Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior

PETER C. BATES
Bellcore

We describe a high-level debugging approach, Event-Based Behavioral Abstraction (EBBA), in which debugging is treated as a process of creating models of expected program behaviors and comparing these to the actual behaviors exhibited by the program. The use of EBBA techniques can enhance debugging-tool transparency, reduce latency and uncertainty for fundamental debugging activities, and accommodate diverse, heterogeneous architectures. Using events and behavior models as a basic mechanism provides a uniform view of heterogeneous systems and enables analysis to be performed in well-defined ways. Their use also enables EBBA users to extend and reuse knowledge gained in solving previous problems to new situations. We describe our behavior-modeling algorithm that matches actual behavior to models and automates many behavior analysis steps. The algorithm matches behavior in as many ways as possible and resolves these to return the best match to the user. It deals readily with partial behavior matches and incomplete information. In particular, we describe a tool set we have built. The tool set has been used to investigate the behavior of a wide range of programs. The tools are modular and can be distributed readily throughout a system.

Categories and Subject Descriptors: C.2.3 [**Computer-Communication Networks**]: Network Operations—*network monitoring*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.2.2 [**Software Engineering**]: Tools and Techniques—*Programmer workbench*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; *monitors*; *tracing*

General Terms: Algorithms, Design, Reliability

Additional Key Words and Phrases: Behavior modeling, debugging, events

1. INTRODUCTION

Software debugging is a process of *locating the causes* for known errors in a software system and *suggesting possible repairs*. Debugging distributed soft-

Much of this work was performed while the author was a member of the Computer and Information Science Department, University of Massachusetts at Amherst, and was 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.
Author's address: Bellcore, 445 South Street, Morristown, NJ 07962-1910;
email: pedro@thumper.bellcore.com.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1995 ACM 0734-2071/95/0200-0001 $03.50

ware systems is a complex and difficult process due to the general lack of adequate methods for debugging complex software as well as problems due to the special characteristics of distributed systems [Enslow 1978]. Among these are the availability of only incomplete or inaccurate state information, latency from request to action, and reduced ability to intrude on nonlocally executing system components.

This article describes *Event-Based Behavioral Abstraction* (EBBA), a high-level approach to debugging complex software. EBBA employs abstraction to manage complexity, to help organize the search for errors, and to enhance the operation of debugging tools for distributed systems. The technique requires that systems be instrumented to make their fundamental behaviors visible and that users employ a set of tools to develop and recognize more-abstract behaviors.

Behavior is activity that has observable effects in an executing system. Behavior may be viewed at various levels of detail and abstraction. For example, to an application programmer, assigning a value to a variable is a simple behavior, as is creating a new process. However, to the system programmer implementing the create process facility, this behavior is composed of many simpler behaviors. Software errors are generally caused by incorrectly designed or implemented behavior.

Our perspective on debugging is that it is largely a process of building *models* of expected or perceived program behavior and fitting actual behavior to these models. Showing how to transform the behavior model that matches actual behavior into a model that represents desired behavior suggests how the software might be changed so that it operates correctly. When using traditional tools to debug a program, users guess what the incorrect behavior is, determine which pieces of state information will best illustrate the incorrect behavior, and then devise plans for obtaining this information. Traditional tools are based on stopping a program at well-chosen points and examining its state. These techniques provide an overly detailed, uninterpreted view of program behavior with few aids to perform more comprehensive and meaningful behavior analyses, especially of interactions of multiple program components.

EBBA changes the emphasis on detailed state examination to one in which a user is presented with abstractions of program behavior and is provided mechanisms for structuring and manipulating those views. EBBA is a systematic approach to the modeling process that a tool user can use to focus on developing behavior models that explain program activity rather than directing activity at obtaining information necessary to verify the models. EBBA-based tools provide means to obtain actual behavioral information, compare the behavior to user models, and show the user how well the models fit the actual behavior.

EBBA employs abstraction in much the same way as the structured programming techniques [Dahl et al. 1972]. A behavior model is repeatedly decomposed and described in terms of its salient component behaviors until each component is expressed only in terms of fundamental behaviors. This formalizes the modeling process and helps to focus a tool user's attention on

significant behaviors. A tool user need only model areas suspected of containing errors in a depth- or breadth-as-needed manner, leaving areas of little value or with poorly understood interrelationships to be explored later.

The use of events as the basic information unit makes EBBA largely independent of target system implementation. Model reusability and modularity are important effects of this system-independent view. Behavior models can be reused easily or recast in new situations. For example, a user might keep a collection of models that describe deadlock situations. When deadlock is suspected as an error in a new system, the user could quickly recall these models and examine the new situation in terms of reliable, well-understood behavior models.

It is not intended that EBBA be used to create large, complex models of entire systems, since these tend to be themselves error prone. Rather, the goal is to permit a user to create succinct models, focused on suspect system behavior, in order to help understand erroneous behavior and its relationship to other system components.

The remainder of this section provides some background for this work. Section 2 explains the basic EBBA concepts and then presents the tool set implementation. Section 3 describes the behavior abstraction component and some of its capabilities in more detail. The last section summarizes, relates some of what has been learned from experience with the tool set, and briefly describes ongoing efforts.

1.2 Distributed Debugging

The programming model assumed here consists of multiple, heterogeneous hardware and software components that cooperate to solve a problem. They operate asynchronously, and the software components may be created, destroyed, and moved in response to local conditions or nonlocal directives. Traditional state-based debugging tools have several weaknesses when applied to this environment.

—Behavior models derived in traditional ways from computationally sophisticated heterogeneous systems lack consistent structure, which makes understanding and comparison difficult. The implementation of a particular operation (e.g., message exchange) may vary considerably from system to system, requiring users to deal with details beyond the scope of their investigation.

—Extant debugging tools rely on the time-invariant execution and availability of controllable, accurate system state found in sequential systems. There is little opportunity to access distributed system state accurately and consistently.

—The state-based tools do not help to manage system complexity or the increased burden of organizing the debugging task among multiple system components. Also, except for the user at the central site, there is no provision for the tools to coordinate their activity.

Distributed system debugging tools, such as EBBA, are distinguished by their integration with the systems on which they are used and by their ability to coordinate the distributed parts of the debugging tool. Attention to these characteristics 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. Distributing the mechanisms leads to tools that are more easily managed by observers, return more accurate and relevant information, and are better able to continue providing information during failures of individual components.

1.3 Related Work

There are some efforts that address the limitations of traditional state-based debugging tools [Bruegge 1983; Curtis and Wittie 1982; Garcia-Molina et al. 1981; Model 1979; Schiffenbauer 1981; Smith 1981; Sollins 1992; Weiser 1982]. A number of these concentrate on investigating specific aspects of complex systems as a means to overall understanding [Bruegge 1983; Schiffenbauer 1981; Smith 1981]. Model [1979] argues for high-level understanding tools and provides extensive display-based monitoring aids. Weiser [1979; 1982] proposes an attractive alternative by "slicing" only statements having bearing on a particular erroneous output, but still provides a narrow computation-level viewpoint that might be hard to extend to a distributed system. Some attempts have been made to coordinate the use of remote debugging facilities such as Curtis and Wittie [1982] and Schiffenbauer [1981]. Events are employed by Curtis and Wittie [1982] to trigger information-gathering and display functions and by Garcia-Molina [1981] to ask questions about what a system has done.

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 (such as McDaniel [1977], Miller et al. [1986], and Snodgrass [1982; 1984]).

A number of complex programs having many cooperating processes have been debugged solely using the tool set. See Bates [1989b] for a detailed description of the search for one such error. The Belvedere project [Hough and Cuny 1987] has employed EBBA in their animation schemes for highly parallel, nonshared memory architectures. Currently, there is an effort underway to instrument the software that operates the existing public-switched telephone network control systems in order to provide a uniform monitoring environment and use EBBA to diagnose software faults [Lai and Bates 1991].

2. EVENT-BASED BEHAVIORAL ABSTRACTION

EBBA models system activity in terms of the observable effects and interactions of program components. An *event* represents the occurrence of a significant behavior. As an instrumented program executes, it generates a sequence of events, called an *event stream*, that represents its activity. Users create

behavior models, expressed as relationships of different event types, to describe aspects of system activity associated with problematic behavior. Actual system behavior represented by events is matched to these models by components of the debugging tool set. The tools inform the user when behavior models are successfully matched to the event stream. Also, as the model-matching process continues, the tools can display differences between models and actual behavior. This ongoing user-model/tool-set-compare process is used as the basis for system investigations.

2.1 Events

There are two types of events used to represent the behavior of a program. Nondecomposed fundamental behaviors are represented by *primitive events*. Primitive events are created and delivered to the event stream by embedded instrumentation in system components. When a user-defined behavior model is successfully matched to the event stream, this derived behavior is abstracted into a representative *high-level event*. High-level events are created by the EBBA tool set and inserted into the event stream along with system-generated primitive events.

A specific kind of behavior is represented by a unique *event class*. Each event class describes a tuple consisting of the event's class name and a list of attributes that provides details about the behavior represented by the class. The general template for an event tuple is:

(event-class-name $a_1 a_2 \ldots a_n$ time location)

where $a_1, a_2, \ldots, a_n$, *time*, and *location* name the attributes possessed by the class, *event-class-name*. For example, the file input/output subsystem of an operating system seen by application programmers could create (among others) the following primitive-event classes:

(e_openFile *process_id filename fd time location*)

This event class represents a successful request to open a file. The *process_id* attribute is the system-assigned identifier of the requester. *Filename* is that of the file being opened; *fd* is the returned file identifier used by the program for further access.

(e_closeFile *process_id fd time location*)

This event class indicates that a process has closed a file. *Process_id* and *fd* are as described for e_openFile.

(e_readFile *process_id fd length time location*)

This event class is created when a process reads a portion of a file. *Process_id* identifies the requesting process, and *fd* identifies the file being read. The *length* parameter relates how much of the file was actually returned to the process.

An *event instance* is created by instrumentation to record the occurrence of the behavior represented by an event class. When an event instance is created, the instrumentation binds values to each of the attributes contained

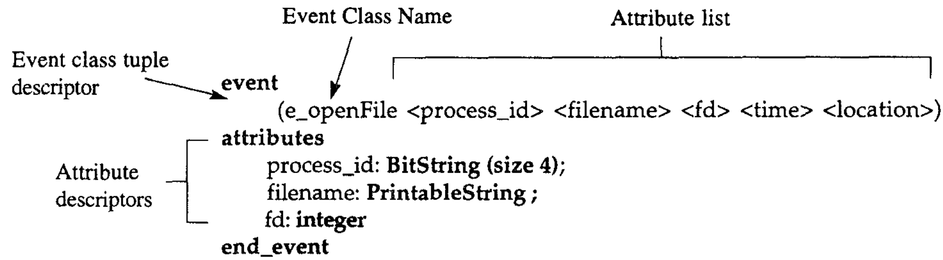


Fig. 1. Event class definition for e_openFile primitive event.

in the class tuple; the record is encoded; and inserted into the event stream. All events have *time* and *location* attributes that are derived from the environment in which the instrumented component runs. The value bound to the time attribute is the local view of time available when the event instance record is prepared. Location is usually the processor node at which the event has been created, although any convenient reference (e.g., program name) is possible. For example, the following event instances record the occurrence of some file system access behavior:

```
(e_openFile 119 "/etc/services" 5 14:30:23.49 "sluggo:xterm")
(e_readFile 119 5 1024 14:30:37.19 "sluggo:xterm")
(e_closeFile 119 5 17:04:22.08 "sluggo:xterm")
```

In these examples the location attribute was formed by concatenating the program name to the node name of the node where the program ran.

Primitive-event classes are defined using the *Event Definition Language* (EDL) [Bates 1987a; Bates and Wileden 1982]. These text descriptions are compiled by a tool set component (see Section 2.4.1) into the forms appropriate for their use by modeling, instrumentation, and analysis tools. Figure 1 is the EDL definition for the e_openFile primitive-event class.

The primitive events that describe a system are determined by the basic functional behavior of the system under investigation. In general, they do not change over the life of a system—unless the system changes to offer more or less function.

2.2 Using Events to Describe Behavior Models

Simply observing the event stream created by an instrumented system does not provide the modeling and abstraction capabilities we seek. This section will describe how we use the event classes defined for a system to create behavior models which can then be compared to actual behavior by the tools described in Section 2.4.

Modeling a behavior is accomplished by using EDL to specify a collection of event classes and expressing how the behaviors they represent are related. This description is compiled into some descriptive information and a set of tables that describe a procedure used to guide recognition of the behavior model. To recognize a behavior in a system, the EBBA tools match event

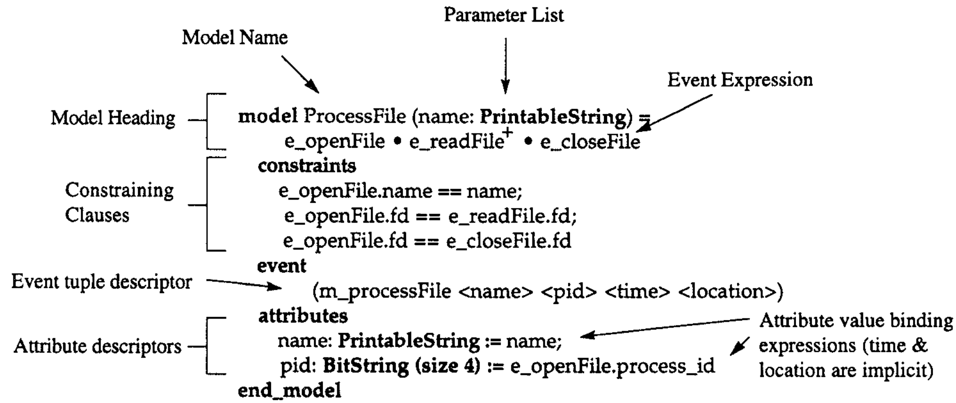


Fig. 2. Simple model describing file access.

instances arriving on the event stream to the corresponding event classes of the behavior model.

A behavior model description consists of two parts: the behavior specification and an optional event definition for event instances to be created when this model is matched to the event stream. Figure 2 is the EDL description for a behavior model describing a simple file-processing action. The behavior model, *ProcessFile*, is expressed in terms of the file I/O event classes described above.

Referring to Figure 2, the model heading specifies a name for the behavior model and a list of parameters used to tailor a model to a specific situation. Actual parameter values are supplied by the user when a request is made to locate instances of the behavior in the event stream. For example, using ProcessFile a user might want to be informed if a particular file had been accessed.

The *event expression* is a regular expression-like specification that names the constituent behaviors of the model and, using event operators, describes their acceptable orderings. The event expression is a notion similar to event expressions described in Riddle [1976], constrained expressions [Wileden [1978], and others described in Shaw [1980]. Event expression operators are available to express the following behavior patterns:

Sequential. Written •, indicates that event instances that match event expression members must follow each other in the event stream, although they are not required to be contiguous.

Choice. | specifies that an event instance must match any one of the alternative member events.

Concurrency. $\triangle$ indicates that instances that match its operand events may be arbitrarily interleaved in time. All operands connected with the concurrency operator must match an instance, but their order is unspecified.

Repetition. $^{+}$ or *, unary, postfix operators that specify iteration of their operand expression. * will match zero or more instances; $^{+}$ matches a series of one or more instances.

The constraining clauses of the description (introduced by constraints) specify a set of relational expressions to indicate what attribute values an event instance must possess to match its corresponding event expression member. In the absence of constraining-clause restrictions, an event expression member is matched by any event instance of the correct event class that satisfies the ordering expressed by the event expression. For the previous example, the first clause ensures that the behavior is related to a specific file (given by the name parameter); the remaining clauses ensure that all of the events refer to the same file in the program. The operand values used to evaluate the expressions are obtained from values supplied as parameters, attribute values bound to constituent event instances, or other values available in the environment (e.g., time of day).

When a behavior model is recognized in the event stream, this constitutes an abstracted behavior and can be represented by an event class in its own right. The event class is described by the optional event definition of the behavior model specification. The attribute descriptors associated with the event definition have an additional expression used to bind actual attribute values when the event instance is created. Operand values are obtained as for the constraining clauses.

2.3 Viewpoints

Debugging programs in a large or evolving system could be unwieldy because of the large number of primitive-event classes needed to describe its behavior. As an organizational aid, EBBA facilitates partitioning program behavior into *viewpoints* that are subsets of overall system behavior. A viewpoint is a collection of related primitive-event classes together with any behavior models based on those events. Viewpoints serve to narrow the initial focus of an investigation and allow tool users to gather only relevant behaviors to explore a problem. By *merging* viewpoints, a user can broaden the investigation to model interacting system components.

For example, an obvious partitioning for an operating system might separate the process control, file input/output, and interprocess communication subsystems. Models expressed solely in terms of a particular viewpoint would allow only a narrow set of behaviors to be modeled. For example, modeling only in terms of process control would allow a view of process creation, death, and other transitions, but tell little about what processes were doing. Merging other viewpoints such as interprocess communication and file I/O would provide ways to capture resource usage and interactions among processes to provide a more complete view of behavior.

2.4 The EBBA Tool Set

Support for EBBA requires a fairly sophisticated collection of tools. An implementation of the basic EBBA model-building and abstraction functions

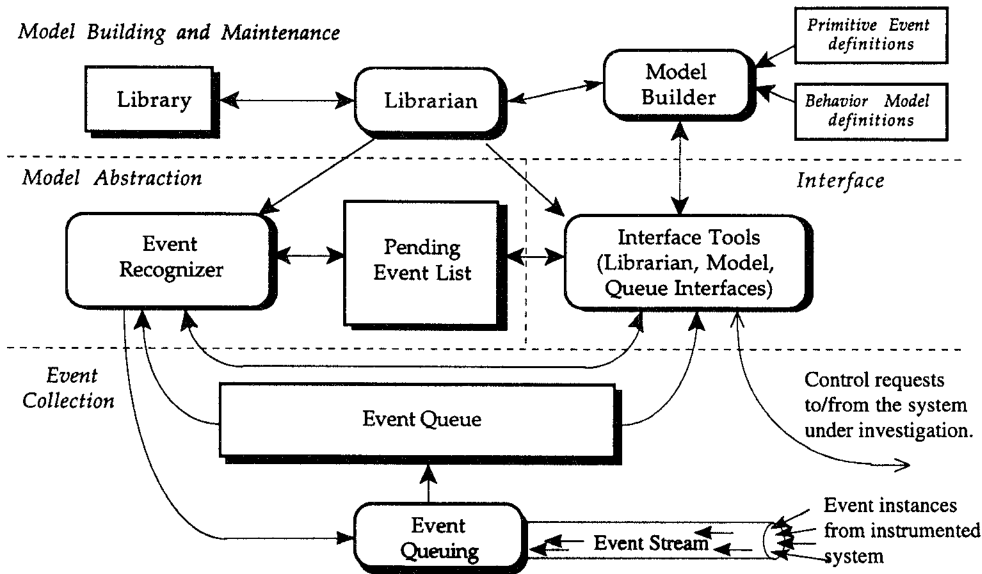


Fig. 3. Components of the basic EBBA tool set.

is provided by the tool set diagrammed in Figure 3 (rectangles are structured data; ovals represent active programs; arrows show information flow).

The tool set is structured as a collection of semiautonomous components that operate in a graphics workstation environment.[1] The system undergoing debugging can have both local and remote parts that supply events to the event stream. The tool set functions are organized into the following areas:

—*Model-building and maintenance* tools are used for compiling primitive-event descriptions and behavior models from EDL source text, storing events together that form viewpoints, and distributing compiled event descriptions to tool set components.

—*Model abstraction* accepts requests for recognition of behavior models, fits event instances arriving on the event stream into these models, and creates event instances that represent recognized high-level behaviors.

—*Event collection and communication* tools wait on the event stream to receive event instance messages and translate these into internal forms for the abstraction tools to use. In the system being examined, tool set components make connections to the event stream and generate and send primitive events.

—*Interface* tools present a graphics-based user interface to the other tools.

[1]This does not limit the techniques to such an environment, but we feel that graphical interfaces are important for interacting with complex systems.

To use the EBBA tool set to investigate the behavior of a system, it is necessary first to define the set of primitive-event classes that represent the basic functions of system components and then to instrument the components so they will generate event instances as the system executes. The primitive-event class definitions are prepared as EDL source text. The text is translated by the Model Builder into internal forms and stored by the event Librarian component into an *event library*. An event library contains all of the compiled event class descriptions related to a single viewpoint.

Instrumentation of the system being studied is aided by the Annotation Tool (Figure 5). This tool allows a user to browse the source text for a program component and insert event-instance-generating code fragments. The instrumented system can then be compiled and prepared for execution.

The tool set user starts the event Librarian and selects the event library containing an initial viewpoint to observe the program through. The library is opened by the Librarian, which acts as a server for the library contents. Behavior models can be described in EDL and added to a viewpoint either before or as the instrumented program executes. Next the user starts the Event Queuing and Event Recognizer components of the tool set. These components contact the Librarian to obtain descriptions of the event classes for the viewpoint in use.

The programs that comprise the instrumented system are started and locate the Queuing and Recognizer components of the tool set before they begin their normal execution. While the instrumented system executes, it sends event instance records into the Event Stream attached to the Queuing component. The user can request that the Recognizer match behavior models to these event instances. The tools provide feedback on the status of these model-matching attempts and execute user-specified actions when a model is completely recognized. The instrumented system can be stopped and started or otherwise manipulated using normal process controls. Event Queuing and the Recognizer can be kept in synchrony with the system through the provided interfaces by clearing the queue or restarting recognition requests.

Some of the tool set components—the Librarian, Recognizer, and Event Queuing—run continuously while a user investigates a problem. Others—the Model Builder and various interfaces—execute as they are needed. Event Libraries are stored permanently in a file system, the contents managed by a Librarian. The Event Queue and Pending Event List are transient structures created and maintained by the Event Queuing and Recognizer components, respectively. More-detailed descriptions of these components follow.

2.4.1 *Model-Building and Maintenance Tools.* An event library contains a set of structured data that gathers descriptions of the event classes and models comprising a viewpoint. For both primitive-event classes and behavior models there is a common set of information that describes the form and content of their event instances. Primitive-event class descriptions contain a list of references to instrumented components and copies of the attribute bindings for each. These are kept so that the code fragments can be modified or recreated if necessary.

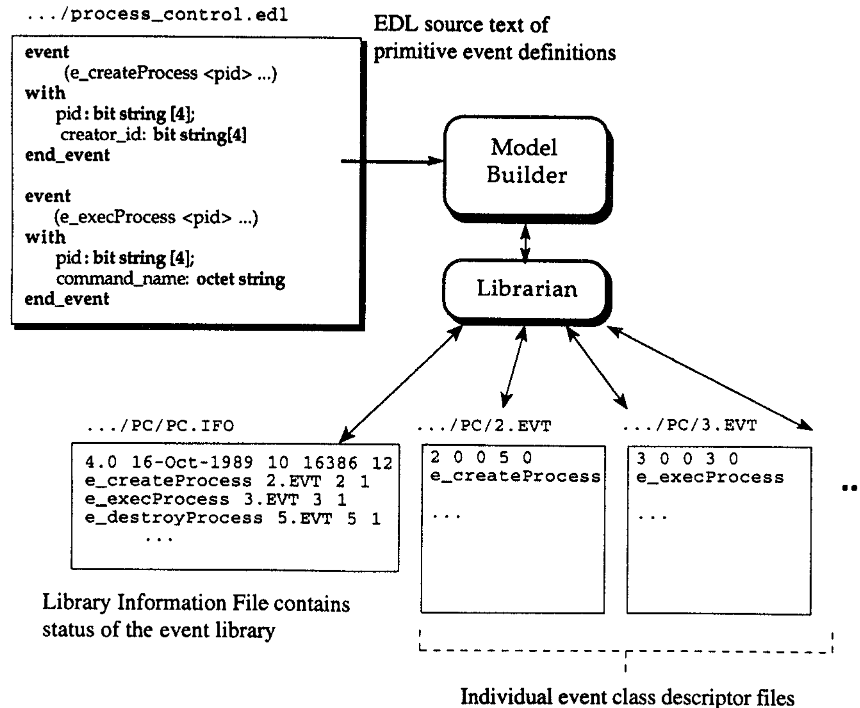


Fig. 4. Model Builder output for a primitive-event description.

Behavior models are more elaborate. They include the names of the constituent events of the model (named in the event expression) and event parameters, the description of the recognition procedure used to match the behavior model to the event stream, and code strings needed to evaluate the constraining (constraints clause) and attribute-binding (attributes clause) expressions.

The library contents are saved as a collection of files in a subdirectory. There are two kinds of files kept in a library. One kind holds translated event class descriptions. Each event class description is kept in its own file. The other kind of file is a distinguished "library information file." This file has a record containing the summary information about the library and a series of records that map each available event class to the file that stores the translated event class description.

Figure 4 illustrates the translation into library files of the definitions for several of the process control event classes. The EDL source text is in the file process_control.edl while the subdirectory that holds the library contents is labeled PC/. This subdirectory is created by the Librarian when the library is first accessed. Within the subdirectory, the Librarian creates PC.IFO as the library information file. The files holding the descriptions have names created from an internal identifier so that they can be made unique for the lifetime of a library (2.EVT, 3.EVT, etc.).

The Librarian has a graphics-based user interface that provides users with access to summary information for events already in the library and editors to create and modify behavior models. When a user wants to create a new behavior model a request is made through the Librarian interface. The Librarian starts an editor and invokes the Model Builder to compile and store the definition into the library. The Librarian helps maintain consistency between the new and existing event classes as the library changes.

The Librarian has the capability to merge a library into the one currently in use. A user might do this to expand the scope of an investigation into system behavior by defining behavior models in terms of the merged viewpoints. When the library changes because an individual model is added or modified, or the Librarian merges one library into another, the Librarian will notify all tool set components that are currently using the library. If interested, the components can request new versions of changed event classes.

When a user wants to monitor a program for occurrences of a behavior model, the request is initially made through the Librarian interface. The Librarian starts another graphics-based user interface to enable the user to interact with the Event Recognizer.

2.4.2 *Event Collection.* The Event Queuing component monitors the event stream and adds arriving event instances to the *Event Queue* structure. The event stream is formed from a set of message-passing interprocess communication connections established between instrumented system parts and the EBBA tool set. As a system being studied starts up, each part that generates primitive-event instances establishes a connection to the event queuer. Event instances generated in the system under study are sent on these connections as its execution continues.

When the event queuer is started, it requests descriptions from the Librarian of all the event classes that make up the viewpoint in use. The queuer uses these descriptions to decode the event instance messages that arrive in the event stream. When an event instance message arrives, its class is decoded to determine if it is part of the viewpoint in use. If so, the attributes are decoded, and an internal representation of the event instance is added to the end of the Event Queue. Event instances that are not defined in the viewpoint are simply discarded.

Rules for Event Queue management are simple. Event instances that are bound to behavior models are kept until the model is no longer of interest to the user. Other event instances are kept in the queue for as long as possible. There are several reasons for this. One is that a user may want to reevaluate a model by changing its parameters or constraints in some way. Another is that the user may want to evaluate a new model over past behavior. Past events are kept visible through a sliding window over the most recently arrived event instances. Event instances at the front of the queue are discarded when the window fills, unless they have been used to match a behavior model. The window size can be adjusted so that all event instances are retained or so that only events that are bound to behavior models are saved.

The tool set is intended to be used interactively as the system being debugged is executing. However, there is provision for storing the event stream so that it may be examined off-line. Alternatively, event streams stored off-line may be fed to the Recognizer.

2.4.3 *Model Abstraction.* The *Event Recognizer* component implements a pattern recognition algorithm that matches event instances appearing in the event queue to behavior models. After defining a behavior model with EDL and compiling this with the Model Builder, a user may request that the Recognizer monitor the event stream for occurrences of the behavior.

The request for recognition of a behavior model is made through the graphical interface started by the Librarian for the model. The interface requests the description of the model and all of its constituents from the Librarian, then displays the complete hierarchical structure of the behavior model. The user supplies event parameter values defined in the model heading and actions to be carried out upon recognition. Finally, the interface sends a recognition request containing this information to the Recognizer.

The Recognizer contacts the Librarian to obtain the description of the model and all of its constituent event classes. The Recognizer decomposes the hierarchy described by the behavior model into manageable units for recognition. A structure is created on the *Pending Event List* to hold the progress of the recognition request for each of these units.

Associated with each recognition unit is a set of actions to be carried out when the behavior represented by the unit is recognized. Some of these are specified by the tool user when the request for recognition is made. There is a wide range of possible actions. For example, the user might specify that, following recognition of a specific behavior model, the tool set send a message to part of the computation under study to suspend itself. Or, the user might specify that the tool set begin searching for another behavior model. This action mechanism is also used by the tool set to perform housekeeping activities. For example, a typical action attached to a recognition request invokes a routine to create an event instance to represent the behavior model just recognized and add it to the event stream.

2.4.4 *User Interface.* The *User-Behavior Monitor* component is a collection of graphics interfaces to the major tool set components. The Librarian interface is the most prominent. It displays a list of the events and models that form a viewpoint and several menus. The menus enable the user to invoke tools to create new behavior models or modify existing models, merge other libraries into the library in use, examine the details of a model or primitive event, and create the recognition interface to a behavior model.

An Event Queue display provides a summary of queue statistics, including an item for each event class in the current viewpoint. This interface permits a user to modify the sliding-window parameters and to clear or shorten the queue. It is also possible to request display of subsets of the event instances in the queue.

The other important interface is the display for individual models. It is started by a request through the Librarian interface. The user can use this

display to request recognition of the behavior and to control the recognition process (e.g., suspend part of it). The interface is also used to interrogate and display the progress made by the Recognizer in matching the model to event instances.

2.5 Primitive-Event Generation

Embedded instrumentation is responsible for generating the primitive-event instances that represent the fundamental behavior of the program under study. A desirable property of any instrumentation is that it is transparent to the normal execution of the instrumented software. However, completely transparent instrumentation requires hardware and software whose execution is totally independent of the monitored system.

The goals of the instrumentation used for EBBA are to minimize the memory requirements and execution time for each code fragment and to support an environment that promotes flexible binding of instrumentation to the monitored software. Completely satisfactory techniques for very flexible binding of instrumentation to programs or simple ways to activate/deactivate instrumentation are difficult to achieve. The main difficulty is that extant programming tools (compilers, linkers) are closed to external augmentation and do not allow access to the rich internal structures they create for a program. Symbol tables, initialization routines, debugging command interpreters, etc. are routinely included by program-building programs such as compilers and linkage editors to aid traditional debugging tools. Further access to structures such as control flow graphs or an ability to augment generated code with externally defined fragments (instrumentation) would simplify EBBA instrumentation.

The Annotation Tool (Figure 5) simplifies creation of instrumentation code fragments and works with the Librarian to maintain these for the user. This tools reads source code modules to be instrumented and the event library containing the primitive-events descriptions for the module, and accepts directives from a user to insert fragments into the source. Using a graphics interface, the user selects points in the source code that should generate primitive events. The event class to be inserted is selected, and a series of displays are presented so that the user can supply value-binding expressions for each attribute. This tool can be used to insert, delete, and modify instrumentation. Also, it is programmable, so that different programming languages and instrumentation styles can be accommodated.

There is a small library of support routines that support EBBA instrumentation. An event stream connection routine is called when the instrumented program begins execution. It locates the event stream and sends a message identifying the program and the event library that it uses. As the instrumented program executes, it encounters the event-generating code fragments. These invoke the library routine that formats and sends primitive-event instance messages to the event stream. Figure 5 illustrates the relationship of the tool set components that support primitive-event generation.

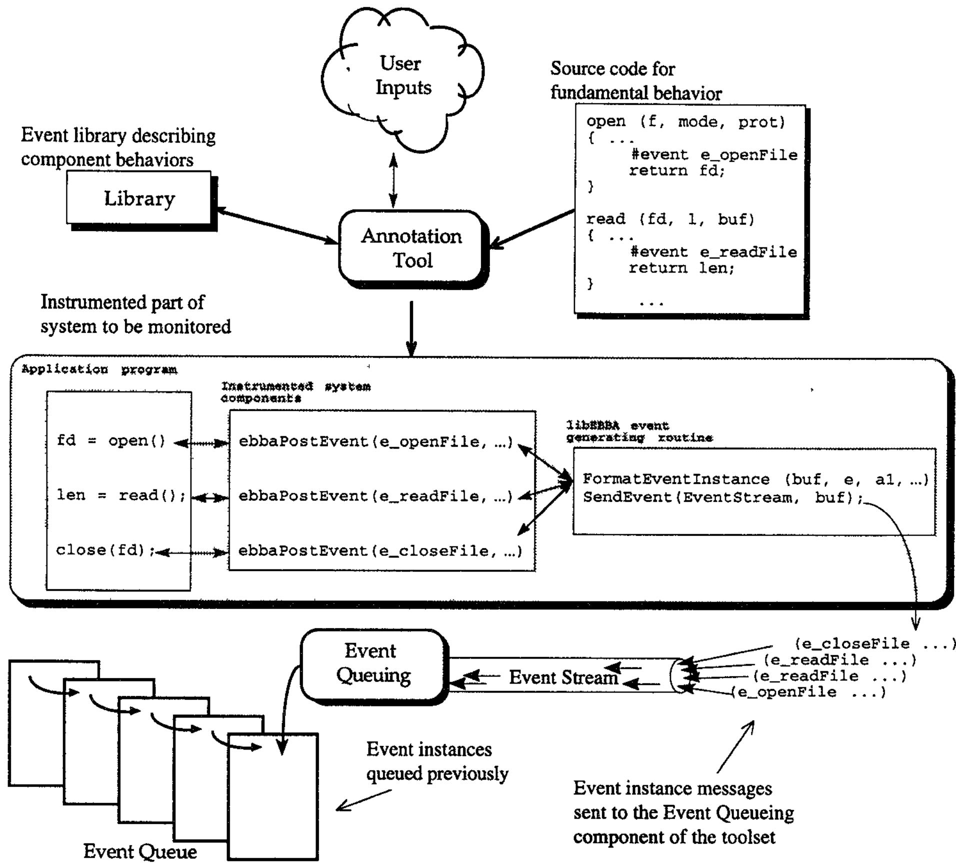


Fig. 5. Relation of primitive-event annotations to generated events.

2.5.1 *Choosing Primitive Events* Adding instrumentation to a software component is conceptually simple: identify the important transitions in a module and then add code fragments that create event instances representing these behaviors. Identification is generally easy, since it follows directly from the functions provided by a component.

We do not attempt to define statement-level behavior as primitive-event classes. The reasons are varied. This level of detail would generate an incredible volume of event instances for any reasonable program. Also, instrumenting the program would be arduous as the instrumented program would consist mostly of event-generating annotations. However, examining a program at this level of detail is important for finding simple programming errors, so work is ongoing that will help examine detailed behavior and stay within the EBBA framework.

2.5.2 *When is the Behavior a Primitive Event?* Our goal is then to identify coarser grains of program execution as primitive events. A number of program execution effects must be considered to ensure that the event instance

```
s_lock (lock)
   register slock_t *lock;
{
   S_LOCK (lock);
#ifdef EBBA
   EBBAPostEvent (e_LockAcquired, getpid(), lock, 0);
#endif EBBA
}
```

Sequent Parallel Processing Library [24] spin-lock routine

```
XPixmapPut(w, source_x, source_y, dest_x, dest_y,
                            width, height, pixmap, func, planes)
   ...
{
   ...
   GetReq(X_PixmapPut, w);
   req->mask = planes;
   req->func = func;
   req->params0 = height;
   req->params1 = width;
   req->params2 = source_x;
   req->params3 = source_y;
   req->param.l[2] = pixmap;
   req->params6 = dest_x;
   req->params7 = dest_y;
#ifdef EBBA
   EBBAPostEvent(e_XPixmapPut, dpy->request, w,pixmap, getpid(), 0,0);
#endif EBBA
   return;
}
```

X Windows Interface Library Instrumentation

Fig. 6. Library routine instrumentation ([24] refers to Osterhaug [1987]).

represents accurately the behavior. A primitive-event instance is generated when the program executes the instrumentation code fragments. This is generally chosen at a point when all of the major program structures involving the behavior have made the transitions that implement the behavior. Alternatively, a behavior has occurred when the relationships of all of its attributes have assumed values within some established range.

For example, Figure 6 shows the instrumentation of a Sequent Parallel Processing Library [Osterhaug 1987] routine and an X Windows[2] interface library routine. Each of these annotations generate primitive events that accurately characterize the behavior of the function. However, what each of these primitive events characterize is quite different. The parallel programming library event represents transitions in the state of a single variable. The e_LockAcquired event is created after the calling routine executes successfully the code that permits the process to acquire the lock.

[2] X Windows is a trademark of MIT

For the X Interface Library example, the e_XPixmapPut primitive event represents a summary of considerable structure-editing activity. At this level of detail it is only cumulative effects that contain enough information to be useful. It is important to note that this event would not be used to debug the X Interface Library. The application writer using this function assumes that the X Interface Library works properly and will be concerned with properly using the function in concert with others. The user would not be interested in the implementation details of the function, only the primitive-event instance reporting that the function was called.

An EBBA user might need to be aware of the precise semantics of the representative primitive events. Buffering effects or other sources of latency in the exchange among cooperating system components are often responsible for "virtual" behaviors. For the X Windows example, request buffering introduces undetermined latency between request and actual server activity. Instrumentation added to the interface library shows these events to have occurred before the server acts on these requests. A more accurate way to characterize server behavior would be to instrument the server to create the same events.

2.6 Tool Set Integration with the Target System

EBBA can be used for remote or distributed debugging. *Remote* debugging is implemented by placing a user and the set of debugging tools at a single node of the distributed system. Each instrumented component is bound to an agent that has tacit knowledge of its local environment and responds to requests for information and control made by the central site. The primary drawbacks to the use of remote debugging are:

—Latency associated with reading and interpreting information and effecting control activities often causes the information to be old and control activity to lack the desired effect.

—Information tends to be low-level and hence large in volume since there is no local filtering or abstraction applied.

Distributed debugging partitions or replicates the functions of the debugging tools at multiple nodes of a network. Distributed debugging using EBBA emphasizes model abstraction and exchange of resulting high-level events, and cooperation by participating nodes. Benefits that accrue from more-distributed tools include:

—lowered communication bandwidth requirements due to exchange of only necessary or important events,

—more accurate control over the system under study when it is based on local abstractions,

—load distribution of the processing required for model recognition, and

—the ability to accommodate complex topologies that include gateways and subnets that are not fully connected.

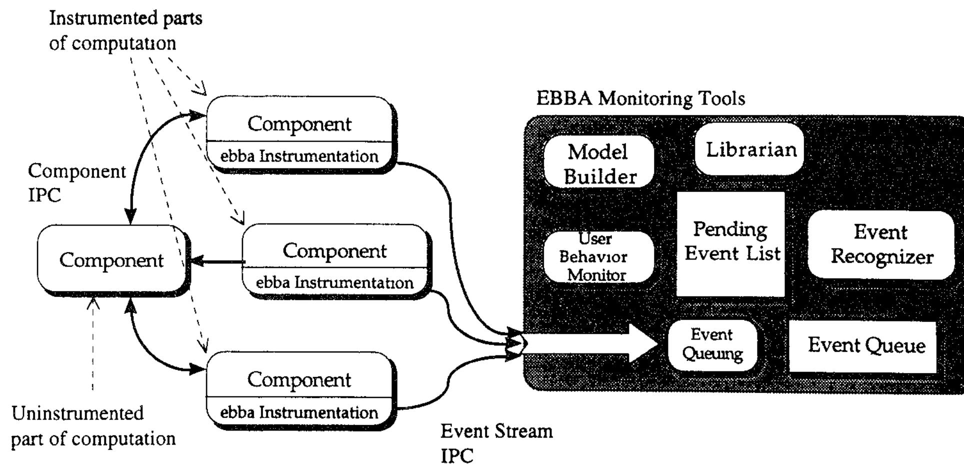


Fig. 7. Relationship of an instrumented system to the EBBA monitoring tools.

Distributed EBBA debugging nodes are capable of much autonomous activity and, once set in motion, may coordinate their activity using a powerful communication abstraction [Bates 1989a]. Figure 7 shows the relationship of an instrumented system to the EBBA monitoring tools.

3. RECOGNIZING BEHAVIORS IN COMPLEX SYSTEMS

The previous section described a set of tools used to build models of behavior and to monitor a system for occurrences of the behavior described by those models. We now describe the techniques used to fit the actual behavior of the system to the user's models.

When debugging a system, the tool user's goal is to understand the difference between the desired system behavior and the models that fit actual system behavior. A user with a good idea about what the system is actually doing will create models that attempt to verify this. A perfect match between a behavior model and system activity demonstrates that a user understands some aspect of a system. If the user is less certain, then the models will contain inaccurate behavior descriptions. Models that fail to match contain some explanation of the mismatch between model and activity. These descriptions will be refined and changed as the user more fully understands the system, i.e., as the user narrows the differences between their models and the actual behavior.

To help users focus their attention on inappropriate program behavior, a guiding principle of behavior model recognition is to provide the user with as much information as possible about how well their models match actual behavior. This is accomplished using two techniques in the behavior pattern recognizer. First is *decomposition* of the model into finer-grained parts, such as subexpressions or other valid partitions of a complete model. This permits the recognizer to match much system activity independently and synthesize

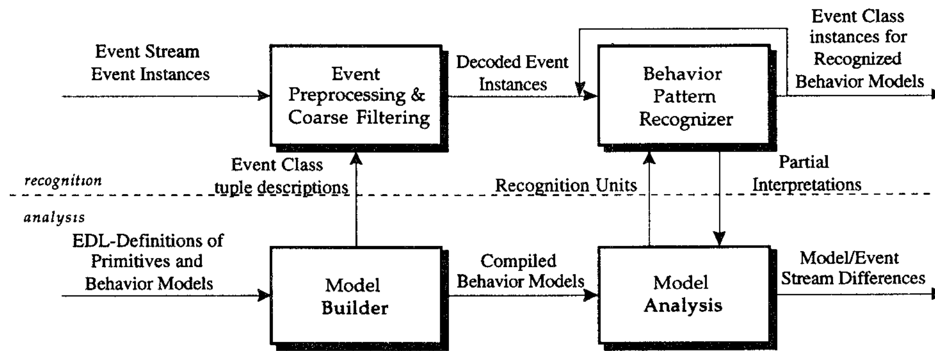


Fig. 8. Pattern Recognition for behavior monitoring.

these results into more-complete matches for a model. Any missing parts represent potential differences to be more closely investigated.

Second is the idea of *partial interpretations*. A partial interpretation of a behavior model occurs when only a portion of the model is matched by actual behavior. The behavior recognition components of the tool set develop multiple partial interpretations for a behavior model and attempt to reconcile these if behavior completely matches the model.

Figure 8 outlines the flow of information through the EBBA behavior recognizer. There are two major functions: *analysis* and *recognition*. Analysis functions include maintenance of primitive-event classes and behavior models and isolation of the differences between behavior models and actual behavior. Analysis is served in the tool set by the Model Builder, Librarian (and its associated event library), User Interfaces, and parts of the Event Recognizer. The recognition functions support analysis by matching behavior models to event instance tuples from the event stream. The recognition components inform the analysis components when they have matched new event instances to a model or created a new partial interpretation. Event Queuing performs the event-preprocessing and coarse-filtering functions by converting arriving event instances to a canonical form and discarding instances not in the viewpoint in use. The Event Recognizer performs the behavior pattern recognition functions.

The token-like event stream and the expression-based EDL model descriptions suggest a *syntactic pattern recognition* approach [Fu 1982] to support behavior recognition. The indefinite-length input stream, the way that the pattern primitives and pattern grammars are created, and the use of partially recognized behavior patterns distinguish pattern recognition techniques required to match behavior models from more conventional, syntactic pattern recognition systems.

3.1 Behavior Model Recognition

The Event Recognizer matches actual system behavior from event stream instances to user-defined behavior models and returns the status of those

comparisons. Behavior recognition is guided by a finite-state transition system that accounts for concurrency, model decomposition, and constraints on attributes of behavior model constituent events. The properties of the formal model are discussed in Bates [1987b].

A request for recognition of a behavior model consists of three elements: the recognition request and parameters, a list of actions to execute when instances of the model are recognized, and a request to start looking for the model. A recognition request accepted by the Event Recognizer passes through three phases. During the *creation* phase, the Event Recognizer obtains the descriptions of each constituent event class for the model. If any are event classes that represent other behavior models, their descriptions are obtained, and so on, until the complete hierarchical structure of the model is known. Next, the analysis part decomposes these models into manageable recognition units and requests that the pattern recognizer create a recognition context for each unit as a Pending Event List entry (Section 2.4.3).

During the *accumulation* phase, pending list entries are filled with event instances generated in the system under observation. As event instances arrive, the event queuer decodes and places them onto the queue. The Recognizer selects an entry from the Pending Event List and attempts to fit the new event instances into it. If an instance can be used to match part of the entry, a new partial interpretation is created that incorporates the instance.

The *instantiation* phase occurs when a pending-event descriptor accumulates sufficient events that match the model it represents. The Recognizer emits an instance of the high-level event class that represents the model and invokes the actions that came with the original recognition request.

These three phases are described in more detail below, followed by an example of behavior model recognition.

3.1.1 *Creation Phase.* During the creation phase the Event Recognition must obtain descriptions of all the components in the hierarchical structure of a model to identify units that may be recognized independently and to ensure that all constituents of the model are defined. The recognition procedure produced by the Model Builder describes a behavior model only in terms of its immediate constituents. This allows the definition of models to change and be made as late as possible. Each unit is a high-level or primitive-event constituent of the event expression, or an anonymous event derived from a subexpression. For example, the following model, AccessConflict, attempts to detect a conflict between local and remote file accesses by determining if their constituent behaviors are interleaved (only the event expressions are shown for clarity):

```
model AccessConflict =
  m_TFTPxfer Δ m_ProcessFile
  ...
end_model
```

AccessConflict is expressed in terms of events, m_TFTPxfer and m_ProcessFile that represent two models (TFTPxfer and ProcessFile, respec-

tively). To recognize AccessConflict both of these will need to be searched for the emit their representative event instances. TFTPxfer describes the general behavior of the Trivial File Transfer Protocol (TFTP) [Sollins 1992]. This protocol is implemented by separate client and server programs to move files from one node on a network to another.

```
model TFTPxfer =
  (e_readFileReq | e_writeFileReq)•
      (e_blockSent•e_blockReceived)+• e_fileTransferred
  ...
end_model
```

The client requests that a file is either accepted from (e_writeFileReq) or returned to it (e_readFileReq). This is followed by a series of data block sends and receives (e_blockSent• e_blockReceived). When the file is completely transferred, this is signaled by the e_fileTransferred event. The subexpression (e_blockSent• e_blockReceived) in the TFTPxfer model is treated as a single constituent of the event expression by the recognizer. This causes the model to be recast as two models. One is the original model with the subexpression replaced with a derived event class

```
model TFTPxfer =
  (e_readFileReq | e_writeFileReq)•S1+•e_fileTransferred
  ...
end_model
```

And the other represents the subexpression

```
model S1 =
  e_blockSent•e_blockReceived
  ...
end_model
```

This subexpression is chosen (and not the other, e_readFileReq...) because it simplifies the recognition procedure. In general, expressions joined by concurrency operators and iterative expressions involving more than one event are recast as subexpressions.

Figure 9 illustrates the AccessConflict behavior as seen by the Event Recognizer. Each dashed node represents a behavior that needs to be recognized. Primitive events do not require pending-event list entries since they are already present in the event stream. Each event representing a behavior model (e.g., m_TFTPxfer) has a pending list entry. As models are recognized, their representative event instances are placed into the event stream and incorporated into higher-level models. Subexpressions (e.g., S_1) within the model are treated implicitly as anonymous high-level events. The AccessConflict model results in four recognition units, one each for the root model (AccessConflict), the high-level constituent events (representing TFTPxfer and ProcessFile), and the subexpression (represented by S_1).

The Pending Event List is organized in two ways. One reflects the hierarchical structure of a model so as to discern easily the role of each unit in an overall context. This is important for determining the differences between behavior models and actual system activity. Recognition units with no corre-

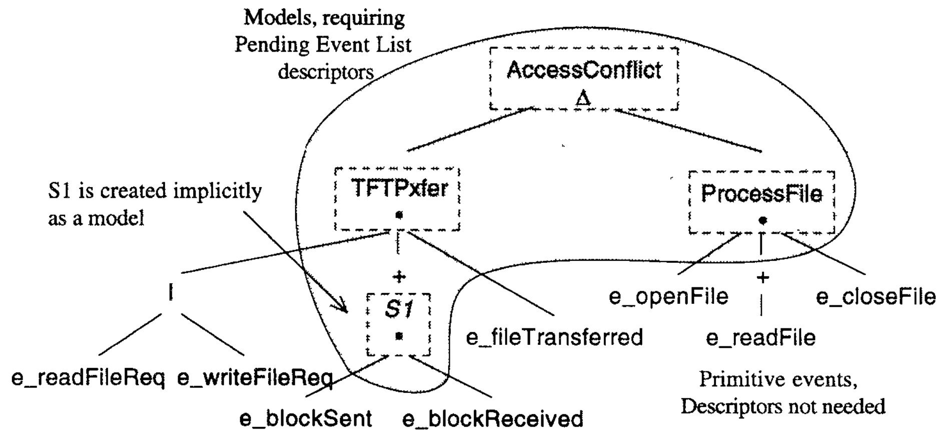


Fig. 9. Hierarchical structure of a high-level model.

sponding event instances indicate potentially erroneous behavior. This structure is also used to determine assignment of a single-event instance that matches more than one recognition unit. The other organization is a scheduling queue that determines the order in which the entries are matched to newly arrived event instances. This prioritizes the fitting of event instances to models and is a simple optimization that permits behavior model constituents that have been recognized to emit their instances into the stream before the entry for the higher-level model is examined.

3.1.2 *Accumulation Phase.* During the accumulation phase, the Event Recognizer matches event instances to the pending list entries. An *event set* is a list of event instances that have been matched to a pending event. The event set has a slot for each event that is needed to satisfy the model. As event instances that match the pending list entry are found they are placed into their slot. If two event instances are eligible to occupy the same event set slot, the recognizer will create a new partial interpretation of the model, differing by the new event instance. In this way, a model may have many partial interpretations, each with a different event set. The multiple interpretations for a model will be resolved when one completes. A model is matched when an interpretation has acquired an event set that advances it to a final state.

There are three important parts of a recognizer description for a model: the *interest set*, the *transition sets*, and the *transition descriptors*. The interest set is the list of event classes that are constituents of a behavior model. If the class of an event instance is found in the interest set, the Event Recognizer will attempt to match the event instance to one or more interpretations of the model. Each *transition set* describes a set of events that may occur concurrently (specified by the concurrency operator). The set description includes the class of each event and references to any constraints expressed in constraints clauses of the behavior model. A transition set is associated with a

```
1   loop
2       PELentry = Next_Pending_List_Entry();
3       loop
4           event = Next_Event (PELentry.current_position) ;
5           if (event = NULL) exitloop;
6           if (event ∉ PELentry.interest_set | Locked(event)) continue;
7           loop
8               interp = Next_Interpretation (PELentry) ;
9               if (interp = NULL) exitloop;
10              loop
                        -- find a slot in the event set and see if the event fits
11                  tset = Next_Transition_Set(PELentry.transition_sets[interp.state]);
12                  if (tset = NULL) exitloop;
13                  if (event ∉ tset) continue;
14                  slot = Event_Set_Index(event, tset);
15                  status = Evaluate_Constraints (PELentry, event, interp, tset, slot);
16                  if (status == failure) continue;
                        -- The event can help this interpretation
                        -- Create a new interpretation if the event set slot is filled
17                  Reserve_Event (event, PELentry);
18                  if (interp.event_set[slot] ≠ NULL)
19                      interp = New_Model_Interpretation (interp);
                        -- Add the event to the interpretation and advance if possible
20                  interp.event_set[slot] = event;
21                  if (Covers (interp.event_set, transtion-set))
22                      New_Model_Interpretation (interp);
23                      Advance_Interpretation (interp, tset);
24              end_loop;
25              if (Final_State (interp.state))
26                  Resolve_Multiple_Interpretations(PELentry, interp);
27                  Instantiate_Event (PELentry);
28          end_loop;
29          Advance_Queue_Position (PELentry)
30      end_loop;
31  end_loop;
```

Fig. 10. Outline of EBBA behavior pattern recognition algorithm.

particular state of the interpretation. When the event set slots corresponding to a transition set are filled, the interpretation advances. The transition descriptors describe a simple state transition system expressed in terms of the transition sets. The outline of the pattern-matching algorithm is found in Figure 10.

This algorithm forms the core of the Behavior Pattern Recognizer (Figure 8). Important features of this algorithm are the event-filtering mechanisms and the criteria for creating multiple interpretations. A brief tour of the algorithm follows.

Once a recognition request has been made, and the system under study is executing, the recognizer will match the incoming event stream to the models. It selects a pending list entry from the schedule queue (line 2), then tries to advance its interpretations using successive event instances from the queue (lines 4–29). If the instance applies to the model (line 6) the recognizer will use it to advance the model. This selecting and filtering mechanism allows interleaving of events for multiple behavior models in the stream,

since events are selected only if they are relevant to the model under consideration.

With an event instance in hand that applies to the model, the next inner loop (lines 8–27) tries to advance each interpretation. If an interpretation reaches a final state, all of the interpretations of the model are resolved, and an event instance is created to represent the model (lines 25–27).

The innermost loop (lines 11–23) performs the actual matching of instances to interpretations. If the class of the event instance is contained in one of the transition sets of the model for the current state of the interpretation (lines 11–14), all constraints (constraints clause of the model definition) that apply to the instance are evaluated (line 15). These constraining expressions filter out event instances based on their attribute values. Constraint evaluation can cause an event instance to be rejected because it is simply unacceptable, or because the instance does not fit with other events already bound to the model.

Constraints are either simple or multidimensional. Simple constraints rely only on the relation of an attribute of a constituent event to a constant value or to another attribute of the same event instance. The acceptability of an event instance as a model constituent can be determined immediately if all constraints applied to its attributes are simple constraints. For example, the constraint in the ProcessFile model (Figure 2)

```
e_openFile.name == name
```

requires that, for the event to be acceptable as a match to the model, the name of a file being opened matches the value of the model parameter, name. Multidimensional constraints involve relations among attributes of two different constituent events, such as:

```
e_openFile.fd == e_readFile.fd
```

The constraints themselves are no more difficult to evaluate than simple constraints, but the interevent dependence requires that all constraint-related events have an event instance bound to the model before a definitive evaluation can take place. To handle this, the constraint evaluator returns a *success*, *failure*, or *don't-know* decision on each evaluation. The constraint evaluator returns the *don't know* result when only some event classes involved in the expression have instances bound to them. All constraint evaluations will return *success* when a model is recognized.

Searching for multiple behavior patterns at the same time creates the possibility that common constituent event instances will match distinct behavior models. If an event instance is used to satisfy more than one model it is said to be *shared* between the high-level events. This is not always acceptable if, for example, the event class appears multiple times in the model.

We control sharing by associating a *lock* list with each event instance that holds references to pending-event list entries. A model may not include an event instance in its event set if a reference to another model in its hierarchy is in the lock list of the event instance. When a model attempts to add an

event instance to its event set it interrogates the lock list (line 6) for a reference to another model in its hierarchy. When a model does add an event instance in its event set, it locks the instance (line 17). Disabling the use of lock lists allows all event instances to be shared. This is useful when an investigation has begun, and the user is trying to survey what the system is doing. When models place the name of the virtual "root" into the list, no instances are shared. This is useful when the user needs to observe the interaction of various models closely.

3.1.3 *Instantiation.* Instantiation occurs after an interpretation has accumulated sufficient event instances to satisfy its model. Now the recognizer must resolve any multiple interpretations for the model and create an event instance that can be inserted into the event stream. Also, the actions supplied with the original recognition request are executed.

Resolving multiple interpretations is accomplished by deleting all interpretations that were created from the one that is to be instantiated. The interpretation that created the instantiated one continues to be searched for. To create the event instance, each attribute-binding expression defined in the model description (attributes clause) is evaluated to supply an attribute value. The completed instance is then delivered to the event queuer and added to the event stream. Finally, the list of actions is executed. The user employs this mechanism to control the system under study and effect other tool set activities.

3.1.4 *A Short Example of Behavior Model Recognition.* Now we can apply our behavior recognition algorithm to match the TFTPxfer model (Figure 9) against a sample event stream. Typically, the user issues a request to recognize the TFTPxfer model through the model interface tool. The request is similar to:

(recognize TFTPxfer "xf − 1")	—*request recognition of the model*
(addRCB'(list_completed "xf − 1"))	—*display the event set of any matched instances*
(start_pe "xf − 1")	—*allow the model to accumulate events*

The string "xf − 1" is a tag used to identify the recognition request. It will be supplied on all subsequent queries. The recognizer requests the definition of TFTPxfer from the Librarian and creates pending list entries for TFTPxfer and the subexpression S_1 = (e_blockSent•e_blockReceived). The pending list will thus have the following entries, each with a single empty interpretation (a solid box represents a pending list entry; dashed boxes are interpretations of the model it represents):

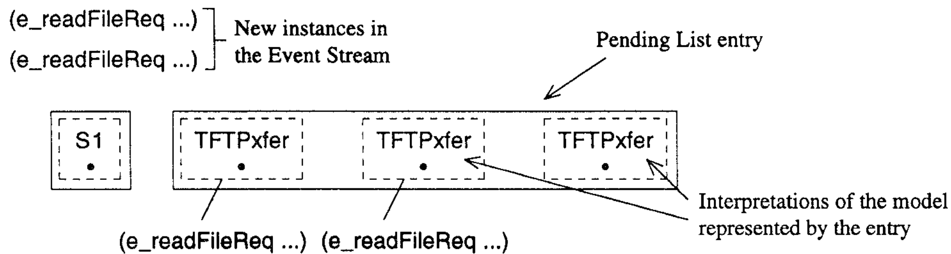
The recognizer then waits for the queuer to inform it of arriving event instances. When the instrumented TFTP client requests a file transfer (in

```
(e_readFileReq "/etc/rc.local" "netascii" 735838645:11718 "sluggo:tftp")
(e_readFileReq "/etc/rc.local" "netascii" 735838645:15624 "labdec:tftpd")
(e_blockSent 1 512 735838645:19530 "labdec:tftpd")
(e_blockReceived 256 512 735838645:23436 "sluggo:tftp")
(e_blockSent 2 512 735838645:23436 "labdec:tftpd")
(e_blockReceived 512 512 735838645:31248 "sluggo:tftp")
(e_blockSent 3 512 735838645:31248 "labdec:tftpd")
(e_blockReceived 768 512 735838645:39060 "sluggo:tftp")
(e_blockSent 4 512 735838645:42966 "labdec:tftpd")
(e_blockReceived 1024 512 735838645:46872 "sluggo:tftp")
(e_blockSent 5 512 735838645:50778 "labdec:tftpd")
(e_blockReceived 1280 512 735838645:54684 "sluggo:tftp")
(e_blockSent 6 512 735838645:54684 "labdec:tftpd")
(e_blockReceived 1536 512 735838645:62496 "sluggo:tftp")
(e_blockSent 7 294 735838645:66402 "labdec:tftpd")
(e_blockReceived 1792 294 735838645:70308 "sluggo:tftp")
(e_fileTransferred "to-client" 8 735838645:70308 "labdec:tftpd")
(e_fileTransferred "/etc/rc.local" 7 735838645:74214 "sluggo:tftp")
```

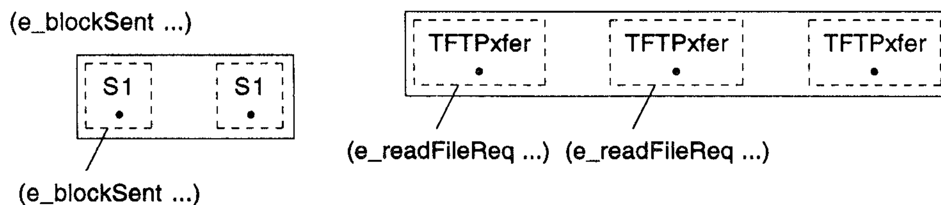
Fig. 11. Event stream resulting from TFTP client file transfer request.

this case, /etc/rc.local) from the TFTP server program the event stream of Figure 11 would result. The client request is noted by the first event instance (e_readFileReq ... "sluggo:tftp"). Receipt and honoring of the file transfer request by the server is noted by the (e_readFileReq ... "labdec:tftpd") event.

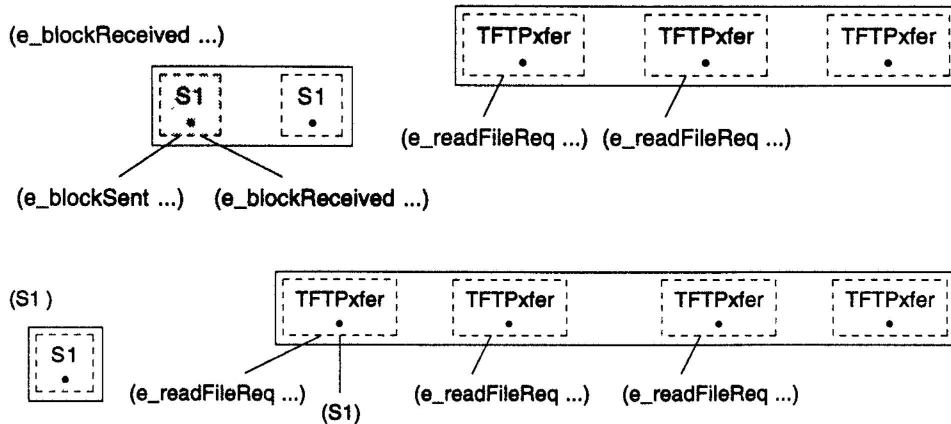
Upon receiving the (e_readFileReq ...) event instance from the TFTP client, the queuer informs the recognizer that new event instances have arrived. The recognizer selects the pending list entry for TFTPxfer and determines that this event instance matches the model. It creates another interpretation and advances the original. Arrival of the (e_readFileReq ...) instance from the server repeats this process. The state of the pending list is then:



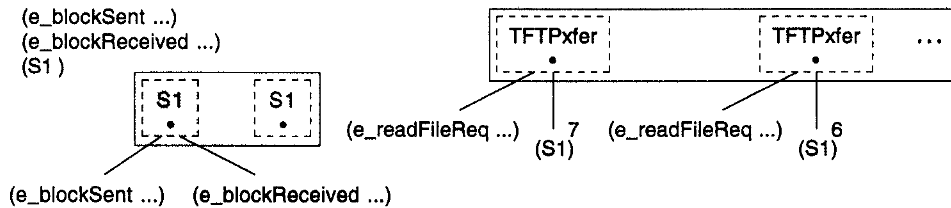
When the e_blockSent, e_blockReceived pairs arrive, the recognizer selects the entry for the S1 subexpression and matches these events to it.



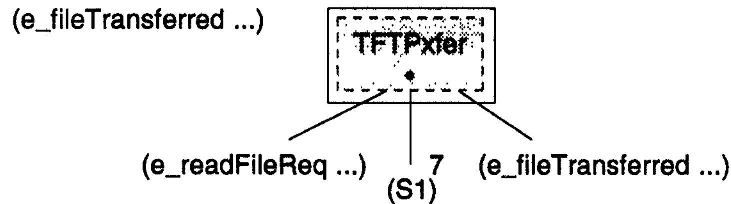
The model completes (indicated by the shaded entry), and the S1 events are added to the event set for TFTPxfer as depicted below.



Each time a new S1 is added, another interpretation is created. The effect is that a number of interpretations are created, each with a different number of S1 instances in its event set (superscripts in the figure).



When the (e_fileTransferred ...) event instance arrives, the interpretation with the longest string of S1 instances will be matched first and the others deleted (since each was derived from it).



The resulting event instance is

(m_TFTPxfer "/etc / rc.local" 735838646:2613 "sluggo:EventMonitor")

The name of the file transferred ("/etc / rc.local") is an attribute of the event as is the time it was recognized (7358 ...) and the location of the Recognizer ("sluggo:EventMonitor").

The action ((list_completed "xf − 1")) executed upon recognition of the model simply displays the event set bound to the recognized model:

```
xf − 1, HighLevel
m_TFTPxfer "/etc/rc.local" 735838646:2613 "sluggo:EventMonitor"
Constituents
  0)  e_readFileReq "/etc/rc.local" "netascii" 735838645:11718 "sluggo:tftp"
  1)  -- undef --
  2)* e_blockSent 7 294 735838645:66402 "labdec:tftpd"
  3)* e_blockReceived 1792 294 735838645:70308 "sluggo:tftp"
  4)  e_fileTransferred "/etc/rc.local" 7 735838645:74214 "sluggo:tftp"
```

The displayed list is marked undefined for slot 1. This is the event set slot for (e_writeFileReq...) which is unmatched because the alternative constituent, (e_readFileReq...), was matched to the model. The entries marked with * have multiple events bound to the slots with only the most recent displayed.

5. SUMMARY AND STATUS

Debugging using EBBA emphasizes model building as the prime investigation tool and intensive use of computational resources to verify and help refine behavior models. Modeling involves describing patterns of expected behaviors and using pattern recognition techniques to determine how well the actual behavior fits the user-defined patterns.

EBBA supports a sophisticated tool set as a distributed program. Using appropriate choices for remote information processing and communicating high-level event communication, an EBBA-based distributed debugging tool set provides easily and naturally a range of solutions to monitoring and debugging 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 abstractions of behavior.

Creating behavior models and trying them out on the executing system is easy. Different behavioral hypotheses are easily modeled and compared to the system. System partitioning into viewpoints and the ability to merge different viewpoints easily has been useful in unanticipated ways. Often a viewpoint will be merged and events from that view incorporated into a model simply to provide a contextual framework for the model. This has allowed models developed from one viewpoint to reinforce and augment models in other viewpoints and is convenient for explaining whether hypotheses about incorrect behavior are valid or not.

Partial interpretations of behavior models have proven to be very useful. Incompletely recognized behaviors indicate that the modeler should more closely examine the class of behaviors that are missing, or explain outright what is wrong with a particular program execution. When missing constituent behaviors do not obviously explain erroneous behavior, they tend to result in perspective shifts to alternative viewpoints and modeling at less abstract levels. This results in true, goal-driven, top-down behavior modeling.

The Annotation Tool has greatly simplified instrumenting systems. We have had some limited success with automatic probe insertion at compile time and an experiment with very flexible binding that involve modifying executable program text. These techniques are important but do not have the preciseness and portability of the simpler source code fragment insertion instrumentation method. Probe effects have occasionally obscured race conditions, but removing annotations and using alternative viewpoints have always solved this problem.

Much of the difficulty with probe effects, probe placement policy, questions about accurate characterization, etc. would be alleviated if it was easy to insert and remove instrumentation code fragments as needed. Adding software probes should be treated much like an oscilloscope probe used by hardware engineers. To effect this, two changes might be made in the way program-building software (e.g., compilers) is developed. First, language compilers must become more open, that is, they should be written to allow diverse kinds of instrumentation to be added to the code they generate. Currently, the compilers most widely used include symbol table information to be used with some kind of a symbolic debugging tool. Aids for debugging tools should be expanded to include ways to insert annotations of an arbitrary but circumscribed nature, such as for EBBA or IPS-2 [Miller et al. 1988]. This would encourage a variety of techniques, each suited to particular types of programming problems.

The second change needed is that heavily used reusable software components should be designed with probes in place that can be activated when needed (once suggested by Knuth [1972]). Language compilers and runtime environments can facilitate this process so that it may be greatly aided by the compiler changes recommended above. The envisioned embedded instrumentation should be integrated with its surroundings and hence be more sophisticated than simply surrounding instrumentation code with *if-then-else* brackets.

A current effort involves instrumenting the software that controls signaling nodes of the Common Channel Signaling system (CCS/SS7) network [Lai and Bates 1991]. This software is responsible for setting up and allocating resources that enable communication through the public-switched network operated by the Regional Bell Operating Companies and other providers. These systems are typically clusters of multiprocessors, interconnected with redundant point-to-point communication links. The goals of the work are to provide uniform, supplier-neutral monitoring and a capability to diagnose errors in the control software. The ability of EBBA to model behavior based on events that occur at geographically dispersed nodes is important to this work since many failures result from systems making local decisions that impact nonlocal resources.

EBBA began as an effort to provide high-level debugging [Bates et al. 1983]. It has evolved to provide sophisticated system monitoring for a variety of purposes based on behavior modeling. Future work on the Behavioral Abstraction paradigm includes developing techniques for automated model analysis and an ability to create models encompassing more aspects of system

behavior. Model analysis is approached largely as finding differences in behaviors using error-correcting parsing [Aho and Peterson 1972] and error-correcting tree automata [Lu and Fu 1978] techniques. Providing for these kind of analyses should help to exploit the information bound into partially recognized models.

REFERENCES

AHO, A. V. AND PETERSON, M. L. 1972. A minimum distance error-correcting parser for contex free language. *SIAM J. Comput. 1*, 4 (Dec.), 305–312.

BATES, P. C. 1989a. Event monitoring and abstraction tools Tech. Rep. 89–17, Univ. of Massachusetts, Amherst, Mass.

BATES, P. C. 1989b. Tracking the elusive Mandelbrot set error using event based behavior abstraction. Tech. Rep. 89–06, Univ. of Massachusetts, Amherst, Mass.

BATES, P. C. 1987a. The EBBA modelling tool, a.k.a. event definition language. Tech. Rep. 87–35, Univ. of Massachusetts, Amherst, Mass.

BATES, P. C. 1987b. Shuffle automata: A formal model for behavior recognition in distributed systems. Tech. Rep. 87–27, Univ. of Massachusetts, Amherst, Mass.

BATES, P. C. AND WILEDEN, J. C. 1982. Event Definition Language: An Aid to Monitoring and Debugging of Complex Software Systems. In *Proceedings of the 15th Hawaii International Conference on System Sciences*. 148–156.

BATES, P. C. WILEDEN, J. C., AND LESSER, V. R. 1983. A debugging tool for distributed systems. In *Proceedings of the 2nd Annual Phoenix Conference on Computers and Communications* (Mar.). 311–315.

BRUEGGE, B. 1983. User Manual for KRAUT—The Interim Spice Debugger. Spice Doc. S156, Carnegie-Mellon Univ., Pittsburgh, Pa.

CURTIS, R. AND WITTIE, L. 1982. Bugnet: A debugging system for parallel programming, environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 394–399.

DAHL, O. J., DIJKSTRA, E. W., AND HOARE, C. A. R. 1972. *Structured Programming*. Academic Press, New York.

ENSLOW, P. H. 1978. What is a "Distributed" data processing system. *IEEE Comput. 1*, 1 (Jan.), 13–21

FU, K. S. 1982. *Syntactic Pattern Recognition and Applications*. Prentice-Hall, Englewood Cliffs, N.J.

GARCIA-MOLINA, H., GERMANO, F., AND KOHLER, W. H. 1984. Debugging a distributed computing system. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar.), 210–219.

HOUGH, A. A. AND CUNY, J. E. 1987. Belvedere: Prototype of a pattern-oriented debugger for, highly parallel computation. In *Proceedings of International Conference on Parallel Processing*. 735–738.

KNUTH, D. E. 1972. *The Art of Computer Programming*. Vol. 1, *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass.

LAI, M.-Y. AND BATES, P. C. 1991. Instrumentation and monitoring in CCS networks for software analysis and debugging. Bellcore Tech. Memorandum, TM-NWT-020497, Dec.

LAMPORT, L. 1978. Time, clocks and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPORT, L. AND MELLIAR-SMITH, P. M. 1984. Byzantine clock synchronization. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 68–74.

LU, S. Y. AND FU, K. S. 1978. Error-correcting tree automata for syntactic pattern recognition. *IEEE Trans. Comput. C-27* (Nov.).

MCDANIEL, G. 1977. Metric: A Kernel instrumentation system for distributed environments. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*. ACM, New York, 93–99.

MILLER, B. P., CLARK, M., KIERSTEAD, S., LIM, SEK-SEE, AND TORZEWSKI, T. 1988. Ips-2: The second generation of a parallel program measurement system. Tech. Rep. 783, Univ. of Wisconsin-Madison, Aug.

MILLER, B. P., MACRANDER, C., AND SECHREST, S. 1986. A distributed programs monitor for berkeley UNIX. *Softw. Pract. Exp. 162*, 2 (Feb.), 183–200.

MODEL, M. L. 1979. Monitoring system behavior in the complex computational environment. Tech. Rep. CSL-79-1, XEROX Palo Alto Research Center, Palo Alto, Calif.

OSTERHAUG, A. 1987. *Guide to Parallel Programming*, 2nd ed. Sequent Computer Systems.

RIDDLE, W. C. 1976. An approach to software system modelling, behavior, specification, and analysis. Tech. Rep. RSM/25, Univ. of Michigan, Dept. of Computer and Communications Science, Ann Arbor, Mich.

SCHIFFENBAUER, R. D. 1981. Interactive debugging in a distributed computational environment. Tech. Rep. MIT/LCS/TR-264, MIT, Cambridge, Mass.

SHAW, A. C. 1980. Software specification languages based on regular expressions. In *Software Development Tools*, W. E. Riddle and R. E. Fairley, Eds. Springer-Verlag, New York, 148–175.

SMITH, E. T. 1981. Debugging techniques for communicating loosely-coupled processes. Tech. Rep. TR 100, Univ. of Rochester, N.Y.

SNODGRASS, R. 1984. Monitoring in a software development environment: A relational approach. In *Proceedings Software Engineering Symposium on Practical Software Development Environments.*

SNODGRASS, R. 1982. Monitoring distributed systems: A relational approach. Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

SOLLINS, K. 1992. The TFTP protocol (Revision II). RFC 1350, Internet Engineering Task Force, Network Information Center (NIC), SRI International, Menlo Park, Calif.

WEISER, M. 1982. Programmers use slices when debugging. *Commun. ACM 25*, 7 (July), 446–452.

WEISER, M. 1979. Theoretical foundations of program slices. Tech. Rep. RSSM/69, Univ. of Michigan, Ann Arbor, Mich.

WILEDEN, J. C. 1978. Modelling parallel systems with dynamic structure. Tech. Rep. 78-4, COINS Dept., Univ. of Massachusetts, Amherst, Mass.

Received May 1988; revised June 1993; accepted June 1994