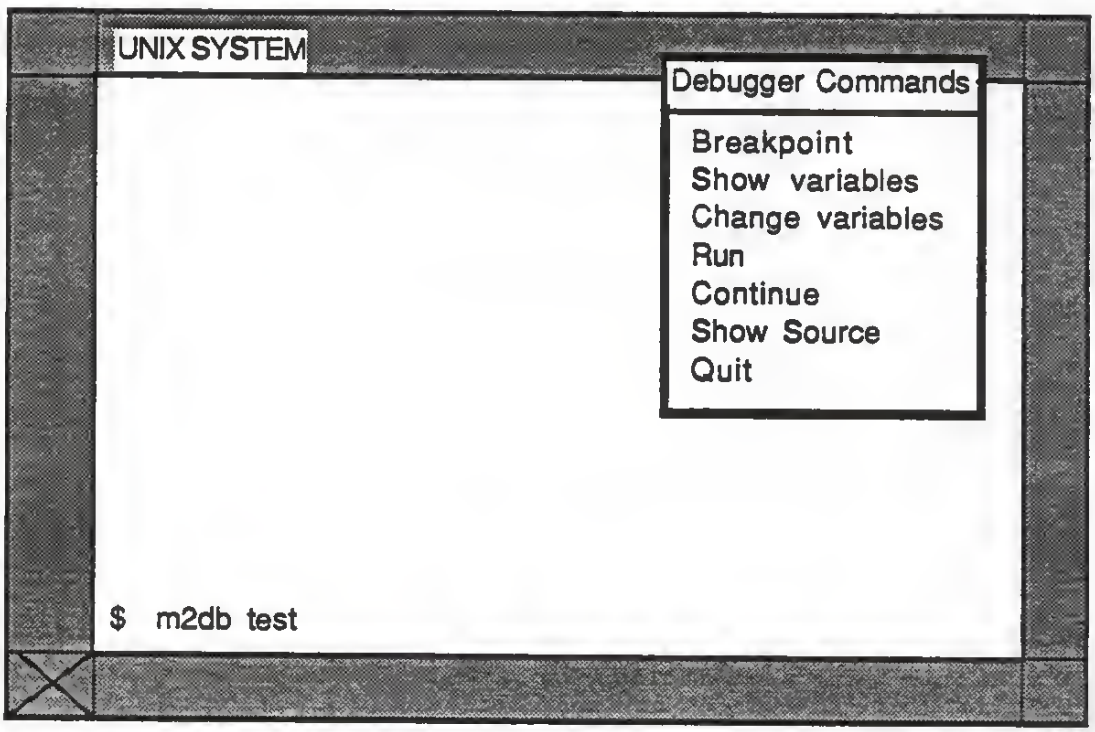


FIG 9. OPENING SCREEN FOR USER SESSION

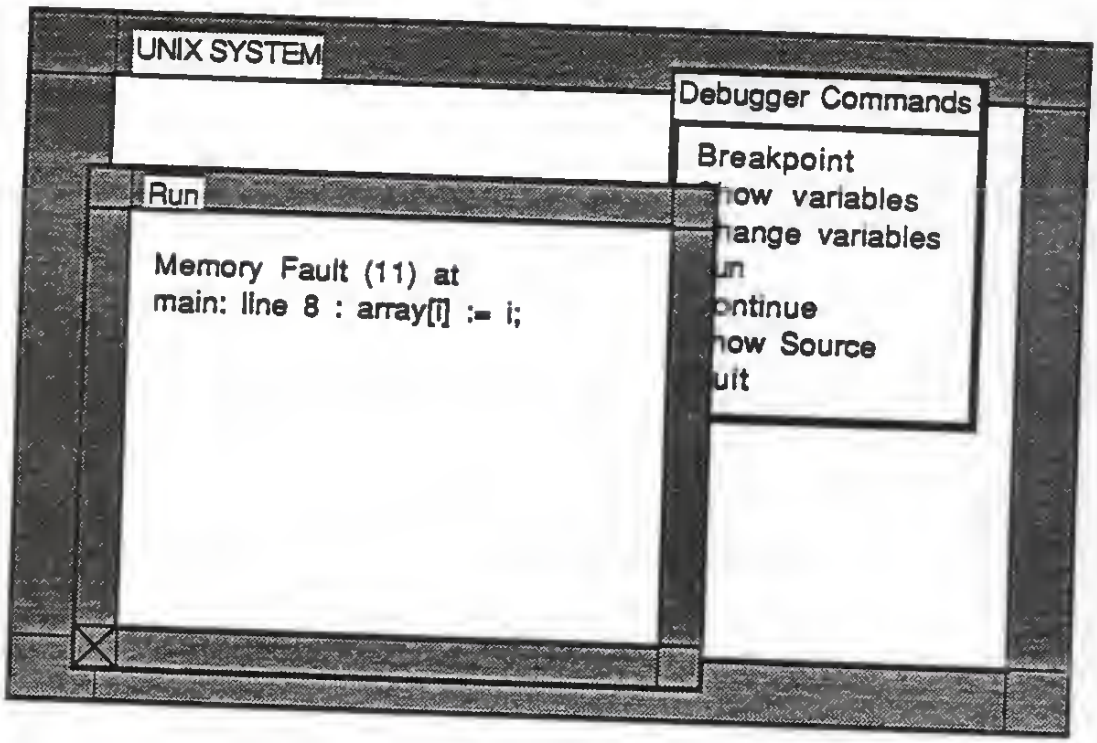


FIG 10. USER SESSION - SCREEN NO. 2

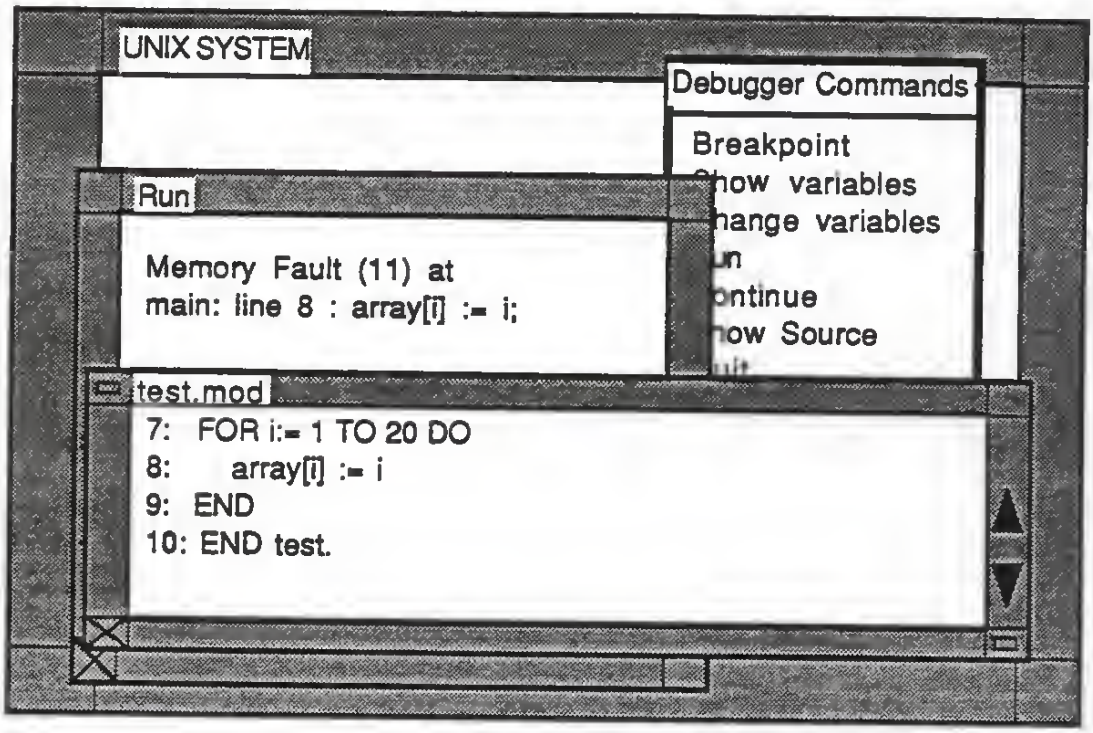


FIG 11. USER SESSION - SCREEN NO. 3

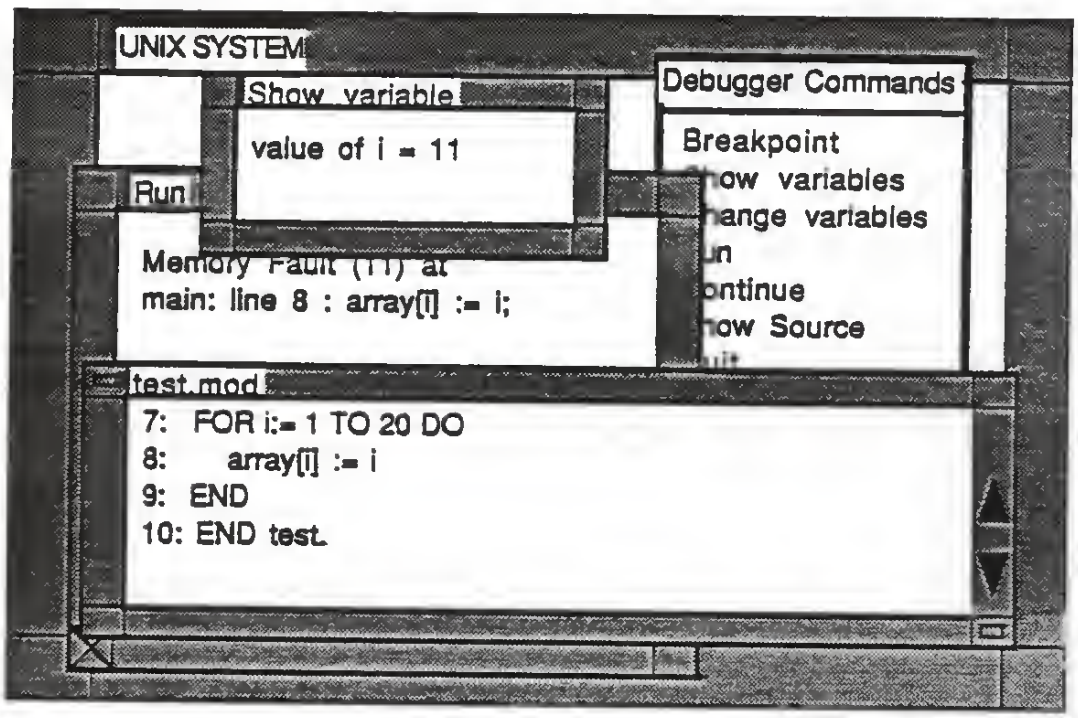


FIG 12. USER SESSION SCREEN NO. 4

Chapter 5

Future Work and Conclusions

5.1 Adding additional features to the debugger.

The capability of the existing debugging facility can be enhanced by adding additional features to it. Some of suggested improvements are

1. single step execution,
2. viewing the variables governed by the scope rules of the language,
3. separate execution of procedures,
4. machine level debugging, and
5. Setting breakpoint at any address etc.

Single step execution is a special case of execution under breakpoints. Here the breakpoint is set at every executable line in the source program. The program stops at every line and the user can closely observe its behavior. Single step execution involves much overhead by the debugger program since breakpoints have to be inserted at every line. Some of the overhead can be avoided by using "single step" for a limited number of lines in the source code.

Viewing the variables generated by the scope rules of the language involves programming the scope rules inside the debugger. This would allow the user to list all the variables with their values, which are visible in that scope. Thus the user would be able to look at values of all visible variables with one command.

With the facility for execution of each procedure separately, the user can test each

procedure individually for its correctness before complete integrated testing. This is specially helpful when dealing with a program consisting of many procedures spread over different files.

Machine level debugging is useful when low level information is required for debugging. With this feature the user can look at the assembly level instructions, the content of various registers, the address of variables etc.

The ability to set breakpoints at any address would require the debugger to disassemble the address to see whether or not it is on a instruction boundary. Hence a disassembler is essential for this feature. It would add more flexibility for debugging by allowing the user to stop a program at any address that is on an instruction boundary.

5.2 Developing a knowledge base environment for effective debugging.

Traditional debugging methods emphasize techniques that apply at the level of computation units and generally allow users to examine and alter the state of computation. Interactive debugging constitutes an important part of the traditional methods. It provides the user with a facility for examining the entire snapshot of that system state at any step of a computation. The programmer can perturb parts of the system in a controlled fashion to verify that each of them functions in a meaningful way.

All these methods work with a small unit at a time and hence can provide only a low level interpretation, which is adequate if we are dealing with simple programs. When we come across complex software involving several communicating processes(although we do not have concurrent processes in Modula-2, coroutines can be used to simulate them) we need a higher level abstract viewpoint that would form the knowledge necessary for making meaningful comparison between actual system activity and the ideal, conceptualized

system behavior.

5.2.1. Concept of Behavioral Abstraction.

The concept of Behavioral Abstraction[BATES 83] allows a programmer to view a system in terms of its activity rather than its state. It can be viewed as observing the system behavior by the sequence of event occurrences which are generated as the program progresses.

In Modula-2 the events can be defined as follows:

1. creation of the process,
2. transferring of control from one coroutine/process to another,
3. changing of global variables,
4. a process blocked on a wait, or a process released on send

   etc..

The programmer can specify these events with constraints which form the expected system behavior. These can be in the form of rules/predicates in YAPS[ALLEN 83] (Yet Another Production System), Prolog[CLOCKSIN 81] or any other language in which the rules/predicates can be suitably expressed. When the program crashes, the actual system behavior would have been recorded in a database. YAPS can take over and do the necessary matching between the actual behavior and the expected one and prompt the user about any discrepancy.

5.2.2 Using YAPS to represent knowledge.

YAPS is a rule-based language and is a member of the family of languages based on the production system model. The inference engine in YAPS consists of algorithms for matching rules, selecting rules and executing rules supporting forward chaining. The

knowledge is represented as rules consisting of a sequence of conditional elements and variables that form the left hand side of the production rule. The right hand side of the rule is composed of a sequence of actions which take place on the firing of the rule. The rule fires when the conditions on the left hand side are matched with the facts in the facts database. YAPS also has what is known as a facts database where all the facts needed to run the production rule are stored. The facts can be added or deleted from the database dynamically as desired. The idea is similar to an augmented Petri Net where the transition fires when the set of constraints associated with it are satisfied.

In our case, the facts database would include the event strings which are added to the database, as the events occur during the actual execution of the program. A set of production rules represent the knowledge of the expected system behavior. The structure of the knowledge base would depend on the granularity of the event description.

A typical scenario would look like this. The program, while running in the debugging mode, would build the facts database by dumping all the event messages into it. When the program finally finishes execution or gets aborted then YAPS would run and prompt the user on the event(s) which did not take place as desired (Fig.13).

5.3 Conclusion.

The Modula-2 debugger whose design and implementation are the subject of this report, can be used to successfully debug normal class programs. Additional capabilities if added, would enable it to handle a wide variety of software bugs. The "user-interface" provides the user with an "overall view" of the debugging session as well as the global picture of the behavior of the program. This is helpful since it leads to better understanding of the program and effective debugging. The on-line menu-driven help facility would be useful to the user. It would allow him/her to browse through the "command-usage" even

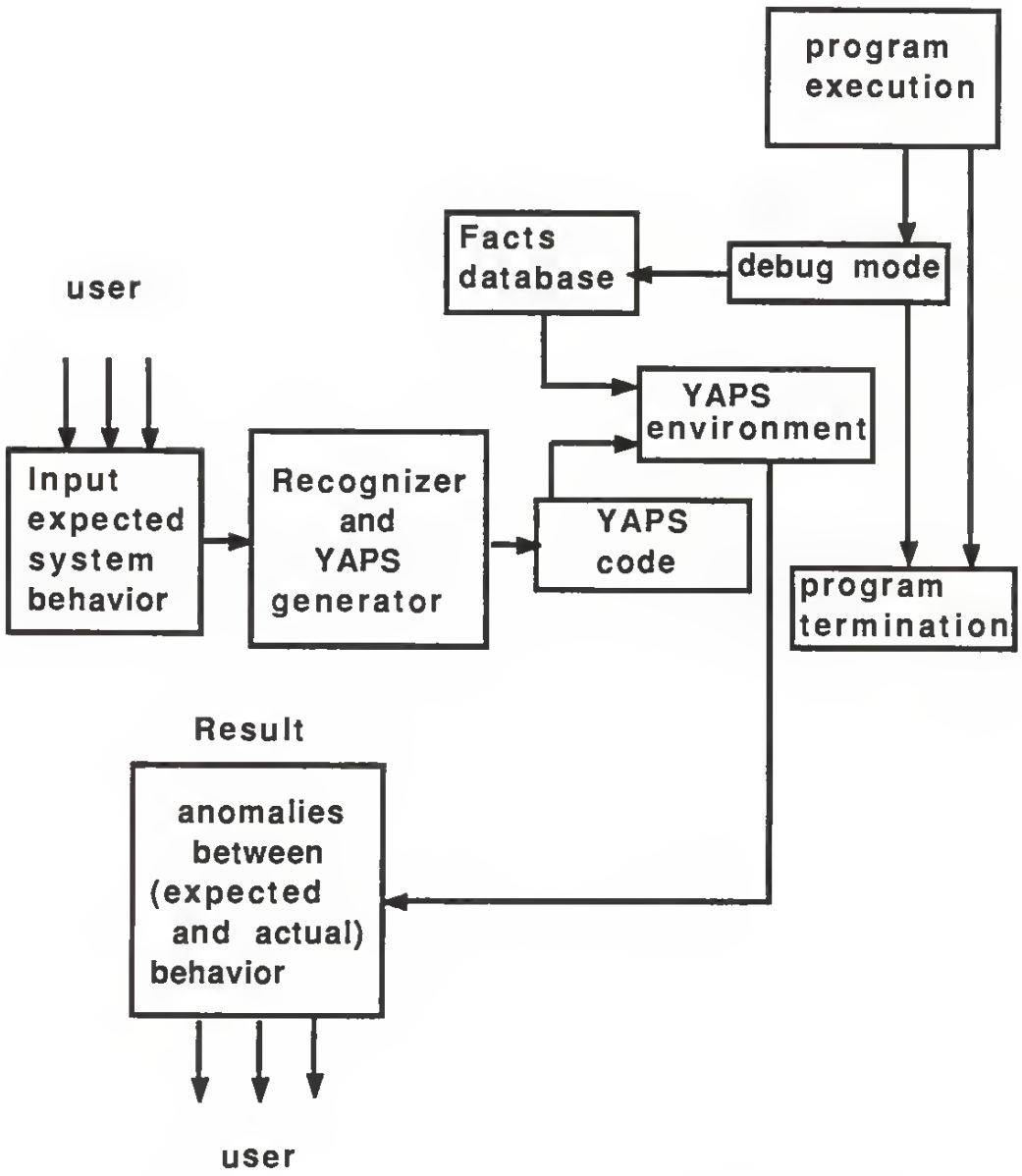


FIG 13. Schematic Overview of the YAPS System

in the middle of the debugging session.

Debuggers will continue to play an important role in software development and testing. In subsequent years we will see sophisticated debuggers with a high level graphic capability that aid the handling of complex software bugs. This will make the world of software development an easier place to work than it is today.

BIBLIOGRAPHY

[AHO 79] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1979.

[ALLEN 83] Elizabeth M. Ellen, "YAPS: Yet Another Production System," *Department of Computer Science, University of MaryLand*, December 1983.

[BAIRDI 86] Fabrizio Baiardi, Niciletta De Francesco and Gigliola vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering*, Vol SE-12, No. 4, April 1986, pp. 547-553.

[BACH 86] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, Inc., 1986.

[BATES 82] Peter C. Bates, Jack C. Wileden and Victor R. Lesser, "A Debugging Tool for distributed Systems," *University of Massachusetts COINS Technical Report*, December 1982, pp. 1-17.

[BATES 83] Peter C. Bates and Jack C. Wileden "An App oach to High-Level Debugging of Distributed Systems," *Communications of the ACM*, Marcl. 1983, pp. 107-111.

[BROWNSTON 85] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin, *Programming in Expert Systems in OPS5*, Addidon-Wesley Publication Company Inc., 1985

[BRUCE 80] Robert C. Bruce, *Software Debugging for MicroComputers*, Reston Publishing Companty, Inc. 1980.

[CLOCKSIN 81] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[C-PROLOG 86] David Warren, *C-Prolog User's Manual*, University of Edinburgh, 1986.

[GAIT 85] Jason Gait, "A Debugger for Concurrent Programs," *Software Practice and Experience*, Vol. 15, No. 6, June 1985, pp. 539-554.

[GARCIA 84] Hector Garcia and Frank Germano JR., "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, March 1984 pp. 210-219.

[GARCIA 85] Miguel E. Garcia and Joseph W..Ierman, "An Approach to Concurrent Systems Debugging," *Proccedings of the Fifth Intnl.Conf. Distributed Computing Systems, IEEE Computer Society, Denver, CO.* May 1985, pp. 507-514.

[GERMAN 82] Steven M. German, David P. Helmbold and David C. Luckham, "Monotoring of Deadlocks in Ada Tasking," *Communications of the ACM*, July 1982, pp. 10-25.

[GERMAN 84] Steven M. German, "Monitoring for Deadlock and.Ilocking in Ada Tasking," *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 6, November 1984, pp. 764-777.

[HARTER 85] Paul K. Harter, JR., Dennis M. Heimbigner and Roger King, "IDD: An Interactive Distributed Debugger," *Proccedings of the Fifth Intnl.Conf. Distributed Computing Systems, IEEE Computer Society, Denver, CO.* May 1985, pp. 498-506.

[HOARE 85] C.A.R. Hoare, "Communicating Sequential Processes," *Prentice Hall International*, 1985.

[ISODA 87] Sadahiro Isoda, Takao Shimomura and Yuji Ono, "VIPS: A Visual Debugger," *IEEE SOFTWARE*, May 1987, pp. 8-19.

[LEDOUX 85] Carol H. LeDoux and D. Scott Parker, JR., "Saving Traces for ADA debugging, Proceedings," *Proceedings of the ADA International Conference*, Paris, May 1985, pp. 97-108.

[KILLIAN 84] Killian J. Thomas, "Processes as files," *Proceedings of the USENIX conference, Summer 1984*, pp. 203-207.

[MAC PASCAL 84] A Pascal Interpreter for Apple Macintosh, *System Reference manual, 1985*.

[MODULA-2 86] Native Code Modual-2, *System Reference Manual*, Modula Corporation, 1986.

[MOTOROLA 82] MC68000, "16.Iit Microprocessor User's Manual," *Prentice Hall, Inc.*, 1982.

[NAGANO 83] Hironobu Nagano, Shuetsu Hanata, Muneo Takahashi and Tetsuro Mikami, "Automated Debugging Method Using Data Checking Specifications," *Journal of Information Processing*, Vol. 6, No. 3, 1983, pp. 156-162.

[STANKOVIC 80] John A. Stankovic, "Debugging Commands for a Distributed processing System," *IEEE COMPCON, Washington D.C.*, September 1980, pp. 701-705.

[SEDLMEYER 83] Robert L. Sedlmeyer, William.I. Thompson and Paul E. Johnson, "Knowledge-based Fault localization in Debugging." *Commnunications of the ACM*, January 1983.

LP [RUSTIN 71] Randall Rustin, "Debugging Techniques in Large Systems," *Courant Computer Science Symposium 1*, Prentice Hall, 1971.

[SMITH 81] Edward T. Smith, "Debugging Techniques for Communicating, Loosely-Coupled Processes." *Phd Dessertation, Dept. of Computer Science, University of Rochester*, Rochester, 1981.

[YOUNG 81] Robert A. Young, "Program Logic Manual for the Pascal Symbolic Debugger," *Department of Computer Science, Kansas State University*, March 1, 1981.

[UNIX PC 87a] AT&T UNIX PC "UNIX System V Programmers Guide," *AT&T System Programmaing Series*, 1987.

[UNIX PC 87b] AT&T UNIX PC "UNIX System V Users Manual," *AT&T System Programmaing Series*, 1987.

[UNIX PC 87c] AT&T UNIX PC "Interface Specification," *AT&T System Programmaing Series*, 1987.

[UNIX 85] UNIX System V. *User Reference Manual, AT&T 3b5 Computer*, 1985.

[UNIX 85b] UNIX System V *Programming Guide, AT&T 3b5 Computer*, 1985.

[UNIX 86] UNIX 4.3.ISD, *UNIX System Reference Manual, 4.3.ISD Virtual VAX - 11 Version*, 1986.

[UNIX 86b] UNIX 4.3.ISD, *UNIX User's Manual, 4.3.ISD, Virtual VAX - 11 Version*, 1986.

[W.IER 83] Janice C. Weber, "Interactive Debugging of Concurrent Programs," *Communications of the ACM*, March 1983, pp. 112-114.

[WIRTH 83] Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, NewYork, 1983.

[ZENITH 84] ZENITH-150 PC. *Z-150 PC Series Operations Manual*, Zenith Data Systems, 1984.

APPENDIX - User manual

Name : "m2db" - Modula-2 debugger on the UNIX PC.

Usage : m2db [object file]

"m2db" is an interactive, source level, symbolic debugger that can be used for Modula-2 programs on the Unix PC. It may be used to examine their object file and to provide a controlled environment for their execution.

The "object file" is normally an executable program file which has been compiled with a "-d" (debug) option. If the object file is not compiled with a "-d", flag the symbolic capabilities of "m2db" will be limited, but still the program can be debugged. Compiling with a "-d" option enables the line number information to be included in the object file.

Names of the variables and procedures can be used as they are in the source program.

A1.1 Some General Instructions.

To make any window/menu active click on it, that will activate it. Clicking involves moving the mouse pointer on that menu/window and pressing the mouse button. If any menu or window is not visible then you can click on the "window manager" icon (a "W" enclosed in a small box) on the top right hand corner of the screen (Fig. 14), or press the "Rsume" key on the keyboard. This will pop up the window manager window and any window/menu can be selected by clicking on it's name inside the window. Clicking on the "cancel" icon (a "X" enclosed in a box) in a menu/window will close it. The "cancel" icon is located at the bottom left hand corner of the menu/window (Fig. 14). If either the "Move" (two arrows pointing outwards enclosed in a box) or "Resize" (a small box and two arrows

enclosed in a box) icon are present they can be used to move the menu/window or change its size respectively. "Move" icon is located on the top left hand corner of the menu/window (Fig.14), while the "Resize" icon is located at the bottom right hand corner of the menu/window. Both "Move" and "Resize" are used by "dragging" it. "dragging" involves pointing the mouse pointer on the icon, pressing the left mouse button, changing the location or the size of the menu/window and then releasing the mouse button. The mouse button remains pressed during the complete operation. If the "Help" (a "?" enclosed in a small box) icon (Fig.14) is present then the on-line help facility can be invoked by clicking on it. Pressing of the "Help" key on the keyboard would also have the same result. On line help instructions for a particular option in a menu can be read by highlighting that option and clicking on the "Help" icon. The on-line help instructions will give a brief description of how to use that command. If you click on the "cancel" icon of the main "debugger commands" menu, you exit the debugger.

A1.2 Start Up.

Type "m2db [object file]".

The "object file" in square brackets means that it is optional. The default for the object file is "a.out". As soon as "m2db" is correctly invoked, a main "debugger commands" menu will pop up.

A1.3 Source file lookup.

To look at the source file(s), choose the "Show Source" option from the main menu. A sub-menu for "Show Source" will pop up. It has two options

1. Current File and
2. File by procedure name

The Current file is set by default to the "main program" file. This is the file compiled to get the a.out file, which holds the main program module. At any instant the "Current file" is the file where the current execution of the program is taking place. For example let us assume that the executable file "a.out" is made up of three files viz. "test1.mod", "test2.mod" and "test3.mod", compiled and linked together. Suppose the program terminates abnormally when it is executing procedure "proc_one" which is in the file "test2.mod", then the "Current file" will be set to "test2.mod". It is useful in looking at the context of the program when it aborts, or has stopped on a breakpoint.

"File by procedure name" is used when you want to view specific procedures. If you want to look at a particular procedure that belongs to the program click on the "File by procedure name option" in the "Show Source" submenu. You will then be prompted for the procedure name. Type the name of the procedure exactly as it is in the source program. If the procedure is found in any of the files belonging to the program it will be displayed in a separate window, labeled with the file name. If, on the other hand, the procedure was not found in any of the files or if the file was not in the current directory, than an error message will be displayed in an error window. Scrolling through the file can be done with all the three mouse buttons by clicking them on the scroll icons. The scroll icons are located on the lower right hand side border of the window. The left button scrolls three lines, the middle button scrolls eight lines and the right button can be used to reach to the top or the bottom of the file.

A1.4 To Run the program

To run your program under the control of the debugger, click on the main "debugger command" menu to make it active and select the "Run" option. It will then prompt you to enter the command line options for your program. For example if you normally run your

program by typing

$ a.out op1 file1 file2

where "op1", "file1" and "file2' are the command line options then you would enter "a.out op1 file1 file2" when prompted. Once you have entered the command line options the program runs on a separate window and all output from the program will be directed to that window.

A1.5 Breakpoints.

Breakpoints are "stop signs" for the program. They allow you to stop the execution of the program at any point you wish, and let you examine the state of the computation of the program. You can look at the current values of the variables or change them if desired. This is useful in isolating the cause(s) of error in your program.

The "Breakpoint" option in the main "debugger commands" menu lets you set or clear breakpoints at any source line number in the program. You can also list all the line numbers in the program where breakpoints have been set.

To work with breakpoints click the "Breakpoint" option in the main "debugger commands" menu. A submenu pops up which shows three options

1. Set Breakpoint.
2. List Breakpoint and
3. Clear Breakpoint.

If you want to set breakpoints, click on the "Set Breakpoint" option. It will prompt you name of the procedure and the line number at which you want to set the breakpoint. The line number is the one that is seen when you look at the file using the "Show Source"

option from the main "debugger commands" menu. The line number must be that of a line number containing an executable statement in the program. the program runs. For instance, you cannot set breakpoints at line numbers which belong to variable declarations, comments or blank lines. For example if you want to set a breakpoint at line 12 in procedure "proc_two" inside the program then at the prompt

Name of the proc and line number: type [proc_two 12]

Then the breakpoint will be set at line 12 in procedure "proc_two". Now when you run the program in the debugger, execution will stop immediately before that statement.

To list all the line numbers where breakpoints have been set, select the "List Breakpoint" option in the "Breakpoint" submenu. All the line numbers with their procedures where the breakpoints have been set are then listed.

To clear breakpoints, you need to select the "Clear Breakpoint" option. This has a submenu which has two options

1. Clear Individual Breakpoints and
2. Clear all breakpoints.

To clear each breakpoint separately, select the "Cle: r Individual Breakpoints" option. This will prompt you to enter the procedure and the line number at which you want remove the breakpoint. Once you give the procedure and the line number is given, the breakpoint will be removed from that line. This can be used after "List Breakpoints" option which shows all the line numbers and the corresponding procedure names where breakpoints are set.

"Clear all Breakpoints" deletes all breakpoints.

A1.6 Continue Execution

You may want to continue execution after the program has stopped at a breakpoint. Click on the main "debugger commands" menu to make it active, and select the "Continue" option, your program will continue execution on the "Run" window.

A1.7 Variable lookup.

You might want to look at the value of a local variable or a global variable during the execution of the program, or when the program aborts. This is specially useful when the program is crashing with a "range error", "cardinal overflow", "bad pointer" etc., and you want to find out the erring variable.

To look at the value of the local variable or a global variable select the "Show variable" option in the main "debugger commands" menu. A Show variable" submenu pops up which has got two options

1. Show local variable, and
2. Show global variable

If you want to look at the value of the local variable, select the "Show local variable option" and click on it. This prompts you to enter the name of the procedure and the name of the variable whose value you are looking for. Type the name of the procedure and the variable exactly as they appear in the source program. The current value of the variable will be displayed in a new window. The value of the variable will be printed according to the type of the variable declared in the source program. If the variable is not found or if the named procedure is not active then an error message is displayed on the error window. A procedure is active if the call to that procedure has already been executed in the program but it did not return from that call.

The "m2db" program also knows about records, arrays and pointers. If you just give the name of the array, or a record than all the index vaiues in the case of an array or all the fields in case of a record, will be displayed. Some of the valid variable names are

* array[2][3]

* Name_rec.id

  Where "Name_rec" is of type "RECORD" and "id" is a field in that record.

* Data_rec^.name

  Where "Data_rec" is the "POINTER" to record and "name" is a field in that record.

In case you want to look for a global variable, select the "Show global variable" option in the submenu. Now you will be prompted to enter the name of the global variable. Once you enter the name of the variable, its value will be displayed in a new window. If the variable is not found the an error message is displayed in the error window.

A1.8 Changing variable values.

You may want to change the value of the variable to test if the program works when you give it the desired value you want. This is useful when you want to see how the program behaves under the new value(s) of the variable, thus helping you to isolate the cause of the error.

If you want to change the current values of the local or global variables click on the main "debugger command" menu and select "Change variable" option. A submenu pops up which has two options

1. Change local variable, and
2. Change global variable

If you are interested in changing values of the local variables, select the "Change local variable" option and click on it. This will prompt you for the procedure name, the name of the local variable and the value. The value may be a number or a character constant. For example if you want to assign the value "new_name" to a variable "Name" which is an array of character than you would type

Name "new_name"

when prompted. The new value can be enclosed either in single or double quotes. The value must be well defined, expressions which produce more than one value such as records are not allowed. For example you cannot assign new values to all the fields of a record at one time. You need to access individual fields one at a time. The same id true for arrays.

Some of the valid values with their variable names are

* a 3

  where "a" is a variable and is of type "1NTEGER" and is changed to contain decimal 3.

* proc.name "test"

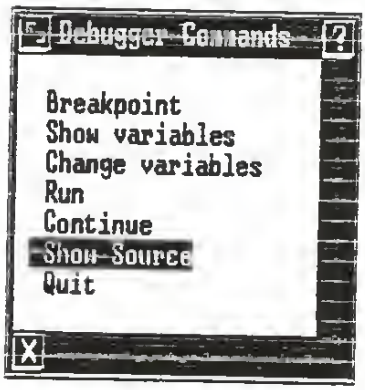
  where the variable "proc.name" is given a new character string "test".

New values given to the variable should be of the same type as of the variable. If wrong types are given an error message is displayed.

A1.9 To Quit the debugging session

To quit the debugging session click on the main "debugger commands" menu to make it active and select the "Quit" option. All the open menus and windows close and you are returned to the "UNIX System" window.

```
$ m2db a.out                    Thu July 23, 10:19 am
```




```
Pick one command
```

FIG. 14 THE DEBUGGER COMMANDS MENU

A DEBUGGER FOR MODULA-2 ON THE UNIX PC

by

SRIDHAR ACHARYA

B.E. (Electrical Engg.), Allahabad University, India

M.Tech (Industrial Engg.), Indian Institute of Technology, India

---

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1987

ABSTRACT

This report describes the design and development of a debugger for Modula-2 on the UNIX PC. During the design phase several existing debuggers on various machines were surveyed with a view to identify their advantages and limitations. This was helpul in the design of the debugger.

The debugger is intended to provide a "user friendly" facility for undergraduate students debugging their class programs. The user interface is designed with menus and windows supported by the UNIX PC. Basic features for debugging such as, source file lookup, setting breakpoints, viewing the values of the variables and changing them if desired are provided in the debugger. An on-line menu driven help facility is included to provide the student with on-line help for the commands during a debugging session.