# Distributed Debugging Tools for Heterogeneous Distributed Systems

Peter Bates

COINS Technical Report 87–28
March 1987

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

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332. And by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract NR049-041.

## Contents

**Distributed Debugging Tools**
**for**
**Heterogeneous Distributed Systems**

*ABSTRACT*

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 tools leads to a style of investigation that is high-level and potentially goal-oriented. These features lead to tools that are more easily managed by observers, return more accurate probe information, and are better able to continue to provide information during failures of individual components.

A distributed implementation of an *Event Based Behavioral Abstraction* toolset is developed here. The toolset itself is a collection of components that form a distributed system for debugging distributed systems. The components that comprise the toolset can be combined in varying ways to provide levels of debugging service appropriate for the resources available at individual nodes.

[illegible]

## 1. Introduction

This paper describes a collection of tools that implement a high-level debugger for distributed systems. The tools are capable of operating effectively in a heterogeneous environment which contains processors of varying design and power[1]. Debugging is viewed as 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 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 promote 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. To provide their basic services extant debugging tools rely on two important features of sequential programs: their time invariant execution; and the availability of controllable, accurate total system state. 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.

The tools described here form an implementation of *Event Based Behavioral Abstraction* (*EBBA*) [BW83] [Bat86], a paradigm for high-level debugging of distributed systems. The main operational features of the paradigm directly support construction of user models of system behavior and comparison of these models to actual system behavior. The behavioral differences that characterize errors are used to direct more focused investigation or to help identify corrections.

[1]The local environment consistes largely of DEC MicroVax, TI Explorer Lisp machines, and several Sequent multiprocessors.

Behaviors are expressed in terms of *events* that represent significant interactions of system components. *Primitive* events represent the lowest observable level of system behavior or characterize some particular aspect of a system's activity (e.g. I/O subsystem or process control). *High-level* events represent user behavior models that attempt to explain some layered system component. High-level models are expressed in terms of primitive or other high-level events.

Debugging distributed programs is a more complicated affair than that of sequential systems. 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 difficulties to be overcome. *Distributed debugging* can mean that there is a centralized tool for debugging distributed programs, or that a tool may be located at one node and be used to debug a program at another node, or that a substantial portion of the debugging tool itself may be distributed along with the distributed program. Most extant interactive debugging tools for distributed systems are of the first two types.

The next section discusses some issues for distributed debugging tools while relating them to an overview of the *EBBA* approach and the debugging toolset. Subsequent sections describe the components of the distributed toolset and the ways they are combined to balance performance against information requirements.

## 2. Distributed System Debugging and *EBBA*

The model of programming assumed here is one in which the procedural and data components of the computation are physically dispersed in a computer network consisting of heterogeneous computational nodes. The collection of components that form a computation must cooperate with one another to produce an overall computational effect. Communication among components is via message passing as well as transfer of control. The components can operate asynchronously with respect to one another, and may be created, destroyed, and moved in response to local conditions or non-local directives. The communication medium is simply a transport mechanism for interprocess communication and is not assumed to be totally reliable.

Issues that are important for understanding distributed system behavior and for distributing the understanding of that behavior include:

- the ***granularity*** of information reported by probes and used by cooperating agents to understand behavior. This will affect the amount of communication resources required to move information as well as the level of detail contained in each report,
- the ***flexibility*** of the communication and control mechanisms in order to provide graceful upgrading of distributed functionality and provide graceful degradation of distributed monitoring components in the face of component failures. Users should be able to concentrate more "power" on areas that require intensive scrutiny,
- an ability to tailor the investigation tools to the needs of ***heterogeneous*** systems, in a systematic way. Modern distributed computation systems consist of collections of systems with different hardware and software architectures that attempt to provide specialized, efficient functions; behavior models derived from such systems lack consistent structure thus making understanding and comparison difficult.

Observing these issues can lead to engineering appropriate solutions to distributed monitoring systems that are system and application level ***independent***, provide powerful analysis ***tools***, and ***reduce overhead*** required for monitoring real-time systems.

An important result of extending *EBBA* as a distributed program is an ability to provide various levels of debugging service tailored to the capabilities and constraints of specific nodes in the distributed system. An ability to operate effectively with varying levels of abstraction granularity permits an *EBBA* distributed debugging tool to balance node performance, administrative and logical system partitioning, communication performance requirements, and logical program residency to achieve a level of tool/system interaction necessary to investigate and explain errorful behavior.

Intervention and experimentation strategies likewise have various levels of service. *EBBA* provides no guidelines for which individual system components are to be manipulated by experimentation activities because such activities and acceptable responses are extremely system and situation specific. However, increased distribution and cooperation of the event monitor components provides a means to improve the accuracy and usefulness of interventions that are to be performed.

While the work reported here is undertaken largely in support of debugging distributed systems, its underlying mechanisms for collecting and interpreting behavioral information, and applying control to such a system are applicable to any system that requires this style of component interaction. [McD77] [Sno84] [Sno82] [MMS85]

The basic *EBBA* model building and abstraction functions are provided by the toolset diagrammed in figure 1. The tool user interacts with the ***Model Builder*** component to construct behavior models of expected system behavior. The ***Librarian*** is charged with maintaining user-defined models and distributing these models to cooperating components of the debugging system, such as an event recognizer. Initially the description of a set of primitive events, those events that define the basic observable functionality of a system or a particular viewpoint on a system's activity, are supplied to form the basis of an event library. As the computation progresses and the user investigates behaviors, new or refined versions of existing high-level behavioral models are added to the library.

Observation of system activity is performed using the ***event recognizer*** to match the user models to events arriving on the event stream. The event recognizer contains a sophisticated pattern matching component that is capable of dealing with a mixed stream of high and low-level events, helping to resolve time ambiguities, and filter event "noise" as it fits events to user-defined behavior models. The primary output of the event recognizer is event instance records representing behavioral models successfully matched by the pattern matching component. Also output from the recognizer is a stream of primitive event instance records that mark the progress of the event recognizer as it fits user models to event stream instances. These event instances are characteristic of the recognizer execution and open the possiblity of monitoring the activity of the event recognizer itself.

These basic functional components together with a library of routines for event formatting and communication, and other routines that implement basic intervention capabilities are the basis for a distributed *EBBA* toolset. The components and other available tools can be combined in ways that permit exploitation of the properties of the system under investigation and observe constraints imposed on the tools by the system itself.

An important component of the toolset is the Lisp-like extension language, *elll*, that serves to bind together all toolset components. The *elll* is a communication protocol and interpreted language that permits toolset components to be structured as a message-based object-oriented system and provides a mechanism for users to extend the basic functionality provided by individual components. Primitive and high-level event instance records are formatted and exchanged as *elll*

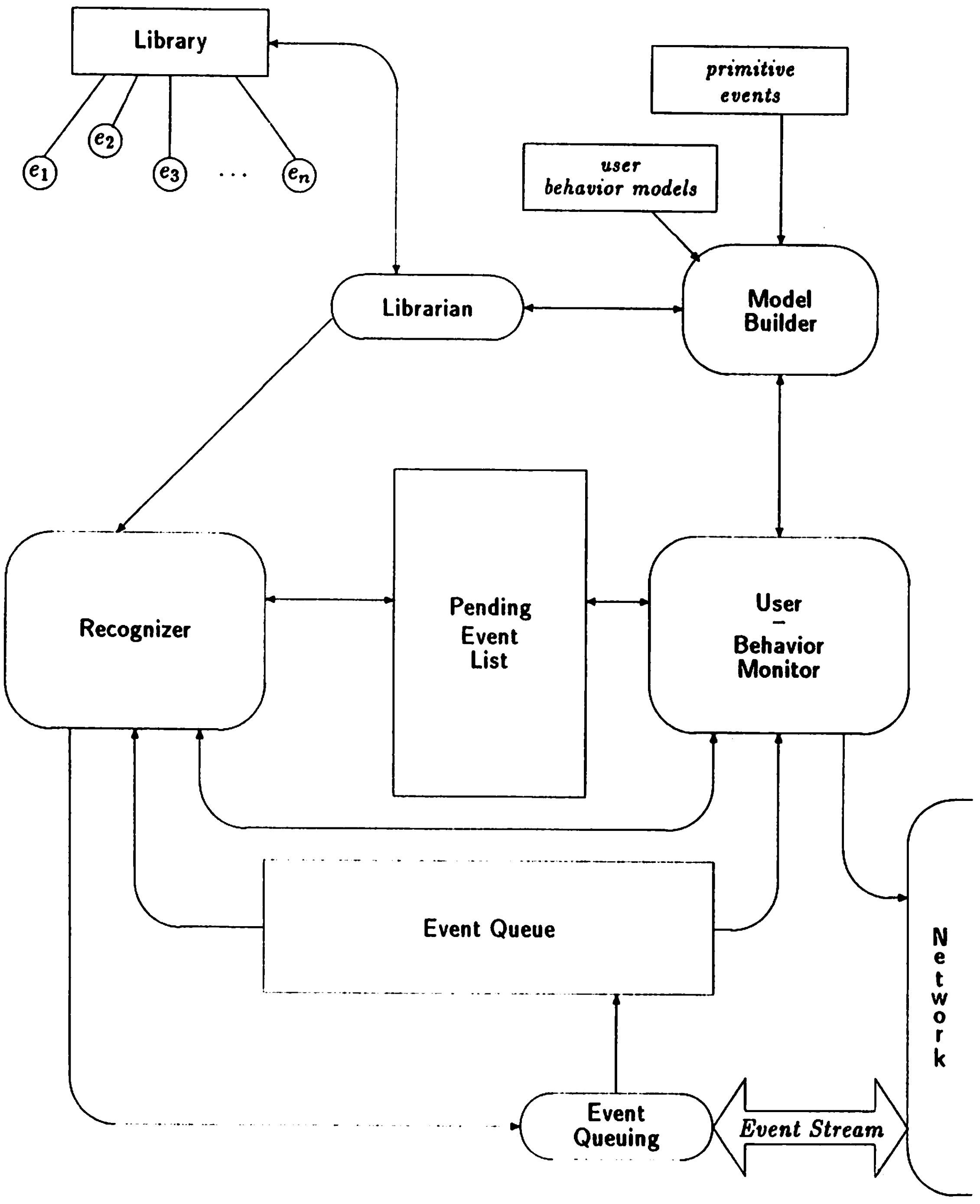

**Figure 1: Basic *EBBA* toolset**

messages.

Event exchange is the medium responsible for all system understanding in *EBBA*. An event instance record is an encoding of a tuple consisting of an event class name and a list of attribute values that characterize the specific instance of the event. Two attributes, ***time*** and ***location***, are characteristic of all events. Implicitly the last two attributes of any event tuple are the time the event instance was created and where it originates. All events take the general form

*(event-class $a_1$ $a_2$... time location).*

The attributes $a_1, a_2, \ldots$ correspond to attributes described in event attribute binding clauses of the modelling language.

Toolset components exchange messages to describe their status and to request services of other components. These messages are coded as expressions using the syntax of the *elll* extension language. All message exchange, both as event instance and for component control, is uniform but not indistinguishable. A message received at a node is executed to return a result or to cause side-effects that alter the local 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 to have the *elll* expression form[2]. Users can enter *elll* 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 *elll* expressions to be executed by the component they represent.

As well as being used for message-based communication between toolset components, programs may be written using the *elll* in order to extend the functionality of an individual toolset component. *elll* contains functions that allow function definition and global variable declarations. Most common arithmetic, relational, bitstring, and character manipulation operations are supported as wired-in functions. Others implement common programming language constructs such as iteration and conditional execution. Each component adds functions that permit access to externally important structures.

[2]With the exception of user-oriented interfaces.

## 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.

Remote debugging facilities are easy to provide. Indeed, debugging tools that implement some form of remote debugging are the predominant sort currently provided for distributed programs [CW82] [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, and
- because of the heterogeneous nature of the processing elements in a distributed system the computation details change from node to node of the system. In traditional debugging tools this creates difficulties for obtaining a coherent view of the computation.

Remote debugging within the *EBBA* framework is 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 library that can locate the tools, format event instance records, and exchange events with the central tool (figure 2). Using events as a medium of tool information exchange provides a uniform system view that isolates both users and distributed tool components from the vagaries of heterogeneous systems. By selecting an appropriate level of service for a node, latency can be managed through tradeoffs that move the use of information closer to the place where it originates.

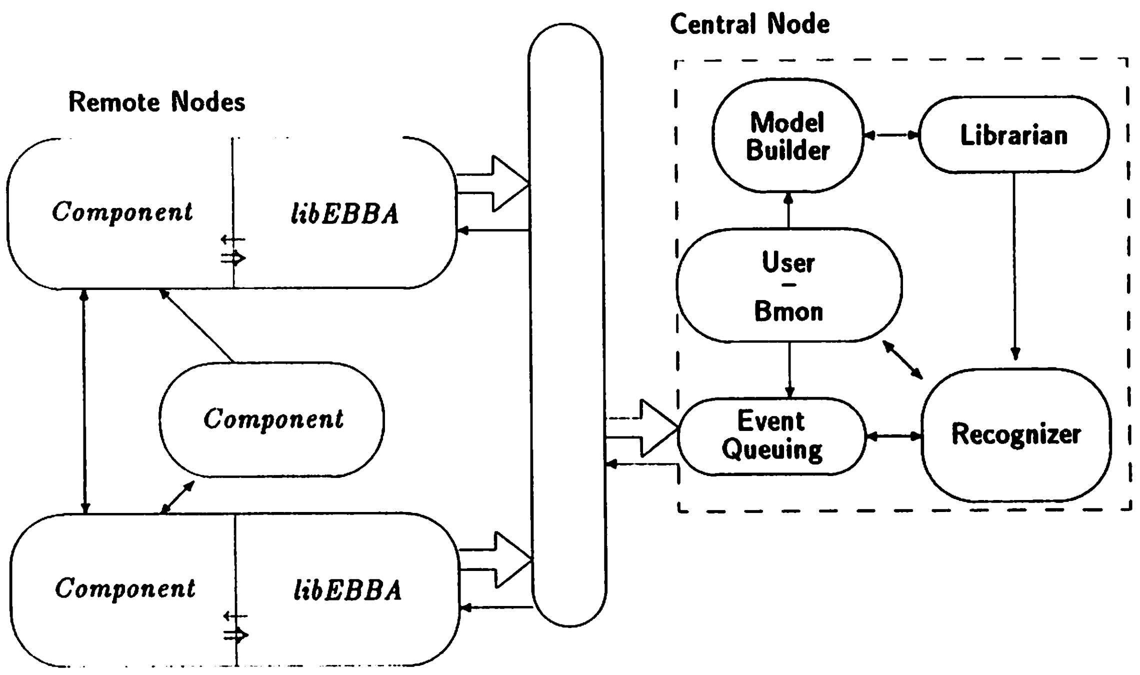

**Figure 2: Remote Debugging Components of *EBBA***

## 3.1 Primitive Event Collection & Posting in the Toolset

In order to implement the *EBBA* 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 *EBBA* is intended to operate interactively, the timeliness of event reporting is important. Behavior investigations 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. *libEBBA* contains three kinds of routines

- A connection establishment routine that takes an event library, local identification, and central tool location as arguments and attempts to find a central node that will service the component. If no location is specified, process environment variables are interrogated for a location. Failing that, the server database is queried to locate a default central node. Default values are likewise located for other arguments.
- 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 slight cost to encode and decode each event instance record. Calls to an event reporting routine must be inserted at appropriate places in system components.
- Intervention control routines provide the central tool with an ability to gain control of the attached component in as timely a fashion as possible.

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 *EBBA* toolset.

Of primary importance is capturing the characteristic primitive events of the system. The

| | No. of attributes | time to dispatch |
|---|---|---|
| *fork* | 2 | |
| *vfork* | 2 | |
| *execve* | 2 | |
| *exit* | 2 | |
| *wait* | 3 | |
| *wait3* | 4 | |
| *kill* | 3 | |
| *killpg* | 3 | |

**Table 1: Process control event creation performance**

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 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]. The event based view of a system attempts to explain *what* a system is doing, rather than *how* it is doing it.

Just how much disruption of normal system operation is caused by event generation is an important consideration. Primitive events can be quickly generated and dispatched to the central toolset. In a UNIX system with event generation included in the process control subsystem the performance reported in table 1 was obtained. How much of overall process execution time is consumed by event generation of course depends on how long a proces runs and how frequently it reaches an event generation point. The times reported in table 1 can be taken as representative of a fixed amount of time required for event generation. (time dominated by any one factor? linear in # attributes? is time and location stamping expensive?)

## 3.2 Simple Remote Debugging

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 for 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.

## 3.3 Filtered Remote Debugging and 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 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 simple binary search 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 effector 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-*

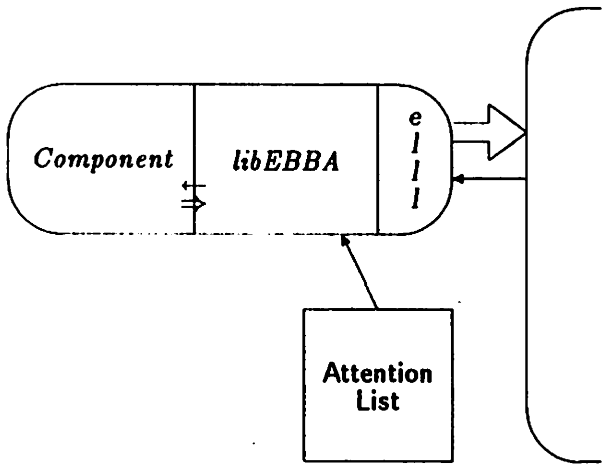

**Figure 3: Remote Agent with Filtering or Preset Actions**

*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.

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 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[3]) interventions might occur.

[3]Repeated, unnecessary interventions might disrupt timing relations more than is desired.

### 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 possiblities 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 (elll) 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 (from figure 1) 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. Therefore, the librarian should be the same one accessed by all nodes cooperating on recognition tasks. 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.

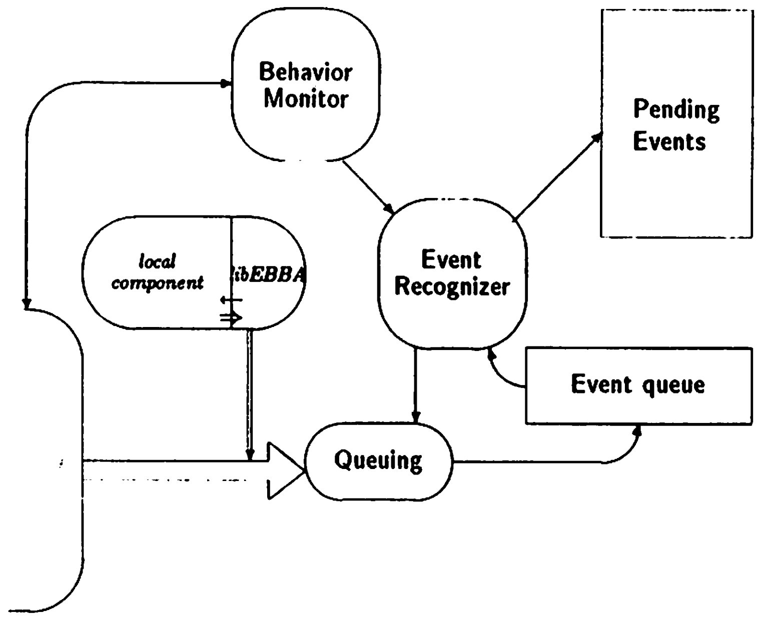

**Figure 4: Abstraction components of distributed toolset**

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 errorful 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 to obtain high-level events and exchange locally recognized event tuples required for effective cooperation. The cooperating nodes contact the librarian to obtain the definitions for the events they have been assigned to observe.

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

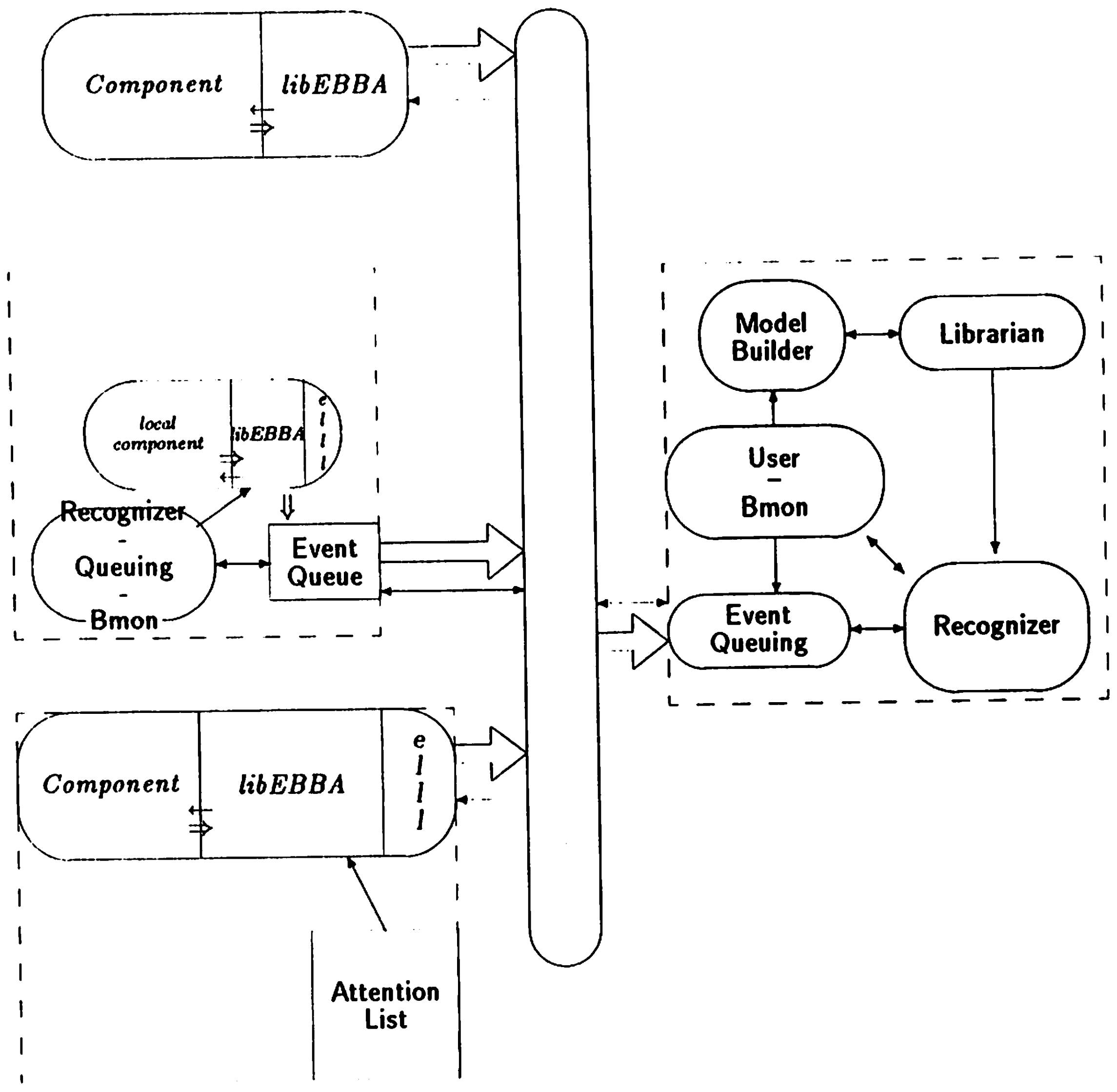

Figure 5: Heterogeneous Distributed Debugging Components

## 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 here 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

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 accomodated 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.

*EBBA*-based debugging treats debugging tool distribution as a metaphor for system organization. Tool distribution evolves naturally from a basic implementation of *EBBA* rather than as an afterthought riddled with special cases. Extension of *EBBA* as a distributed program can enhance tool transparency, reduce latency and uncertainty for monitoring and intervention, and take better advantage of computational power available in a network computer system.

## Acknowledgements

## REFERENCES

[Bat86] Peter C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, Univ. of Mass/Amherst, January 1986.

[BW83] Peter C. Bates and Jack C. Wileden. High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3(4), 1983.

[CW82] R. Curtis and L. Wittie. Bugnet: A Debugging System for Parallel Programming Environments. In *Third International Conference on Distributed Computing Systems*, pages 394–399, October 1982.

[LC83] V.R. Lesser and D.D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *AI Magazine*, 4(3):15–33, Fall 1983.

[McD77] G. McDaniel. Metric: A Kernel Instrumentation System for Distributed Environments. In *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 93–99, November 1977.

[MMS85] Barton P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley Unix. In *5th Internal Conference on Distributed Computing Systems*, IEEE Computer Society, May 1985. See also: Software-Practice & Experience, Vol 16 #2, Feb 1986.

[Mod79] M.L. Model. *Monitoring System Behavior in a Complex Computational Environment.* Technical Report CSL-79-1, XEROX Palo Alto Research Center, Palo Alto Ca, January 1979. Stanford University Computer Science Department/CS-79-701.

[Sch81] Robert D. Schiffenbauer. *Interactive Debugging in a Distributed Computational Environment.* Technical Report MIT/LCS/TR-264, MIT, September 1981.

[Sno82] R. Snodgrass. *Monitoring Distributed Systems: A Relational Approach.* PhD thesis, Carnegie-Mellon University, Computer Science Dept., December 1982.

[Sno84] R. Snodgrass. Monitoring in a Software Development Environment: A Relational Approach. In *Sigsoft/Sigplan Symposium on Practical Software Development Environments, Sigplan Notices*, April 1984.