A DEBUGGING TOOL FOR DISTRIBUTED SYSTEMS

Peter C. Bates*, Jack C. Wileden** and Victor R. Lesser*

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

Our belief that traditional debugging techniques will be inadequate for distributed programs has led us to take a new look at what debugging entails and how to approach debugging in the distributed system environment. This paper reviews the basis of an approach to debugging called behavioral abstraction and presents a design for a debugging tool based on behavioral abstraction. The tool itself may be distributed in the network and employs methods to overcome a number of the difficulties posed by distribution of programs over systems of processors.

THE PROBLEM

At some stage of their existence, most (non-trivial) computer programs contain errors that cause unacceptable behavior [8]. Errors reveal themselves in two ways — either a program exhibits poor operational performance or it produces incorrect results that subsequently lead to partial or total failure of the software. In many cases, poor performance might be acceptable, at least for some period of a system's life. Rarely, however, is outright failure an acceptable system behavior. Debugging as an activity is an attempt to discover the causes of, and in some fashion repair, existing errors in programs. Our belief that traditional debugging techniques will be inadequate for distributed programs has led us to take a new look at what debugging entails and how to approach debugging in the distributed system environment. In particular, the traditional methods fall short due to problems related to truly concurrent activities, the lack of absolute control over a distributed system and the increased complexity arising when tens or hundreds of processors are cooperatively performing a complex computational task.

The remainder of this section presents our view of what debugging entails and the traditional approach to debugging, followed by arguments as to why this approach will not be satisfactory in a distributed programming environment. The second section reviews the basis for the application of behavioral abstraction to debugging. The third section presents a design for a debugging tool based on behavioral abstraction.

A Characterization of Software Debugging

An error is a manifestation of a difference between an assumed system model and the actual behavior of the system. The debugging activity is concerned with isolating these differences and explaining the behavior that has produced them. In general, the effects produced by errors are only indicators or symptoms of the true errorful behavior. It may be impossible to predict all of the effects that a particular error will cause, but the ability to accurately predict effects other than the ones leading to the original error observation is important to the verification of the system model under scrutiny.

There are two elements essential to the successful completion of the debugging task: the ability to monitor, in some meaningful way, the relevant system behavior so as to gather detailed information about that behavior and how it differs from the expected model, and the ability to perform experiments based (implicitly or explicitly) on the information gathered [1,6,7,9,10]. Through the interaction of these two elements a debugger (i.e., the human who is undertaking the debugging task) attempts to gain an understanding of the causes of an error or at least to note where the implementation and the expected behavior differ. In attempting to attain this understanding (which is the difficult part of debugging) a debugger usually abstracts parts of the program's activity to match parts of the understood system model. Through experiments that perturb parts of the system in a controlled fashion, the debugger is attempting to verify that the implementation, the abstractions and the model fit together in a meaningful way.

Current debugging technology can be characterized by the following attributes, which are all related to the process of model building:

*Supported in part by the National Science Foundation under Grant MCS-8006327 and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041.

**Supported in part by the National Aeronautics and Space Administration under grant NAG1-115.

- It is state-based — errors are considered to be the result of one or many incorrect state transitions during a program's execution. These errorful transitions are ultimately the result of incorrect operator applications or bad data. Finding the improper transitions should indicate the sources of the errorful behavior.

- The techniques used to monitor and experiment with the transitions from state to state are unit-at-a-time. Individual variables and flow between program statements are monitored and the results obtained serve as the behavioral model to compare to the assumed model. The debugger, although possibly aided by various bookkeeping tools, must create this model and perform the comparison. The quality of the debugging will depend on how much complexity the debugger can master.

- In order to effect this unit-at-a-time technique, the notion of absolute control over the state of a program has become central. Debugging tools generally allow breakpoints to be set and/or dumps of various pieces of state information to be made at points that are believed to correspond to crucial transitions in a program's execution. Implicit in this notion is the assumed availability of the entire state of a computation. Further, most debugging monitors allow a debugger to alter this state (in virtually uncontrolled ways) to provide the experimentation capability required for successful debugging.

Debugging in a Distributed Environment

Many of the desirable properties of distributed programming [3,5] make these attributes much less applicable to the debugging of programs written in a distributed fashion. Exercise of absolute control is inhibited by several properties of distributed programming. The high degree of autonomy of processes and processors and the sharing of the system resources in a distributed system imply that simply stopping an arbitrary process to examine its state may not be feasible. First of all, ordering a process to suspend its operation is a violation of its autonomy. Also, with server processes acting on behalf of other computations in the system, stopping a shared process will affect the autonomy of a (potentially large) number of processes. If too much control is exerted, the possibility exists of creating an 'artificial' system in which debugging is taking place. The end result of this is that the system being debugged will only approximate the system under which errors were detected and it might become difficult to correctly identify error sources. While the value of a system emulator or simulation cannot be completely discounted, errors due to timing mismatches or contention for resources cannot be effectively investigated in the simulated environment.

A further disadvantage of attempting to provide absolute control over the system is the difficulty in synchronizing the application of the control. When debugging on a single processor, a simple breakpoint instruction will usually freeze the process to permit examination of its state. The mechanisms involved in doing this for a distributed system are not so simple [4]. A small but possibly annoying aside to this is that it may be difficult to determine the whereabouts of a particular component of a distributed application, thereby making it difficult to direct attention to the component.

Abstraction based on unit-at-a-time techniques will almost certainly fail in distributed systems, if for no other reason than the large volume of information that it might be necessary to sort through to identify error sources. The complexity of the interactions of the components as well as the perceived interactions of unrelated components spread across a multitude of processing elements and their controlling software will likely overwhelm even the most skillful bug finder. In addition, for reasons related to the lack of absolute control, the information that is gathered from various units will very likely be stale or inaccurate. Finally, when a debugger has determined what units relate to some problem behavior and gathered information about them, the question of consistency among the units becomes important.

With the physical distribution of a system, a number of new ways to induce errorful behavior in programs are possible. Truly concurrent activity occurring in several processing elements of the system is an error source that state-based thinking cannot adequately address. Traditional synchronization problems are generally concerned with access to common storage locations. In distributed systems, synchronization problems are also related to access to the communication media and critical timing relationships imposed on processing among cooperating processes [2]. Moreover, because of the complexity of the collection of state transitions that represent an execution of a distributed program and the difficulty in synchronizing the updating of a state vector of all the components related to this execution, the concept of total system state has no meaningful interpretation for a distributed system.

BEHAVIORAL ABSTRACTION AS AN APPROACH TO DEBUGGING

Since one of the fundamental things done while debugging is abstraction, it seems natural to design debugging tools that will aid this process. Methods and tools based on behavioral abstraction [1] provide a foundation for viewing a system in terms of its behavior rather than its state. Behavioral abstraction relies on viewing a system as a generator of a stream of events that represent significant behaviors. Basic system activity is represented by a set of primitive events which are the finest grain of relevant system activity visible to a debugger. More complex behaviors are considered to be the results of interactions or combinations of the primitive events.

The use of behavioral abstraction for debugging is fundamentally different from other debugging techniques. A debugging monitor based on behavioral abstraction will allow a user to describe models of system behavior in terms of system activities and will then compare these abstractions with the actual behavior of the system. The choice of primitive events determines the lowest level of system activity that it is possible to observe and may also suggest a particular viewpoint on the system. Behavioral abstraction, however, permits the debugger to define alternative, higher-level viewpoints on the system and then to observe the system's behavior from those alternative perspectives.

In our approach to distributed system debugging, two techniques, filtering and clustering, are used to give a debugger the ability to define abstractions over the event stream as an aid in attaining an understanding of the system's behavior. Clustering coalesces a designated sequence of events into a single higher-level event and provides the principal abstraction mechanism of our approach. Such higher-level events can then be considered as part of the event stream and can be used as constituents in still higher-level clustering operations. Filtering the event stream has the effect of removing selected event instances from consideration as constituents of higher-level events. This aids the debugger in focusing on those events that are relevant to a particular viewpoint on system activity.

Filtering and clustering greatly increase a debugger's ability to project a model onto a system's activity. This contrasts sharply with traditional methods in which the user must first determine which among a (generally large) set of state variables are relevant to the pertinent system model, then observe their behavior, and subsequently attempt to integrate these observations into abstractions of system activity suitable for comparison to the assumed system model.

The Event Definition Language (EDL)

The Event Definition Language [1] was created to aid a user in describing event-based abstractions. An EDL event definition describes how an instance of an event might occur and what the attributes of the instance will be if it does occur. Means are provided for specifying both filtering and clustering operations on the event stream generated by a system. Clustering is accomplished in EDL by indicating a set of events and the sequencing relations among these events that will constitute a higher level event. This specification is known as the definition's event expression. In addition to providing the principal abstraction mechanism of EDL, the event expression leads to a coarse filtering of the event stream since only event types that are considered relevant to the abstraction are mentioned in the expression. A finer filtering based on the attributes of the events that are the constituents of the event expression is possible by specifying relations among these attributes in a series of constraining expressions. To allow for flexible use of this type of filtering, EDL definitions may also designate the set of attributes to be associated with the higher level event being defined, specifying how values are to be bound to these attributes when an instance of the defined event occurs.

MONITORING SYSTEM BEHAVIOR

A significant part of a debugging system based on behavioral abstraction is having the ability to detect the occurrence of behaviors that match the abstractions. EDL provides a means for defining system abstractions in terms of events and event attributes. Rather than the bottom-up approach to abstraction employed by the unit-at-a-time methods, a behavioral abstraction based debugger describes significant system behavioral models to be observed by the debugging tools. The simplest version of a behavioral abstraction based debugging tool would simply compare the user models to the actual system behavior. More elaborate and helpful tools would be able to note differences between user models and system behavior and have a rudimentary advisory capability to aid the user in the search for the error sources. A system to support the necessary detection capabilities is being constructed based on the design to be described in this section. Figure 1 is a diagram of the monitor indicating the major components, their connections and the kinds of information that are exchanged by the components. It is intended that this debugging tool be capable of being distributed in the network to take advantage of the desirable properties of distributed computing, to exploit the conveniences of behavioral abstraction and to aid in providing a debugging tool that has limited impact on the system it is being used to debug.

Interface to the Monitored System

The monitoring system is reasonably independent of the system it monitors. Interface to the monitored system is accomplished through provision of three elements: a means of observing the event stream generated by the system; a description of the primitive events and their attributes contained in this event stream; and a means of making requests on and receiving information from the system being monitored. The first two elements are required for even the most rudimentary event based monitor while inclusion of the third interface can add a higher level of sophistication to the monitor and will aid in distribution of the debugging task.

Event Stream Considerations The form of the event instances observed by the monitor is considered to be a name plus a list of one or more attributes associated with the instance. For discussion purposes, an event instance is a tuple of the form:

<event_name> <attribute_1> ... <attribute_n>.

The event name distinguishes an event from other types of events generated in a system and the event attributes distinguish instances of events of the same class from each other. This simple active interface allows the application of the event based monitoring method to a broad range of systems. On a uniprocessor or tightly coupled multiprocessor, event generation could be handled through the normal signalling mechanism that is usually a part of the system. For a networked distributed processing system, the communication scheme can most likely have an additional message type added — in some cases the existing message traffic might be the event stream needed.

System Definition The primitive system events must be described to the monitor. Since the monitor only sees the system in terms of event names and attributes, a natural method is to use EDL definitions consisting only of an event name and a list of attribute specifications for each primitive system event. Supplying different sets of primitives for the same system is obviously an application of the filtering technique for specifying viewpoints. Definition of primitive events simply gives a basis for the construction of abstractions -- one node's primitive could be another node's abstraction. However, it must be kept in mind that some entity must actually supply the event instantiations. If the primitive event definitions for a system do not match some subset of what is actually generated by the system, the obvious problems will result.

Interaction with the System The level of interaction with the system that is possible will define the limits of the monitoring task's power. The simplest form of interaction with the monitored system is to simply observe the event stream. If the user has the ability to interact with the system, the monitor might be given the ability to request transmission of certain events and in turn respond to requests for recognition of behaviors from other nodes of a distributed system. An important aspect of distributing the recognition of behaviors in a distributed system is the ability to coordinate the activity of recognizers residing at key locations throughout the system.

When system interaction is available at a sufficiently sophisticated level, a number of the properties of distributed systems and behavioral abstraction may be exploited. In particular, abstracted events and partially recognized events may be passed among cooperating recognizers, thereby reducing the amount of overhead due to event traffic as well as aiding in the use of abstractions for higher level recognition tasks. Further, passing of abstractions and requests for abstracted events will aid in reducing the impact of a debugging tool on the system by allowing load sharing of the debugging task by the processors involved. Finally, the ability to deal in abstracted behaviors will help solve problems concerning the visiblity of certain behaviors due to network topological constraints or security considerations.

Components of the Monitor

The following description of the monitor components is intended only to show the extant parts of the monitor and how they interact, not their implementation details.

Event Compiler and Librarian The event compiler and event library create and hold the internal form for all event types that are known to the monitor. Monitor users can add system abstractions in terms of previously defined events using these components. Additionally, they can interrogate the library for details of these abstractions. The event recognizer accesses the library when it needs definitions to perform its recognizing task.

Behavior Monitor The behavior monitor is the chief machine intelligence in the system. The capabilities of this component can range from simply interacting with the user and passing direct requests to the recognizer to a quite sophisticated version that could digest system events and notice relations among events or help the user by noting inconsistencies between what the user is using for diagnosis and what the system is actually doing. Our current version of the behavior monitor lies between these two as its advisory capability is limited to accumulating statistics on user requests and requesting the status of recognition tasks from the event recognizer.

Interaction Handler and Event Receiver The interaction handler and event receiver are the monitor's interface to the running system. The event receiver is simply the hook into the system that filters system event messages from other, non-event message traffic and changes the resultant event stream into a form for the event recognizer to use. The interaction handler knows the formats for messages into and out from the system and thereby allows monitor components to make requests of the system.

Event Recognizer The event recognizer is the heart of the monitoring system. It accepts requests for detection of primitive and high-level events, makes requests on the event librarian for event definitions and watches the event stream for occurrences of these events. When a requested recognition takes place, the requestor is signalled and the recognized definition may be placed into the event stream for possible later use as a constituent in a further high-level recognition. Demands on the capabilities of the recognizer stem from two sources, the needs of the recognition algorithm and the needs of other components of the debugging tools using the recognizer. The latter are considered to be mainly requests for recognition of abstracted events and questions concerning the status of such requests. Responses to inquiries are easily derivable from structures involved in the recognizer's implementation and are not terribly difficult to deal with.

The most important capability needed by the recognizer is that of extracting from the event stream a string of events that matches an event expression representing an abstraction. The events that will match a particular abstraction are not required to be contiguous in the stream. Other events, as well as normal system message traffic, are present in the stream and the recognizer must filter this 'noise' from the stream.

Filtering based on the attributes of events is a principal aid to abstraction of behavior that must be supported by the recognizer. This filtering is expressed in a set of constraining expressions defined over the attributes possessed by an individual event and can eliminate many events in the stream from consideration as instances to be used in satisfaction of the event expression.

Finally, it is desired that the recognition algorithm itself be capable of being distributed in a distributed system or at least that the algorithm not frustrate attempts to distribute the debugging task around the system. The motivation for investigation of behavioral abstraction based debugging has resulted from recognizing the potential advantages of writing programs distributed across a distributed computing system [3,5]. It is envisioned that the debugging tools developed for this type of work can also benefit from these advantages in performing their functions.

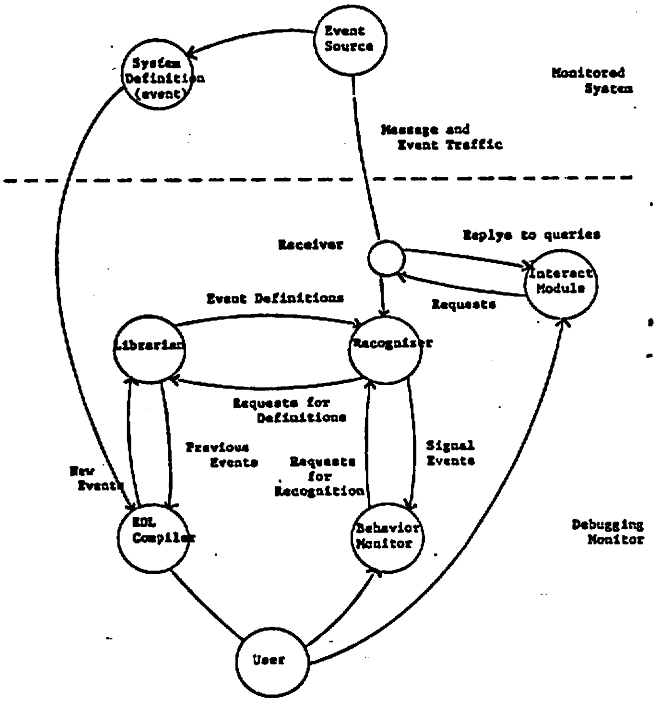

Figure 1 - Components and Flow of Information in the Tool

CONCLUSION AND FUTURE WORK

We feel that behavioral abstraction based techniques for debugging may be applied to any type of system and will contribute to controlling problems related to complexity, concurrency and decentralized control introduced by distributed computing environments. Our prototype debugger is under construction and will be used on a local area network that is being assembled in our department. One ongoing area of research is the construction of powerful and efficient algorithms for detecting abstracted behaviors in the system being monitored.

The problem of experimentation, mentioned in the first section, has yet to be adequately addressed. Experiments performed by a debugger on parts of a software component often provide the key to detecting the causes of errorful behaviors. The ability to exploit this may also prove valuable in behavioral abstraction based debugging monitors.

REFERENCES

[1] Peter C. Bates and Jack C. Wileden, "EDL: A Basis For Distributed System Debugging Tools," Proceedings of the Fifteenth Hawaii International Conference on System Sciences, (1982) pp.86-93.

[2] Jeremy Dion, "Reliable Storage in a Local Network," Technical Report 16, Computer Laboratory, University of Cambridge, February, 1981

[3] Philip H. Enslow, "What is a 'Distributed' Data Processing System", IEEE Computer, Vol. 11, no. 1, pp. 13-21, Jan. 1978

[4] A.W. Holt and F. Commoner, "Events and Conditions", Record Project MAC: Conference on Concurrent Systems and Parallel Computation ACM, Dec. 1970

[5] Victor R. Lesser and Daniel D. Corkill, "Functionally Accurate, Cooperative Distributed Systems," IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-11, no. 1, pp. 81-96, Jan. 1981.

[6] E.H. Satterthwaite, "Source Language Debugging Tools," Technical Report STAN-CS-75-494, Computer Science Department, Stanford University, 1975.

[7] Robert D. Schiffenbauer, "Interactive Debugging In a Distributed Computational Environment," Technical Report, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-264, September 1981

[8] Jacob T. Schwartz, "An Overview of Bugs," in Randall Rustin (ed.), Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, Prentice-Hall, 1971, pp. 1-16

[9] D.C. Swinehart, "COPILOT: A Multiple Process Approach to Interactive Programming Systems," Technical Report, Stanford University, STAN-CS-74-412, July 1974

[10] D. Van Tassel, Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Englewood Cliffs, New Jersey, 1978