Source level Symbolic debugger

by

David Livshin

written in 1986
published in *"Computer Language"* Volume 4 Issue 6, June 1987
recreated in 2020

Introduction

The traditional way of the software-product development involves repeatedly execution of the edit/compile/link/debug chain. In a such an environment changing a program may be time and resource consuming.

The approach, that has become popular recently uses interpreters for software development (see [3]). This solution has the virtues of being interactive - which speeds development – but you end up paying enormous time penalties when using interpreted code.

This paper describes a **ddb** - interactive source level symbolic debugger that in addition to standard debugger features (breakpoints, memory view/modification etc.), possesses the ability to execute C code on line. Its debugs the compiled object code and in the case that the source should be changed – extracts the changed code from the object file and interprets the new source instead. This allows enjoyment of the high speed of the object file execution as well as changing the program to be a low-penalty operation (no recompiling and relinking).

In addition to this, C constructs (expressions, flow control etc.) are incorporated with the debugger commands allowing much more greater and powerful flow control.

General Description

ddb is an interactive source level symbolic debugger. It is implemented for UNIX systems and may be used to examine COFF object and core files (see [1]) and provides a controlled environment for their execution. **ddb** also allows debugging of the slave processors. The object file is normally created by a C compiler with the ***-g*** option turned; however **ddb** may be used with object files of programs written in any (other than C) language with or without debugger information present.

Number of additions and modifications are under progress now; some of them are an implementation of a multi-window environment and support for debugging of the UNIX kernel.

ddb has two working modes:

- sdb mode
  in this mode **ddb** behaves as a slightly extended **sdb** symbolic debugger (see [1]). It has all **sdb** features implemented. Section 3 gives a brief description of the native **sdb** commands as well as new ones.
- C mode
  in this mode **ddb** command is a sequence of the C language statements. This statements may refer to the symbols of the debugged object file as well as may introduce new symbols. Commands in C mode may use **sdb** commands as escape sequences. Refer to section 4 for further information about C mode.

sdb mode

The **sdb** is a general purpose debugger that allows interaction with the debugged program on the source language and machine levels. Its provides a rich variety of commands such as modification/view of the memory, breakpoints, on line procedure execution etc.
The **sdb** debugger is well documented (see [1]); therefore we will give only a brief explanation of additional (to those of the native **sdb**) commands. Section 5.1 gives a more detailed explanation of the commands used in this article.

New Commands

Every new command has a two-character mnemonic that distinguish it from **sdb** commands and make it name be more meaningful.

History list

ddb maintains a history list into which it places every entered command. The number of commands available from this list is controlled by built-in variable '_HST_SZ_' (that may be changed in C mode).
The commands that access and manipulate the history list allow printing of the existing history list, re-execution of the command from the history list and editing command from the history list - the editing is done by a line-editor with the set of commands similar to the Emacs Editing Mode of the Korn Shell (see [2]).

Macros

Macros are implemented by making a history event permanent (it would be kept in the history list even if more than _HST_SZ_ commands are entered since this event).
The commands that supports macros allow creation of the macro, deletion of macro and printing the list of the active macros.

Memory check

This command accepts as an argument initial address and length of the memory region to be checked. It resumes execution in the single-instruction mode until at least one of the locations in the specified region is modified.

Switch modes

This commands enable establishment of the debugger working mode - sdb or C.

C mode

The **ddb** command in this mode is a sequence of the C language statements. In might be thought that these statements are the body of the block which is inserted in program right after the position that the program pointer points to. Thus, executing:

```
statement 1 ... statement n
```

is equivalent to executing program:

```
debugged program before program counter

{
  statement 1 ... statement n
}

debugged program after program counter
```

starting with the entrance to the newly introduced block.

If statement starts with `***$***', **ddb** assumes the text till semicolon or end of the line - whatever comes first - to be a **sdb** command.

Being evaluated in the context of the stopped program, **ddb** commands treat symbols exactly the same way as C does. This means that if symbol is acceptable from the inserted block - **ddb** treats it as a corresponding C symbol; otherwise **ddb** creates a local symbol with the type `long'.

Examples

Commands

This subsection describes **ddb** commands that are used in the following text. It may be skipped if there is no intention to go into greater details in the code session of 5.2.

- addr:**b** [commands]
  set breakpoint at the specified address.`commands' (if specified) are executed when breakpoint is encountered and execution continues. If **k** is used as a command to execute at breakpoint, control returns to **ddb** instead of continuing execution.
- **c**
  continue after a breakpoint or interrupt.
- linenumber **g**
  continue after a breakpoint with execution resumed at the given line.
- **i**
  single step by one machine instruction.
- **mc**
  set **ddb**’s working mode to C mode (see 2).
- **mo**
  set **ddb**’s working mode to sdb mode (see 2).

Examples

When involved **ddb** starts itsf work in the sdb mode. It means that the user may debug his program exactly as he will do it with the $B(sdb) debugger. This allows execution of debugger commands (setting/deleting breakpoints, running the program in different ways, memory display by different formats, source examination etc.). In order to transfer to C mode `**mc**' or `**mo**' should be executed.

Lets assume that program to be debugged is:

1.

```
2. #include <stdio.h>
3.
4. main()
5.  {
6.   struct
7.    {
8.     long y_m;
9.     struct
10.           {
11.            long y[6];
12.           } x_m;
13.         } x[6];
14.        long ar[6][6];
15.        long i, j;
16.
17.        for( i = 0; i < 6; i++ )
18.         {
19.          x[i].y_m = i*100000;
20.          for( j = 0; j < 6; j++ )
21.           {
22.            x[i].x_m.y[j] = i*10000 + j;
23.            ar[i][j] = i*1000 + j;
24.           }
25.         }
26.        for ( i = 0; i < 6; i++ )
27.         {
28.          for ( j = 0; j < 6; j++ )
29.           printf( "\t%d", x[i].x_m.y[j] );
30.          printf( "\n" );
31.         }
32.
33.      }  /*   main  */
```

Note, that user input appears in **bold** typeface, all terminal output is in regular typeface and comments appears in *italic* typeface. Note also that examples were computed on 68000 based computer (see disassembly in 5.2.1).

Simple example

```
$ ddb tst         debugging program `tst' by ddb
1 ddb>main:b      setting breakpoint at `main'
            132 - main+12     [x.c:17]
2 ddb>r           begin program execution
Debugging process 851.
bpt/trace at
132  |  main+12   [x.c:17]   17:   for( i = 0; i < 6; i++ )
3 ddb>mc          entering C mode
we are going to write a C program to execute 6 machine instructions
iii = 0;           `iii' is not acceptable from the `main' - assuming
                         it is not external - therefore it would be
                   created by ddb as a local symbol and will
                   disappear after exiting the created block
while( iii < 6 )
 {
  $i              Note how sdb command is used in C mode
```

```
  iii += 1;
 }
^D                terminates input of the C mode command
      [tst.c:19]
main+14:     0x86:            MOVE  i,D2
main+16:     0x88:            MULU  #$1c,D2
main+20:     0x8C:            LEA   (-168,FP),A2
main+24:     0x90:            MOVE.L      i,D1
main+26:     0x92:            MOVE.L      #0x186A0,D0
main+32:     0x98:            JSR   lmult
4 ddb>$mo          switch to sdb mode
5 ddb>c            continue execution
      0     1     2     3     4     5
      10000 10001 10002 10003 10004 10005
      20000 20001 20002 20003 20004 20005
      30000 30001 30002 30003 30004 30005
      40000 40001 40002 40003 40004 40005
      50000 50001 50002 50003 50004 50005
Process 851 terminated.
6 ddb>q         exit ddb
```

Code patching

Lets assume that we want to change assignment

```
    x[i].x_m.y[j] = i*10000 + j;
```

on the line 22 in the source program to

```
    x[i].x_m.y[j] = i + j;
```

when i is 1 and j is 0 or when i is 5 and j greater than 1.

```
$ ddb tst
1 ddb>22:b mc       setting breakpoint at line 22 and specifiing
                    command to be executed when breakpoint is reached
if ( ( ( i == 1 ) && ( j == 0 ) ) || ( ( i == 5 ) && ( j > 1 ) ) )
 {
  x[i].x_m.y[j] = i + j;
  $23 g
 }
^D
      164 - main+44     [x.c:22]
2 ddb>r            begin program execution
Debugging process 886.
      0     1     2     3     4     5
      1     10001 10002 10003 10004 10005
      20000 20001 20002 20003 20004 20005
      30000 30001 30002 30003 30004 30005
      40000 40001 40002 40003 40004 40005
      50000 50001 7     8     9     10
Process 886 terminated.
4 ddb>q
```

This example demonstrates the fundamental difference between interpreter and **ddb**. The program will be executed on the full speed as a regular object code. Only when

a specified breakpoint is reached (breakpoint at the line 22), the code, associated with this breakpoint, will be interpreted.

Conditional breakpoints

In many cases we wish to stop the program at a certain location only when some predicate is fulfilled. The usual **sdb** commands provide little help in solving such a problem (especially when location is inside large loop and/or predicate is complex).
The following example demonstrates the way, conditional breakpoints may be implemented in **ddb**. It also shows how C language constructs may be combined with the **sdb** commands.

Lets assume that we want to stop at line 23

```
      ar[i][j] = i*1000 + j;
```

but only when `j >= 5'  and `x[i].y_m >= 400001'; otherwise we want to see values of `i', `j' and 'x[i].y_m'.

```
1 ddb>23:b mc
if ( ( j >= 5 ) && ( x[i].y_m >= 400001 ) )
 {
  $k
 }
else
 {
  if ( ( i + j ) >= 8 )
  {
   printf( " i = %d  j = %d x[%d].y_m = %d\n", i, j, i, x[i].y_m );
  }
 }
^D
            202 - main+82     [x.c:23]
2 ddb>r
Debugging process 1002.
 i = 3 j = 5 x[3].y_m = 300000
 i = 4 j = 4 x[4].y_m = 400000
 i = 4 j = 5 x[4].y_m = 400000
 i = 5 j = 3 x[5].y_m = 500000
 i = 5 j = 4 x[5].y_m = 500000
bpt/trace at
202  |  main+82   [x.c:23]   23:       ar[i][j] = i*1000 + j;
3 ddb>c
      0     1     2     3     4     5
      10000 10001 10002 10003 10004 10005
      20000 20001 20002 20003 20004 20005
      30000 30001 30002 30003 30004 30005
      40000 40001 40002 40003 40004 40005
      50000 50001 50002 50003 50004 50005
Process 1002 terminated.
4 ddb>q
```

Conclusions

In this section we will summarize features that distinguish **ddb** from available debuggers.

Easiness to master

The traditional UNIX system development environment for a C programmer involves numerous tools, each of which implies different language (Shell, C, make etc.). This might lead to a large amount of time, spent learning them. Consequently programmer handle only part of the language, thus not exploring the tool to the fulliest extend. In the approach, that **ddb** presents such a problem does not exist. The **ddb** is designed to suite best the needs of the C programmer. Accepting as a command a C language statements will allow a C programmer (familiar with **sdb**) to master it fast and efficiently.

Power

ddb uses the C language for constructing debugger commands as well as patching the source being debugged; the patching, actually becomes the command of the debugger. This leads to constructions of remarkable power and flexibility. As most outstanding features are complete flow-control, on line code execution and code patching.

Speed

Because the program being debugged is executed in the object code, there is no slowdown in the execution time. This allows to combine the best of the two worlds in one product: fast turnaround for modifications, (offered by interpreters), and full speed of execution (that interpreters lack).

References

1. AT&T, "UNIX System V Programming Guide", April 1984.

2. Kochan Stephen G. and Wood Patrick H., "UNIX shell programming", Hayden Book Company, 1985.

3. Feuer Alan R., "si - An Interpreter for the C Language", USENIX Conference Proceeding, June 1985.