Symbolic Debugging: An Introduction

Timothy D. Chase
Corporate Computer Systems, Inc.
33 West Main Street Holmdel, New Jersey

Next to a trusty compiler it's a good symbolic debugger that tops my list of important software development tools. I would fear facing a day in my chosen programming profession without symbolic debug. It never fails to amaze me, though, how few people use symbolic debuggers let alone know what they are. In this article I will introduce you to symbolic debuggers, their technology and how they can be used to help programmers do their work. In addition, I will give you a list of basic features which should be looked for when selecting a symbolic debugger. No one debugger has all of the features I'll mention, but some do have an impressive subset.

There are as many different symbolic debuggers as there are compilers on different computers. For this reason, it is virtually impossible to talk specifically about any one debugger or language in detail. Instead, I will try to discuss symbolic debugging in general terms and leave you to find out the details of the symbolic debuggers available to you. I have tried to select a cross section of actual debuggers to use for examples. Those mentioned include Codeview for the IBM PC by Microsoft, HP's Toolset and CCS' TRAX both for the HP3000 and HP's DEBUG/1000 for the HP1000 RTE-A system.

Finally, if you do any serious programming or if you have to maintain the work of other programmers and you *don't* have access to some form of symbolic debugger, you should complain to someone. Loudly.

What is a symbolic debugger?

Before answering that, we need to briefly review the program development process. If you recall the *dark times* in computer programming then you remember assembly language. Assembly, or machine language, was the original way to program computers. Only one level removed from the 1's and 0's computers *really* understand, assembly language is very difficult for biological systems (people) to understand. So to save our sanity *high level* programming languages were designed. The idea behind high level languages is that the computer, itself, can be used to translate the high level language into the more basic machine language. This method of programming quickly became successful but its success brought about another problem — debugging.

The problem with debugging high level languages stems from two basic sources. First, operating systems are designed to support applications written in many different languages. In fact, on most

computers, every language finally compiles (is translated) into a *relocatable* or *binary* module which is the same regardless of the original source language. This is called an *object module*. Object modules are then *linked* together by a system utility to form a finished application. The problem is that after the linking, the operating system has lost track of the original form of the language. When problems show up, the operating system only has information about the object form of the application and very little information about the original source.

The second problem comes from a mismatch in computational models. You, as a high level language programmer, think of the computer in an *artificial* way. If you program in COBOL, then you think of your computer as a COBOL machine regardless of the underlying hardware. This is one of the basic features of high level languages: you don't have to understand the *real* computer in order to do your work. The real computer, however, is usually quite different from the conceptual model created by the high level language. The problem is that the operating system relates bugs to you in terms of the real computer rather than the high level programming model. Faced with the error message like `SYSTEM STACK UNDERFLOW` the COBOL programmer is at a total loss. The COBOL model of programming does not contain system stacks or underflows. In order to deal with bugs of this sort, the high level programmer is forced to go to low levels in the system.

In a nutshell then, symbolic debugging is a method by which the programmer can receive information about bugs as well as look for bugs at the same level as the program was originally written. The successful symbolic debugger maintains the illusion of the computational model created by the high level language. By keeping the programmer at the source level during debugging, the symbolic debugger makes the same impact on the debugging process as the compiler made on the coding process.

How are bugs found without symbolic debug?

Without symbolic debugging there are three basic ways in which programmers debug their programs. They can analyze system output, they can put special debugging code in the source or they can resort to assembly language debuggers. Although widely used, each of these techniques has some serious problems.

Analyzing system output usually means pawing through miles of printer dump output. The application was running fine for a while and then it blew up. The dump comes out and then you try to reconstruct what happened. The problem, of course, is that it's difficult to find something wrong in the maze of maze of registers and memory locations. When you finally do find *something* you might actually be looking at second or third order effects. In other words the *real* bug caused something to happen which caused something else to happen which you saw (second order effect). In addition, you need to learn how to read dumps and, usually, you also have to learn a bit about assembly language in order to navigate the output. This type of debugging was the primary debug weapon in the arsenal of the batch programmer of the 60's

The next approach is to add special debug code. It's surprising how many programmers still use the "put in DISPLAY statements" method to finding bugs. This technique requires the programmer to insert little "hi, you got this far" messages in the program source code. The program is run and the output is then analyzed. This technique has several problems. First, you have to know where the

bug is (roughly) in order to know where to insert the print statements. Second, inserting debugging statements alters the program and may even make the bug go away. Finally, the debugging statements are, by definition, never in the program when you need them. This means you have to put them back in, recompile, re-link and they try the program again. Also, unless the debug statements have testing associated with them, they can generate reams of output which must be carefully analyzed (shades of the output dump).

The final nonsymbolic approach is to use an assembly language debugger. These debuggers were developed by the assembly language programmers and are, in fact, pretty close to symbolic debuggers if you program in assembly language. The problem here, however, is that you must descend to the depths of assembly language in order to use them thus defeating the major reason for using high level programming languages. If, as many other programmers, you program in a *portable language* you may regularly work on several different computers. This means that you'll have to be adept at the assembler on several machines. Still, even with its problems, the assembly language debugger is usually the best alternative if full symbolic debugging is not be available on your system.

How is symbolic debugging different?

The approach used in symbolic debugging is different from the first two alternatives given above. It is also much simpler than the assembly language approach. The fundamental difference is that it is an interactive action. The programmer is dynamically interacting with the program being debugged. Other nonsymbolic approaches are static. You are analyzing a print out of what happened or you are watching the output generated by special debug code. With symbolic debugging you are encouraged to try experiments to prove or disprove theories about what might be causing a bug. If unexpected results are seen, you can quickly try a new course of action. In a large system without symbolic debugging it is often difficult even to discover which module the problem is in. With the right set of symbolic debug features this can border on simple.

As a maintenance tool for supporting a software system symbolic debugging is invaluable. Usually during this part of an application's life cycle it is supported by people who did not do the original development work on the project. Symbolic debugging techniques enable these people to watch the program execute. Its logic flow and operation become very real to the support people and their job is made simpler so their throughput is increased and the support costs are reduced.

How does symbolic debugging work?

The symbolic debugger is actually only one part of a series of cooperating programs. The symbolic debugger (depending on implementation techniques) requires information from both the compiler and the linker. Remember, during normal compiling, most of the source information is lost when the object modules are produced. In order to have symbolic debugging, the information usually discarded by the compiler must be passed through into the object modules. In addition, because object modules may be *relocated* in memory as a result of the linking process, the linker (or loader or segmenter) must output information to a special file called a *debug information file* or a *debug map file*. The result of the compile-link operation then is a completed application program along with a debug information file.

The basic idea behind a symbolic debugger is to be able to execute the application normally by issuing the appropriate start up commands to the host operating system. If, however, the program displays some aberrant behavior, the symbolic debugger may be invoked which then runs the program in a different mode making it available for debugging. During debugging there are two program executing: the application program and the debugging package itself. The programmer interacts with the debugging package while the debugging package interacts with the application program being tested.

There are, essentially, two classes of symbolic debuggers. These are *intrusive* and *nonintrusive.* An intrusive debugger is one in which the application is compiled in a special way which causes some amount of debug code to be placed in it. A program compiled with debug code inserted in it is said to be *instrumented.* A noninstusive debugger does not require any special debug code to be inserted into the application. Totally nonintrusive debuggers are rare and need a great deal of help from the host operating system. More than likely the symbolic debugger you will be using is an intrusive debugger.

It is important to determine the amount of intrusion which is required for debugging. Normally the compile process is altered in some way when the symbolic debugger is to be used. For example, in Microsoft's QuickC compiler, the user selects if the compile will result in a debuggable object module or not. Likewise in HP's TOOLSET COBOL debugger all source modules to be debugged must have the $CONTROL SYMDEBUG option. If the intrusion is small in terms of consumed program resources (memory space and execution time), then applications can be routinely compiled with the debug option. By doing this, especially on new applications, the debugger is always available when a problem is discovered. This removes the need to recompile and re-link the application when a bug shows up. In fact it is quite common for an address sensitive bug to vanish when a program is compiled in the debug mode. By always compiling in debug mode this can't happen.

What are symbolic debugger features?

Although source languages differ widely when it comes to features, symbolic debuggers are fairly similar in terms of the basic offering. Different debuggers are differentiated by how the features are implemented and the ease of their use.

The debugger model the programmer deals with is the original source file and the currently executing statement. Usually the source statements around the current statement are displayed on the screen along with a command area. The program under test is placed in a suspended state so that it is not executing and the debugger, itself, is waiting for commands from the programmer. Note that because of the required programmer *think time* a symbolic debugger is not normally useful for real time applications which must execute without interruption. If a program must read a sensor every 2 seconds and it takes you 4 seconds to type a command to begin execution, you're in trouble!

Most symbolic debuggers are command driven. This means that you type in commands to bring the debugger into action. DEBUG on the HP/1000 RTE-A is fairly typical. DEBUG's commands are one or two characters followed by one or more arguments delimited by various characters depending on the selected options. The arguments, however, reference objects normally found in the source language using at least some of the syntax of the original source language. These objects

might be statement numbers or user identifiers (as opposed to octal memory addresses). PC's, having powerful screen management technology, can support debuggers which use a mouse.

The Microsoft Codeview debugger is a good example of a debugger which is both command and mouse driven. The mouse is a natural tool for the symbolic debugger. If you want to execute your program up to a given line, you just point at the line with the mouse and click a button. This highly tactile way of manipulating your program is both natural and easy to learn.

A good symbolic debugger should have as many of the following features as is possible:

Breakpoints

The basic operation of the symbolic debugger is to insert breakpoints into the program under test and then execute the program until it *hits a breakpoint.* When a breakpoint is hit, the program suspends execution and you can *look around.* For example, if you suspect that a problem is at line 100 (the program aborts from that location), you can set a breakpoint at that line and execute the program. When the program attempts to execute line 100, the breakpoint is hit and you can check the values of various data locations to make sure everything is as it should be.

The simple "stop when you hit it" breakpoint may be augmented by several other different breakpoint types. This may include *iterative* and *conditional* breakpoints. Iterative breakpoints usually have a count associated with them. The program is allowed to pass thought the breakpoint for a specified number of times (the count) and then the program stops. This is especially useful when you know that a problem occurs on a given line after a certain number of "events" happen. For example, your program blows up after reading 125 input records. You could place an iterative breakpoint with a count of 124 on the input read statement. The program will stop just before reading the 125th record.

The conditional breakpoint actually allows you to specify some type of test condition. When the breakpoint is struck, if the condition is met, the program will suspend. If the condition is not met, the program continues execution. For example, your program aborts when a value gets greater than 6742. You could set a conditional breakpoint with the condition "value greater than 6742". Each time the breakpoint is hit the debugger checks to see if the value is greater than 6742. If yes, the program under test is suspended for you. This is especially useful when a bug displays itself only when certain data values are being processed.

Single step

In addition to breakpoints another vital feature is the ability to single step source statements. This is usually accompanied with a representation of the current statement on the screen. This might be a marker to one side of the statement (CCS TRAX) or, more spectacularly, by changing the color of the current statement (Microsoft QuickC). Whatever the display technique, the single step operation should have these important features:

Simple command to repeat single stepping. When you are single stepping, you normally want to execute several lines one after the other. This should not require long command sequences. A softkey or a carriage return is all that should be required to execute the "next" statement.

Single step should come in two different "flavors". You should be able to step *into* subroutines or to step *over* subroutines. Stepping into subroutines means that you continue to single step when executing the subroutine. This gives you a look at the operation of the subroutine if you think the bug might be there. Stepping over a subroutine causes the subroutine to execute at full speed with the next single step operation at the source statement which immediately follows the source level call to the subroutine. This is important when you know that the subroutine is bug free, but you wish to trace the flow of the calling program.

Although not provided by all symbolic debuggers (multiple breakpoints accomplish the same thing) single step execution is a real time saver when trying to follow complex logic flow. For example, the outcome of a "go to depending on" statement is simple to figure out with single step, but quite a bit harder to figure out using breakpoints.

Variable display and modification

There are several different ways to implement this important feature. Some debuggers (HP DEBUG/1000, CCS CView, HP TOOLSET) have commands to display the current contents of program variables whenever you want. The display command uses the source level name for the data object as well as the source level syntax to access the object. This can be quite complex. For example, the C debugger CView from CCS allows you to enter a complete C expression which is then interpreted with the computed results printed out. A more common approach is to enable the programmer to print out simple data identifiers, perhaps indexed with constants.

A different approach is the one used by Microsoft's Codeview and to a lesser extent HP's TOOLSET. These debuggers allow you to define *watch expressions* which automatically display the contents of user identifiers whenever the identifier changes during the execution of the program. This feature coupled with single step can be very useful for determining when a data value gets incorrectly changed. The Microsoft implementation of this feature allows the programmer to open up a special watch window which contains the current values of specified variables.

The automatic display of variable changes as opposed to upon user request can take its toll in extra added code unless there is some assist from the hardware. The compiler must add special subroutine calls on every assignment statement in order to trap variable changes. The overhead may become prohibitive.

Another, less costly way of achieving much the same effect is to be able to *attach commands* to breakpoints. The commands are usually display commands. The idea is that the commands will be held in ready waiting for the breakpoint to be hit. When the breakpoint is hit, the attached commands are executed. Doing this, for example, would allow a programmer to attach a number of display commands to a breakpoint followed a continue execution command. Whenever the breakpoint is hit, the display commands execute followed by the continue command. The net effect is that the data objects are printed will little overhead.

A final aspect of data display is the format of the output. It is a useful feature if the debugger is capable of outputting data in several different user selectable ways. Sometimes, for example, you may wish to view a decimal number as a decimal number, but other times you may want to see the same value in octal or hex. Most symbolic debuggers accommodate this.

The flip side of variable display is variable alteration. Most symbolic debuggers enable the programmer to change the contents of variables. Again, the original source names for the variables are used along with the source syntax. All debuggers support changing the values of simple scaler variables (non array type). Some allow you to change the values of an array variable with a single command.

Being able to change the value of a variable is a surprisingly useful feature. It often allows you to change a good value to a bad value just to see how the program will react. Or it allows you to patch an incorrect value in order to continue looking for a different bug without re-linking. On the HP/3000, for large COBOL programs, the time required to re-segment can be large. Being able to find *another* bug without recompiling and segmenting is an important feature.

Flow control

Most debuggers offer several commands which effect the execution flow of the program under test. There are usually two different types of flow control. The first is the proceed command. Using a proceed will cause the program you are debugging to begin execution under the control of the symbolic debugger. The program will continue execution until it either terminates, is terminated by the host operating system or hits a breakpoint. There are several variations on this theme which are available.

For example, the DEBUG/1000 package gives the programmer the option of keeping the program under test *alive* after a system termination. This means that if the program under test executed an illegal instruction or violated the memory protection scheme, rather than terminating the program, control is passed back to the debugger so that the programmer can examine the program after the crash. DEBUG/1000 even allows you to re-execute the program after the halt (presumably after changing data values).

The proceed commands found in symbolic debuggers usually have several different options associated with them. The CCS TRAX debugger, for example, provides for a *temporary* breakpoint which may be set with the proceed command. The proceed command can optionally indicate that the program under test should execute up to a given point and then stop. Although the same effect may be achieved with a breakpoint, combining the breakpoint with the proceed makes life a little easier.

Another important (but dangerous) flow control command is the *go to* command. This allow the programmer to redirect control to another part of the source program. This is often used to re-execute statements after a variable has been changed to check the new outcome. The problem with this command is that its power often lets you abuse the source language by going where the compiler never intended you to. In COBOL, for example, you might circumvent a proper exit from a PERFORM statement by using a debugger go command. This might ultimately introduce a (non-

repeatable) bug which isn't really a bug at all. As in programming using a go to in debugging should be done with care.

Some miscellaneous features...

We have discussed most of the mainstream symbolic debug features and their typical usage. What remains is a brief collection of features which have appeared in various debuggers and are interesting enough to comment on. Unfortunately, no one debugger has all of these features — too bad.

Dual displays

This is a nice feature found in the Microsoft Codeview debugger. It enables you to have access to the source code as the program executes, yet still see the output generated. You can switch back and forth between the source display and the output display by pressing a function key on the PC. This feature is vital to anyone designing interactive applications.

A similar feature is available in the CCS TRAX debugger. Designed to aid in the debugging of COBOL based V/PLUS programs, TRAX provides a feature similar to Codeview with HP character oriented terminals. Again, you can switch between the output (V/PLUS) display and the source code debugger display.

Low level machine

Some symbolic debuggers (in a seeming contradiction to their major claim to fame) enable you to gain access to the underlying machine. These features are used infrequently, but when you need them they are nice. For example both Codeview will *explode* the source code into assembly language with intermixed source code statements. You can then extend the commands to include assembly language versions. In other words single step becomes single *instruction step* instead of single source statement step. DEBUG/1000 provides the ability to view assembly language, but not to interact with it. TRAX gives assembly language in a separate window of the output display which overlays the source.

These features are useful if you just can't figure out what went wrong at the source code level. All symbolic debuggers that I've seen also include an optional register display if you are working in assembly language mode so that you can see the effect of the machine state as instructions execute.

Another (not quite so low level) feature is the ability to map addresses back into source line numbers. All symbolic debuggers have the ability to map statement numbers into the absolute addresses of the executable program, but some have the ability to reverse this process. This becomes useful when you have to interpret system error messages. The operating system usually gives you an error message which references an octal address. The ability to map this address back into a source file and a line within the source file can be a real time saver.

There is another class of features which give you a history of your executing program. This is done in a variety of ways with differing degrees of usefulness. Toolset provides a *paragraph trace* feature which prints out the last several paragraphs which have been executed in a COBOL program.

TRAX will display the *call chain* which displays which subroutines were called in order to arrive at the statement you are currently executing. Codeview has a similar feature, but with the added ability to actually breakpoint the call chain so that when a subroutine returns the breakpoint is struck.

All of these features provide a way to determine how you got to a given source statement. At first this might seem silly, but it is quite important. If you have a subroutine which is used in several different places you may not know where the subroutine was called from. Placing a breakpoint in the subroutine and then using a *history* feature will give you a good idea how you got there.

Conclusions...

Symbolic debugging has finally come of age. There are currently a number of excellent debugging packages out on the market which greatly reduce the time required to find and fix bugs. Although sometimes looked upon as a *frill*, the symbolic debugger is as important a tool for serious software development as the compiler itself.

Although debuggers differ substantially in detailed implementation many have the same features. This paper has discussed the basic features required of any symbolic debugger in order to be useful. Briefly they are:

- **Source based.** This means that debugger commands and displays should be oriented around the original source program. Commands should reference source statement numbers and source user identifiers rather than machine addresses.

- **Breakpoints.** This are markers which may be set in the program being debugged. They may include *simple*, *conditional* and *iterative*.

- **Single step.** This feature lets you execute a source statement at a time. It should offer the ability to step *into* subroutines and to step *over* subroutines.

- **Variable access.** You should be able to *display* user variables and *alter* their values. Variables are accessed by their source names using the source language syntax.

- **Flow control.** You should have the ability to navigate your source program; to proceed to specified statements and to skip statements by using a *go to* type of command.

For most programming applications symbolic debugging is by far the best way to debug programs. Because it is highly interactive it enables the programmer to easily develop hypotheses about what is wrong and then try experiments on the executing program to uncover the bug.