THE DESIGN AND IMPLEMENTATION OF DAD, A MULTIPROCESS, MULTIMACHINE, MULTILANGUAGE INTERACTIVE DEBUGGER

Kenneth E. Victor
Augmentation Research Center
Stanford Research Institute
Menlo Park, California

The Design and Implementation of DAD, A Multiprocess, Multimachine, Multilanguage Interactive Debugger

Kenneth E. Victor
Augmentation Research Center
Stanford Research Institute
Menlo Park, California

THE DESIGN AND IMPLEMENTATION OF DAD, A MULTIPROCESS, MULTIMACHINE, MULTILANGUAGE INTERACTIVE DEBUGGER

ABSTRACT

Debugging tools and techniques have varied from toggling switches and reading lights at a CPU console, to very sophisticated, high level language interactive debuggers. Because of its unique internal organization, the interactive debugger described here can provide high level language debugging for several languages and allows the programmer to debug one or more processes, which may be executing on different machines from each other and/or from the debugger.

THE DESIGN AND IMPLEMENTATION OF DAD, A MULTIPROCESS, MULTIMACHINE, MULTILANGUAGE INTERACTIVE DEBUGGER

1. INTRODUCTION

As the sophistication of computer programs and the environments in which they execute has grown over the years, so has the sophistication of the approaches used to debug these programs. This paper describes a newly implemented interactive debugger developed at SRI. We discuss the requirements and philosophy that guided our design. We present a brief analysis of the interactive debugging process and the internal design and implementation of our debugger.

2. DAD'S GOALS

The Do-All Debugger (DAD) grew out of a need for a debugging tool to operate in the National Software Works (NSW) environment[2]. The NSW is a distributed multi-processing system where users operate primarily on-line and interactively. Interactive tools are used to edit and specify batch processing, and to examine the results of batch processors. The NSW and its tools are written in a variety of languages that execute on a variety of machines. DAD would assist in the development and maintenance of portions of the NSW and some NSW tools.

We wanted to provide a "consistent" interface to the user, regardless of the process being debugged. With such an interface the commands (and where feasible, the techniques) for debugging would be the same for all machines and languages. Evans and Darley claim that "... if the appearance to the user of the debugging system ... could be made the same over a number of ... time-sharing systems ... considerable savings could well be realized."[3]

The following goals were determined by the target NSW environment and from our previous experiences developing software engineering tools and interactive systems:

(1) It would be an interactive debugger, since it would be used initially in an interactive environment and interactive debuggers are generally accepted as the most powerful.

(2) It would be capable of debugging one or more processes, where these processes might or might not be executing on the same machine as each other or on the same machine as the debugger. This is necessary to assist in the development and maintenance of a distributed multi-processing system.

(3) It would be a high level language debugger, capable of supporting many languages. This is because the tools whose development and maintenance it would support would be written in a variety of languages, and because high level language debugging is more powerful and more "natural". However, the debugger would operate on compiled code rather than interpret source statements in order to avoid problems such as those associated with a poor simulation.

(4) A basic set of core commands and capabilities would be defined that would be useful for a broad set of implementation languages and operating system environments.

(5) The implementation of the debugger should allow growth in both additional core commands and capabilities and the support of new languages and machines. This design goal dictates that the internal structure of the debugger be modular, and that modules be dynamically loadable.

(6) The design should allow the debugger to run in both an NSW environment and in a stand-alone TENEX environment[8].

There were debuggers that satisfied various subsets of these requirements, but no single debugger could satisfy all of the requirements.

3. THE INTERACTIVE DEBUGGING PROCESS AND DAD'S ORGANIZATION

The tasks in interactive debugging fall roughly into the following categories:

The user (the programmer who is doing the debugging) specifies an action.

The user's functional specifications are translated into calls on specific debugger routines.

The debugger routines read and/or write the bits in the address space of the program being debugged, and read and/or modify the state of the program being debugged.

The bits from the program, and the program's state information, are interpreted in a manner that is meaningful to the user and consistent with the language the program is written in.

The resulting interpretations are presented to the user.

Upon examining these tasks, we saw that the internal functions of a debugger can be divided roughly into those that: communicate with the user; are independent of the language in which the program being debugged was written and independent of the operating system under which the program is executing; are language dependent; and are operating system dependent. These functional areas suggested an internal modular structure for the debugger.

DAD consists of a number of modules, each supporting one of the above functional areas. Modules may or may not execute on the same machine as each other, and inter-module communication occurs via well defined communication protocols. The debugger configuration can change dynamically, loading and unloading modules as needed. The modules are:

A frontend module (FE) for all communication with the user. This module consists of, among other parts, a Command Language Interpreter (CLI)[1], a data base called a grammar that represents the commands of the debugger, and routines for communicating user commands to the rest of the debugger and for receiving information from the rest of the debugger for display to the user.

A debugger dispatcher (DD) module that receives functional command specifications from the frontend and calls various routines to implement these requests and transmit results meaningful to the user back to the frontend.

A language module (LM) for interpreting the bits and state information of the program being debugged in a manner appropriate to the language in which the program was written.

An operating system module (OSM) with responsibility for

reading and writing the address space and state information of the program being debugged.

This modular approach, along with well defined and published functional specifications for each module and for inter-module communication, produces some benefits. As long as each module conforms to its functional specifications and communication standards, it can be implemented in any manner, and in any language, desired. It is also possible to dynamically load and unload individual modules.

This collection of modules, namely the frontend, the debugger dispatcher, the collection of language modules, and the collection of operating system modules, comprise DAD. This modular approach enables DAD to be extensible and to "be all things to all people".

While each of DAD's modules could exist as a separate process and communicate via an inter-process communication protocol, in the current implementation, the frontend exists as a unique process; the debugger dispatcher, the language module, and the operating system module coexist in a second process known as the backend. The backend may be on a different machine from the frontend. Thus, in a debugging situation we have three (or more) processes, executing on one or more machines: (1) a frontend process for all communication with the user, (2) a backend process for performing the commands specified by the user in his/her communication with the frontend, and (3) one or more target processes, i.e., the processes containing the programs to be debugged.

The user need not be concerned with loading and unloading language and operating system modules. These functions are performed automatically by DAD. The user merely indicates which process he is concerned with and DAD does the rest based on models of the process structure built and maintained by DAD. Additionally, when an event associated with a specific process occurs (such as executing an illegal instruction, or encountering a user defined breakpoint) DAD ensures that the proper modules for the process in which the event occurred are loaded.

4. DAD'S MODULES

In discussing the individual modules comprising DAD, we will

start with the frontend, the module closest to the user, and work our way down through the logical structure until we reach the bits of the program being debugged.

4.1 THE DEBUGGER FRONTEND

The debugger frontend (FE) is responsible for all interactions with the user and contains (among other parts) a Command Language Interpreter (CLI), which interprets the debugger grammar. The grammar is a data structure produced as output of the Command Meta Language (CML) compiler[4]. The user command language, and all user interactions, are thus specified as a CML program. This program is compiled by the CML compiler, producing a grammar that is then interpreted by the CLI.

The command language has not been tailored to the syntax or semantics of any one language. Rather, the command language is general and deals with concepts that are common to a number of languages. The command language is basically of the form:

Command, Argument, Qualifiers, Confirmation.

The user specifies the action to be performed, the base on which the action is to be performed, optionally some qualifiers for an instance of the command, and when he or she is done specifying the command, a confirmation. After the command confirmation, the command is executed (by the backend).

4.2 THE DEBUGGER DISPATCHER

The debugger dispatcher (DD) is that module of DAD's backend responsible for communication with the debugger frontend and for dispatching user requests (made via the debugger frontend) to the appropriate routines in language and/or operating system modules.

Included in the data structures maintained by the DD are models of the target process structures and indications of

the appropriate language and operating system modules for each target process. (The DD also contains the code that supports the communication protocols for frontend-backend communication, and the runtime environment for the high level language used by DAD itself.)

In response to user actions, the FE calls procedures in the debugger backend, namely in the debugger dispatcher. There is not a one-to-one mapping between DD routines and commands, since different commands may call the same DD routine, passing it different arguments, or one command may use more than one DD routine. The DD routines called then perform the requested action -- perhaps by calling other DD routines or routines in a LM or OSM -- and return to the FE.

The DD will look in the dispatch table associated with the current LM or OSM to determine if the request is supported. If the request is not supported, an appropriate error message will be generated and returned to the FE to be presented to the user. The DD also performs some syntactic and semantic checks on the arguments for a request. If the arguments are invalid or illegal, the DD will either generate an appropriate error message and return to the FE, or the DD will interact with the user (via the FE) to get valid arguments.

Finally, the debugger dispatcher will invoke the appropriate DD, LM, or OSM routine(s) to satisfy the request. The invoked routine will perform its function and return to the DD. The DD then returns to the FE, passing along any strings, including error messages, generated by the invoked routine(s) to be presented to the user.

4.3 LANGUAGE MODULES

A language module is responsible for any language specific function, such as interpreting symbolic input according to the semantic and syntactic rules of the current high level language, or displaying a cell in the current high level language. Each language module in the debugger can support one and only one language, running in a specific environment. For example, there would be separate BCPL[7] language modules to support BCPL on a TENEX and to support BCPL on an ELF[5].

Although a programming language may be functionally

independent of the target execution machine, the implementation of that language is almost always highly machine dependent. Thus, to interpret the bits of a program requires that the LM have knowledge of the implementation of the language.

We have separated the functions of the OSM from the machine dependent part of an LM because for any specific machine there are usually a number of languages which can compile programs for that machine. Each of these languages requires its own LM, but all the LMs use the same functions to access the target program, and hence require only one OSM. Thus, within DAD there is one OSM for each machine, but for each machine there may be many LMs.

4.4 OPERATING SYSTEM MODULES

Any interactive debugger must provide facilities to examine and manipulate the address space and state informaton of the process it is debugging. The function of the operating system module is to isolate all such code into a single module with a well defined interface.

Isolating these functional routines into a single module makes it possible to dynamically load the module or to replace one module with another. Processes that execute either on the same machine as the debugger or on a remote machine may be debugged by loading the proper OSM (and LMs as appropriate).

To illustrate, a debugger routine for examining a cell in the address space of a target process will always call a routine in the OSM to return the contents of the cell, rather than reading the cell directly. The debugger routine need not know how the OSM got the contents of the cell, or, for that matter, whether or not the target process is a process on the same machine as the debugger.

4.5 TARGET PROCESS MANIPULATION

An OSM may perform its functions in any manner it chooses as long as it obeys the specified interface conditions. To interactively debug any process, the OSM must be able to exercise certain controls over that process, such as reading and writing the process' address space, and stopping and resuming its execution.

When running under a process oriented operating system, with the debugger at the top of the process tree, the OSM could exercise these functions by operating system primitives that exert control over inferior processes. If the debugger and the target process are both under the control of a common process, the debugger may perform some of these functions via operating system primitives directly, while for others it may have to request the common head process to perform the function.

If the debugger and the target process are not running on the same machine (i.e., cross-debugging), or under a common process in the same machine, the OSM must communicate with (procedures in) the target process (or in a process that has control over the target process).

The OSM functions required for debugging are few and simple (e.g., read and write the address space and/or control state of the target process). Thus, the procedures required on the target machine to implement these functions are also few and simple. By providing a small set of procedures on a target machine to communicate with an OSM which is executing on a more powerful machine, we are able to provide previously unavailable power for the debugging of programs executing on minicomputers or microcomputers.

5. CONCLUSION

DAD, with its unique internal organization, provides interactive debugging of multiple processes distributed over a number of machines. The modular structure allows high level language debugging of programs using a variety of higher level languages. In addition, the modular approach allows extensibility of the

debugger to cover new environments without the user having to learn a new debugging discipline.

BIBLIOGRAPHY

[1] D. Andrews, et al., "User Interface System for a Computer Network Marketplace", Augmentation Research Center, Stanford Research Institute, Menlo Park, Ca., December 1976, SRI-ARC Journal Number <27266,>.

[2] R. Balzer, et al., "Design of a National Software Works", Information Sciences Institute, Marina Del Rey, Ca., December 1975, ISI-RR-73-16.

[3] T. G. Evans & D. L. Darley, "On-Line Debugging Techniques - A Survey", AFIPS FJCC Conference Proceedings Vol. 29, AFIPS Press, 1966, pp. 37-5Ø.

[4] C. Irby, "The Command Meta Language System", Augmentation Research Center, Stanford Research Institute, Menlo Park, Ca., January 1976, SRI-ARC Journal Number <27266,>.

[5] D. Retz & B. Schafer, "The Structure of the ELF Operating System", AFIPS NCC Conference Proceedings Vol. 45, AFIPS Press, 1976, pp. 1ØØ7-1Ø16.

[6] K. E. Victor, "The Design and Implementation of DAD, a Multiprocess, Multimachine, Multilanguage Interactive Debugger", Augmentation Research Center, Stanford Research Institute, Menlo Park, Ca., January 1976, SRI-ARC Journal Number <27399,>.

[7] "The BCPL Reference Manual", Bolt Beranek and Newman Inc., Cambridge, Mass., September 1974.

[8] "TENEX Jsys Manual", Bolt Beranek and Newman Inc., Cambridge, Mass., September 1973.

Mr. Victor graduated from Brandeis University with an A.B. in Physics in 1968. He has worked for IBM, Hewlett Packard, and for the past six years with the Augmentation Research Center (ARC) at Stanford Research Institute. While at ARC, he has been involved with the design and implementation of NLS - the oNLine System, and with the design and implementation of various software engineering tools and systems.

* A longer, more detailed, version of this paper is available from the author[6]. The work reported here was supported in part by contract #F30602-75-C-0320 with the Air Force Rome Air Development Center for the National Software Works project of the Defence Advanced Research Projects Agency.