Distributed Debugging Tools for Heterogeneous Distributed Systems

Peter Bates

Laboratory for Distributed Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

This paper describes a collection of tools that form an implementation of Event Based Behavioral Abstraction (EBBA), a paradigm for high-level debugging of distributed systems. The tools are capable of operating effectively in a heterogeneous environment containing processors of varying design and power (DEC MicroVax, TI Explorer Lisp machines, and several Sequent multiprocessors). Toolset users construct libraries of behavior models and observe the behavior of the system through the models. The toolset is a collection of components that are collectively a distributed system for debugging distributed systems. The components can be combined in varying ways to provide levels of debugging service appropriate for the resources available at individual nodes.

1. Introduction

Software debugging is a process in which debugging tool users synthesize selected artifacts of program execution into models that reflect the actual behavior of a system under investigation. Models of actual system behavior are compared with models of expected behavior held by system users and designers in order to identify significant differences. This model building and comparison process serves to illuminate the errors in the system and suggest corrections or direct further investigation.

Debugging tools facilitate modelling by providing support for monitoring a system under study and experimenting with its future behavior. Monitoring is accomplished by inserting output-producing probes into the software to make the behavior of the system constituents visible and provide alternatives to standard system outputs. Experiments are performed through closely controlled execution of system constituents by the debugging tools at the direction of a user. Most commonly, a software component will be brought to an important point in its execution and then have some elements of its environment changed before proceeding.

Extant interactive debugging tools rely on two important features of sequential programs: their time invariant execution; and the availability of controllable, accurate total system state. Neither of these can be reliably provided by distributed systems. In order to use traditional tools to debug a system, users must guess what the incorrect behaviors are, determine which pieces of state information will best illustrate these incorrect behaviors, then devise a plan for obtaining this information. In distributed systems organizational and operational complexity of system components makes these tasks unwieldy.

1.1 Event Based Behavioral Abstraction

The tools described here form a distributed implementation of Event Based Behavioral Abstraction (EBBA) [BW83], [Bat86]. EBBA employs abstraction in a systematic way to manage complexity and help to organize the search for errors. Models reflecting user understanding of system behavior are developed in a top-down manner and compared to the actual system activity by toolset components. Tool users then explore the differences between their models and the behaviors to determine why the differences exist. When user models match actual system behavior, they have demonstrated an understanding of some aspect of the system and might need to shift viewpoints or focus more closely on suspected components. When user models fail to match, the tools attempt to characterize the differences. Previously developed models may be incorporated or altered to produce new behavior models, as well as being reused when new debugging situations arise.

Behaviors are expressed in terms of events that represent significant interactions of system components. An event instance that records the occurrence of an event is an encoding consisting of an event class name that places the event in a general category and a list of attribute values that distinguish the specific instance of the event from others in its class. Two attributes, time and location, characteristic of all events, are implicitly included in any event instance. All events, take the general tuple form

\[(\text{event-class } a_1, a_2, \ldots, \text{time location}).\]

The shape of the tuple for an event class is determined by its description using the notation of the Event Definition Language (EDL) [BW82] [Bat87].

Primitive events represent the lowest observable level of system behavior or characterize some particular aspect of a system's activity. The process control part of an operating system would have (among others) create-process, suspend-process, and resume-process as primitives. The same system seen through its file I/O subsystem might have open-file, create-file, etc. as primitives. Each collection of primitive events forms the basis for a new poni on that system. For example the concurrent Mandlebot Set [Mar82] generator (a running example, see appendix) generates a characteristic set of primitive events, some of which are
The event tuple described by the `e-TakePatch` description is

\[
(e\text{-}TakePatch \text{ id } patchidz \text{ time } location)
\]

The following tuple is an example of a `PatchAllocated` event:

\[
(e\text{-}TakePatch \text{id pid } patchidz)
\]

The event tuple described by the `e-TakePatch` description is

\[
(e\text{-}TakePatch \text{id pid patchidz } time \text{ location})
\]

The `e-TakePatch` event is the class. Instances of the class are distinguished by the values bound to the `id`, `patchidz`, `time`, and `location` attributes. Some instances of this class might be

\[
(e\text{-}TakePatch 3 26 \text{ pid 00.17 } \text{min.3})
\]

\[
(e\text{-}TakePatch 2 47 \text{ pid 01.03 } \text{min.2})
\]

High-level events represent user behavior models that attempt to explain some layered system component or some complex interaction of primitive system elements. High-level event models are expressed in terms of primitive or other high-level events. For example, the following is a high-level model that describes when a patch has been given to a worker, completed, and displayed.

\[
e\text{-PatchAllocated} \cdot e\text{-TakePatch} \cdot e\text{-DisplayPatch} \text{ cond}
\]

\[
e\text{-PatchAllocated}.\text{pid} = \text{pid};
e\text{-TakePatch}.\text{patchidz} = \text{pid};
e\text{-DisplayPatch}.\text{pid} = \text{pid}
\]

\[
\text{with}
\text{patchdone:integer := pid;}
\text{processid:integer := e\text{-}TakePatch.id}
\]

Here the high-level `mh` model is expressed in terms of three primitive events with a temporal ordering given by an expression. The `o` operator indicates that the event instances that match the event expression members must occur sequentially. Other operators are available that denote choice, concurrency, and iteration.

The relational expressions following the `cond` keyword denote filtering constraints applied to the attributes of potential model constituent events. These three expressions guarantee that the event instances that only the behavior for a specific patch be looked for. The expressions following the `with` keyword indicate how to bind values to event instance attributes. The template for event instances derived from the `mh` model description is

\[
(mh\text{ patchdone processid time location})
\]

with a representative instance such as

\[
(mh 26 3 00:00.56 \text{EventMonitor.1})
\]

**EBBA** is not an automatic error detecting/correcting system, nor is it intended to be used to form complete formal models of entire systems. The intent of the **EBBA** approach and the goal of the toolset is to permit a tool user to quickly and easily create small, succinct behavior models and investigate system behavior from that perspective.

### 1.2 Distributed System Debugging and **EBBA**

The programming model assumed here is one in which the procedural and data components of the computation are physically dispersed in a network of systems with diverse hardware and software architectures that attempt to provide specialized, efficient functions. The collection of components that form a computation must cooperate with one another to produce an overall computational effect. The software components operate asynchronously with respect to one another, and may be created, destroyed, and moved in response to local conditions or non-local directives. Communication among components is via message passing as well as transfer of control. Behavior models derived in traditional ways from such computationally sophisticated systems lack consistent structure, thus making understanding and comparison difficult.

Debugging distributed programs is a more complicated affair than that of sequential systems. In addition to the increased complexity of managing activity in distributed systems, distributed systems present difficulties for performing basic monitoring and experimentation activities. Timely access to distributed state, delivery of the selected state elements, experiments involving component synchronization, and uncertainties about temporal relations among distributed components are among the more obvious difficulties to be overcome.

**Distributed debugging** can mean that there is a centralized tool located at one node used to debug a program executing at a subset of system nodes or that a substantial portion of the debugging tool itself may be distributed along with the distributed program. Interactive debugging tools of the first type are fairly straightforward to provide for a distributed program by using existing traditional low-level debugging tools connected to an array of terminals (or equivalently, terminal emulator windows on a workstation). An important weakness of this approach is that it does not help to overcome system complexity or the increased user complexity of organizing the debugging task among heterogeneous system components. Also, except for the user at the central site there is no provision for the tools to coordinate their activity.

The ability to coordinate is the single important feature that separates a true distributed debugging tool from simply a debugging tool used on a distributed system. The behavior of distributed systems can be investigated most effectively by distributing the mechanisms responsible for observing and controlling system behavior. Distribution of monitoring and debugging
their status and to request services of other components. These messages are coded as expressions using the syntax of the ell extension language. Used in this way ell is a communication protocol and interpreted language that permits toolset components to be structured as a message-based system and provides a mechanism for users to extend the basic functionality provided by individual components. A message received at a node is executed to return a result or to cause side-effects that alter the environment. Since the execution context for an expression is determined by the recipient, the same message given to different parties can have different effects.

All toolset component inputs and outputs have been structured to use the ell expression form. Users can enter ell expressions to directly control the activity of a component if the component allows direct entry of messages from type-in windows. Likewise, user interfaces, based on mouse inputs and graphical display output, format messages as ell expressions to be executed by the component they represent. For example, if the user was interested in the occurrence of the mh event for a specific patch, the graphics-based behavior monitor interface would issue a message to the Event Recognizer similar to

```
(recognize "Patch-55" (mh 55)
' (reply "td1-mh" "completed"))
```

This message requests that the Event Recognizer observe the event stream for an instance of mh applied to patch 55 of the Mandelbrot set. When the model is recognized a message is sent to the model display interface for td1-mh that the model is completed.

In addition to the use for message-based communication between toolset components, programs may be written using ell constructs that allow function definition and global variable declarations bound to component data structures. Most common arithmetic, relational, bitwise, and character manipulation operations are supported as wired-in functions. Other functions implement common programming language constructs such as iteration and conditional execution. Each toolset component adds wired-in functions that permit access to externally important structures.

3. Remote Debugging of Distributed Systems

Remote debugging is implemented by placing a user and the set of debugging tools employed by the user at a single node of the distributed system. Each remote node that is participating in debugging tasks necessarily contains an agent to aid the central debugging tool. Each agent has tacit knowledge of its local environment and will respond to requests made by the central site. The node that contains the debugging tools may or may not be a participant in the distributed computation under investigation. The central node with which the user directly interacts provides a way to direct attention to a specific node among the various participants in the session. As the computation progresses, the tool user interacts with the programs in the computation through the central toolset and its remote agents.

As mentioned earlier, remote debugging facilities are easy to provide. Some attempts have been made to coordinate the use of remote debugging facilities such as [CW82] and [Sch81]. The primary drawbacks to the use of remote debugging are:

- latency associated with reading and interpreting information and effecting intervention activities often renders the information out of date and the intervention lacking the desired effect,
- the information moved tends to be low-level, hence large in volume and the request/reply protocol needed to obtain it can be a disproportionate amount of resources, and
- in an heterogeneous system the computation details change from node to node of the system creating difficulties in traditional debugging tools for obtaining a coherent view of the computation.

3.1 Simple Remote Debugging

Remote debugging within the EBBA framework is quite simple to provide and provides a level of service that is least disruptive to activity at a remote node. Connecting a component to the central debugging tools is accomplished by attaching the component to a runtime function library (libEBBA) that can locate the abstraction tools, format event instance records, and exchange events with the central tool (figure 2).

![Figure 2: Remote Debugging Components of EBBA](image)

The minimal arrangement of remote agents and central event monitor provides simple remote debugging. The toolset remote agents gather and send all locally observed event traffic, unfiltered, to the central site. The central site consists of a complete toolset (figure 1) which is responsible for recognition and abstraction of higher level behaviors based on the primitive events received from the remote agents. When the central event monitor detects a higher level event of interest to the debugging tool user, it informs the user (or other requestor) and optionally issues requests to relevant remote agents to intervene in the activity of the participating computing elements.

This is the classic form of remote debugging. This blind form of event reporting requires communication costs to be directly proportional to the volume of local primitive events. The latency to intervention is the time required for the message exchange plus whatever is necessary for the central abstraction node to perform its task. Simple remote debugging is useful because it is quite easy to provide and, if monitoring is central to uncovering errorful behavior, intervention latency is less an issue.
3.2 Primitive Event Collection and Posting

In order to implement the libEBBA paradigm, primitive and abstracted high-level events must be collected from the remote sites and distributed to suitable cooperating nodes in the system. Given the nature of computer systems it is generally not possible for an observer to sit passively to the side and note when events occur. Primitive event generation is an active process which consumes time in a system that may have genuine time and other resource constraints. Since libEBBA is intended to operate interactively, the timeliness of event reporting is important. Behavior investigation that answer user queries result in new queries for further information. For the new information to be used effectively requires that it be current. Primitive event generation must be accomplished with resource usage commensurate with the granularity of the system being observed.

The libEBBA component (figure 2) of the toolset provides the lowest level of interaction of debugger and system constituents. A system component that is to participate in debugging by providing events for abstraction tools must use libEBBA facilities (or some equivalent) for event reporting and intervention control. A connection establishment routine takes an event library, local identification, and network hostname to locate abstraction tools that will service the component. Event formatting and reporting routines provide a number of styles for creating primitive events to be sent to an abstraction node. The current implementation encodes all event tuples as readable text strings. This promotes system and data format independence at a cost to encode and decode each event instance record. Calls to an event reporting routine must be inserted at appropriate places in system components. Finally, routines provide the central tool with an ability to gain control of the attached component so as to effect any needed intervention or control exchange.

The need to explicitly attach libEBBA to a system constituent raises questions regarding the ease of use of the toolset. Most debugging tools must be explicitly added to a software component as it is constructed. Symbol tables, initialization routines, debugging command interpreters, etc. are routinely included by program building programs such as compilers and linkage editors. This aspect is no different for the libEBBA toolset.

Of primary importance is capturing the primitive events that are really characteristic of the system. The rule of thumb is that each call or operation on implementation level routines and structures is a candidate for a primitive event. For example, system application programmers would require a set of primitive events resulting from invocation of basic system services, e.g. create-process or open-file. Primitive events are a level of system granularity. A high-level artificial intelligence program might define a level of primitive events related to access to a blackboard structure [Mod79, LC83].

How much disruption of normal system operation is caused by event generation is an important consideration. It is quite important that primitive events can be quickly generated and dispatched to an abstraction node of the toolset. How much of overall process execution time is consumed by event generation of course depends on how long a process runs and how frequently it reaches an event generation point.

The performance reported in table 1 can be taken as representative of a fixed amount of the time required for event generation. For timing purposes the events were created from a script containing appropriate information needed to generate an event stream. The column under PostEvent is the event rate achieved when events are formatted as text strings by libEBBA routines and sent to an event recognizer. The StreamEvent column reports rates for the same event messages that are already formed and need only be collected into a unit to be sent. The difference between the formatted PostEvent and StreamEvent is the formatting time required to create the event instance string. Using binary (no translation to text) strings is obviously optimal but data format changes and byte reordering in heterogeneous systems also carries a penalty.

These times compare favorably with those reported in [CHKM88]. The conclusion to be drawn is that event reporting rates are most affected by the capacity of the underlying communication protocols.

For a very high-level system such as DVMT [LC83] these costs (10-20 milliseconds per event) are not high. For investigating most classes of system applications these are acceptable, especially if the system contains no hard time constraints. However, for highly detailed (low-level events) analysis or real-time system systems these times are less acceptable. Either more selective reporting (filtering to reduce volume) or other means to capture the characteristic event stream is necessary (possibly a parallel event generating processor). Both of these techniques are under investigation.

3.3 Filtered Remote Debugging/Preset Actions

An improvement in the communication performance results if the remote agent is instructed to report only certain primitive events. This comes at a slight cost. The remote agent must be altered to hold an attention list containing event class names and supplied with appropriate table manipulation routines to maintain this list (figure 3). Each primitive event generated at the

Table 1: Event Creation & Movement Performance

<table>
<thead>
<tr>
<th>Component</th>
<th>libEBBA</th>
<th>Event Rate/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node1</td>
<td>text</td>
<td>62.53</td>
</tr>
<tr>
<td>Node3</td>
<td>text</td>
<td>62.53</td>
</tr>
<tr>
<td>I</td>
<td>text</td>
<td>62.53</td>
</tr>
<tr>
<td>I I I I I</td>
<td>text</td>
<td>62.53</td>
</tr>
<tr>
<td>118.8</td>
<td>118.8</td>
<td></td>
</tr>
<tr>
<td>120.27</td>
<td>120.27</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Remote Agent with Filtering or Preset Actions

remote agent's node is checked against the table and only those
Currently requested are sent out. This *filtered remote* debugging reduces the communication bandwidth requirements by removing event instances that would be filtered as unnecessary by the central tool. In the toolset implementation, local filtering incurs little additional overhead. Table checking is a cheap table lookup and changing the filtering status of an event is effected by a single message from the central tool.

With the addition of inspection to the remote agent's capabilities a new level of intervention service is provided. In simple remote debugging the physical separation of the originator and effecter of the intervention request can introduce delays which render the response too late for the desired effect. In order to improve on the latency, requests for intervention at a remote node are made in advance of the time they are to be carried out. The attention list is changed to accommodate (event-class, preset-action) pairs. When a local event instance is reported that matches the event-class part of one of the table pairs, the associated preset-action is executed. The event observed at the remote node that results in the intervention action is necessarily a primitive event since these are the only events available to the node.

As a simple example, to reduce the event traffic, all of the participating nodes could be instructed to only send and listen for *e_PatchAllocated* events. At the occurrence of the event for patch 50, the nodes would start to send all of the following *e_TakePatch* and *e_DisplayPatch* events. In this way, the abstraction node might only see the immediate context around which the mb at patch 50 occurs. The volume of primitive event traffic would be reduced by approximately two-thirds.

The preset actions technique is useful because all high-level behavior models ultimately are composed from primitive events. Thus a user requiring intervention activity based on recognition of a high-level model implicitly specifies the action in terms of primitive events. The difference, of course, is that a primitive event incorporated as a high level model constituent is a product of filtering and constraint satisfaction. With preset actions the latency to intervention has improved, but since the intervention occurs in response to a locally observed primitive event which has not undergone high-level filtering and constraint satisfaction, many unnecessary (and, depending on their nature, error causing) interventions might occur.

### 3.4 Task Distribution Through Simple Cooperative Debugging

A simple next step to further distribute the debugging task is to give each remote agent the ability to examine the network event stream. This extension allows remote nodes to initiate local debugging activity based on events occurring at other remote nodes. This limited listening capability implements simple cooperative debugging in which many nodes may be active participants in debugging activity. The remote agent now must listen to the communications medium for event traffic and extract the *event-class* fields of incoming event tuples to be used as keys to search the attention list. When a match occurs, any preset actions associated with the matched attention list entry are carried out. Now it is possible to effectively cause network-wide patterns of debugging activity to occur. Instead of being able to act only in response to locally observed events, groups of nodes may react to conditions that affect each other. The simple cooperative capability is useful where an intervention is required to affect program components at multiple nodes of a system. An example is an experiment where a tool user would like to synchronize distributed components upon the occurrence of an event at another node.

This exhausts the possibilities of remote debugging. Remote debugging is simple and does not consume a large amount of resources at remote nodes. However, the communication medium is heavily used for low-level event traffic. This potentially poses a problem where contention for the communication medium is affected by this traffic. Problems resulting from the latency to intervention are improved by adding simple event detection capabilities to each remote agent. In order to greatly reduce the communication requirements and provide more meaningful remote node intervention, it is necessary to perform local model abstraction and communicate higher level information among participants. The next section explores this approach.

### 4. Distributed Debugging with Distributed Event Recognition

Distributed debugging from the *EBBA* perspective is more than remote debugging of distributed programs. *EBBA*-based distributed debugging emphasizes model abstraction at remote nodes and exchange of resulting high-level events by participating nodes. While simple cooperative debugging, described in the previous section, forms a distributed program for debugging distributed programs which is quite powerful, it can still impose unacceptable levels of event message traffic and result in unwanted interventions in system activity. Benefits accrued from a more fully distributed event recognizer include:

- lowered communication bandwidth requirements due to exchange of only necessary or important events,
- improved intervention accuracy when intervention is based on local abstractions,
- load distribution of the processing required to effect debugging, and
- an ability to handle more general distributed system architectures that include gateways and subnets that are not fully connected.

Distributed *EBBA* high-level debugging nodes are capable of much autonomous activity and, once set in motion, may carry out a large portion of the monitoring and intervention activity necessary to understand a system error without requiring interaction by a tool user. Indeed, through the capabilities of the extension language (*elh*) quite complicated activities can be carried out by remote nodes.

#### 4.1 Centrally directed

The components of the remote agents at *EBBA* nodes that perform event abstraction are indicated in figure 4. Each component performs the same task as its central toolset counterpart. Missing are the components dedicated to viewpoint creation and maintenance: the Model Builder and event Librarian. The Model Builder only responds to user created behavior models so it only needs to reside at a node where a tool user might need access. This is fully in keeping with the *EBBA* caveat that debugging requires some user to direct the search for errors.
However, the services of the event Librarian are required by all nodes that perform behavior abstraction. Each node that is involved in high-level event recognition should have the same view of the system. The Librarian, which may be located at any single node or be a distributed component itself, acquires an additional connection to the network so that all high-level nodes may access its contents. The Librarian in effect becomes an event model server.

In the centrally directed use of the distributed toolset, coordination of debugging activity at a remote node is fully under control of the user acting through the central toolset. The tool user is responsible for partitioning the modelling tasks that are designed to uncover erroneous system behavior and then directing the appropriate remote nodes to work on their portion of the overall modelling activity. The remote nodes that have been assigned activities make arrangements with each other and the Librarian to obtain high-level events and exchange locally recognized event tuples required for effective cooperation. For example, a request for recognition of the following definition:

\[
\text{event } \text{Clustered} (\text{patchidz}: \text{integer}) \Rightarrow \\text{mh} (\text{patchidz}) [1] \Delta \text{mh} (\text{patchidz} + 1) [2] \\
\text{with} \\
\text{clusterid: integer} := \text{patchidz} \\
\text{end}
\]

requires that two high-level \text{mh} events must occur. The \(\Delta\) operator indicates concurrency, and means that all of its operand events must occur but their order is not important (their constituents may also be interleaved). The \text{Clustered} event requires that the two \text{mh} events occur on succeeding patches.

(\text{recognize "paired-patch" (Clustered 55)})

would ask for occurrence of the \text{mh} events on patches 55 and 56. The three high-level events (\text{Clustered} and two \text{mh}) could all be parceled out to different recognizers, to lessen the abstraction burden or as a response to the precursor \text{e.PatchAllocated} at the nodes that are doing the work.

The centrally directed, distributed use of the EBBA toolset is the limit of its capability in the current implementation. Further enhancements that more fully automate searches for erroneous behaviors are best covered by artificial intelligence techniques beyond the scope of this research.

4.2 Cooperative Debugging – future work

The central theme to extending distributed debugging with artificial intelligence techniques is to apply more of the information that is available to the modelling process. The goal is to improve the accuracy and relevance of that process. We are not looking for an automatic debugger which, given an errorful program, indicates where its erroneous behaviors originate and what needs to be done to correct them. Instead, the assistance envisioned would be of several kinds:

- Explanatory aids, which could give extensive analysis of the difference between user behavior models and actual system activity, and
- Speculation aids, which would take notice of user goals and information accessing trends to try out models derived from this information. The objective is to provide a suggestive role for the tools by filling in areas of a model that the user may have overlooked; or the tools might work on variations on the models the user has specified.

The partitioned design of the toolset components allows easy integration of these kinds of aids into the behavior monitor component.

Various techniques suggest themselves to assist in providing these tools. One purpose of event libraries is to encourage reuse of abstractions. Sophisticated user aids might extend reuse by structuring strategies and plans for debugging complex problems around libraries and users’ prior experience with similar situations. More detailed task decomposition, driven by the goal-directed nature of plans for debugging, could result in partial result exchange among cooperating distributed debugging nodes.

Supplying this more cooperative debugging environment will require much more intensive use of system computational resources. The need for and use of these higher order tools will naturally need to be balanced against their impact on the system being debugged and the subtleness or complexity of the errors undergoing investigation. It is seldom advisable to crack an egg with a pile driver.

5. Summary and Status

This paper contains a description of the EBBA toolset as a distributed program. It was argued that by working tradeoffs between remote information processing and communication, an EBBA-based distributed debugging toolset easily and naturally provides a range of solutions to monitoring and intervention in a distributed system. Complex, heterogeneous, or arbitrarily structured network architectures are accommodated easily because of the uniform view of system activity provided by events and the ability of the distributed EBBA tools to operate on high-level abstractions of behavior. The increased distribution of abstraction capabilities helps EBBA to overcome inaccuracies in debugging activity that result from physical distribution of the computation undergoing investigation.
The toolset currently runs on an Ethernet-based local network consisting of a large number of DEC VAXstations, TI Explorer Lisp machines, a few IBM RT/PC workstations, and two large Sequent Symmetry systems. All of the systems are capable of serving as primitive event generators and the abstraction components of the toolset runs on all but the Lisp machines (due to implementation differences).

Debugging "practice" has been performed on several canonical software systems to refine the underlying EBBA algorithms and explore the needs for user interface to the tools. These systems span the collection of local heterogeneous processors. In addition to the DVMT project, the toolset is being integrated with the Gutenberg system [CRSV86] and its use is being explored for several other local projects. The Belvedere project [HC87] has employed EBBA in their animation schemes for highly-parallel, non-shared memory architectures (and uncovered unexpected errors).

Future work on the Behavioral Abstraction paradigm includes developing techniques for automated model analysis and creating models at lower levels of abstraction. Model analysis is approached largely as finding differences in behaviors using error correcting parsing [APT72] and error correcting tree automata [LF78] techniques. Performance monitoring displays using EBBA abstraction levels are being developed both to enhance the toolset and be used for local network performance analysis. EBBA began as a high-level debugging technique but now methods are being sought to automate primitive event generation for program (or small system) instrumentation with the view towards augmenting traditional tools with the abstraction and organizational methods of EBBA.

REFERENCES


Appendix

The concurrent Mandelbrot Set generator is a program that generates and displays quadratic Mandelbrot sets in a parallel manner. The selected region of the complex plane is partitioned into equal sized patches that form the basic unit of work. A collection of processors to compute the set are assembled by an initial process. As each available processor becomes idle, it requests the description of a patch that needs to be done, performs the Mandelbrot function on each pixel of the patch, and then sends the completed patch to a display node.
Interactive debugging requires that the programmer be able to halt a program at interesting points in its execution. This paper presents a definition of distributed breakpoints with an algorithm for implementing the detection of these breakpoints, and presents an algorithm for halting a distributed program in a consistent state. The definition of distributed breakpoints is based on those events that can be detected in a distributed system. Events that can be partially ordered are detectable and form the basis for the breakpoint predicates, and from the breakpoint definition comes the description of an algorithm that can be used in a distributed debugger to detect these breakpoints. The Halting Algorithm extends Chandy and Lamport's algorithm for recording global state and solves the problem of processes that are not fully connected or frequently communicating.

1. Introduction

Interactive debugging requires that the programmer be able to halt a program at interesting points in its execution. Halting consists of the mechanisms to stop the program's execution and the predicates, called breakpoints, that are used to trigger the halting. This paper presents a definition of distributed breakpoints with an algorithm for implementing these breakpoints, and presents an algorithm for halting a distributed program in a consistent state.

Breakpoints in a sequential program have an implied reference to time. When we say "stop when procedure X is entered or when procedure Y is entered", we mean to stop the program when any of these conditions becomes true. When we say "stop when procedure X is entered and i[j]=7", we mean to stop the program when, at the same instant, both of these conditions are true.

We have no single, global notion of time in a distributed system [1], so we may not be able to determine whether one condition really occurred before another. This means that we will have to tolerate breakpoints that occur independently on different machines. Likewise, we cannot determine whether events on different machines occurred simultaneously. This means that we must replace the concept of simultaneous events with one that is suitable for a distributed system. In Section 2, we present a definition of predicates for breakpoints in a distributed program. This definition is based on detectable orderings of events. We describe an algorithm from which one can implement a satisfaction detector for these predicates.

Halting a single-process, sequential program is well-understood. There is a single thread of execution that can be stopped without regard for other activities in the system. When a program consists of cooperating processes executing on different machines, halting decisions are affected by unpredictable communication delays between machines. We cannot instantly transmit a command to halt all processes, nor can we guarantee that the halt command will simultaneously reach all processes. In Section 3 of this paper we present an algorithm for consistently halting a distributed program given the inherent communication delays. This algorithm is derived from Chandy and Lamport's algorithm for recording global state [2], and extends this algorithm to work for processes that communicate infrequently or are not fully connected.

Section 4 discusses the application of these ideas to current research in distributed debugging.

2. Distributed Breakpoints

This section describes a definition of predicates for breakpoints in a distributed program. This definition is based on detectable orderings of events.