Models for Monitoring and Debugging Tools for Parallel and Distributed Software*

DAN C. MARINESCU
Computer Sciences Department, Purdue University, West Lafayette, Indiana 47907

JAMES E. LUMPP, JR.,† AND THOMAS L. CASAVANT†
Parallel Processing Laboratory, Department of Electrical and Computer Engineering, University of Iowa, Iowa City, Iowa 52242

AND

HOWARD JAY SIEGEL
Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907

Several aspects of the multidimensional problem of providing monitoring support tools for the debugging and performance analysis of software for distributed and parallel systems are presented. A formal event-action model at the process level and a layered architectural model are introduced. The application of the event-action model to the development of the layered architectural model is shown. This effort was motivated by the need to understand the ways in which a monitoring system may intrude upon a monitored system. An understanding of the fundamental ideas underlying the relationship between monitoring and monitored systems is necessary to build practical tools for software development. These models are currently being used in the development of monitoring tools for the PASM parallel processing system prototype. © 1990 Academic Press, Inc.

1. INTRODUCTION

The debugging and analysis of multiprocessor software is extremely difficult. In order to produce efficient solutions the programmer must gain insight into the interaction between the system and the executing program. This information can only be provided by monitoring the executing software through some form of instrumentation. Monitoring of parallel programs can be done with different degrees of intrusiveness. At one end of the spectrum there are software-only probes. Such systems are referred to here as intrusive monitors because they have the greatest effect on the execution of the program being observed. At the other end of the spectrum, there are systems with extensive hardware support for the monitoring functions. Such systems that do not affect, in a "noticeable way," the behavior of the programs being monitored are called nonintrusive monitors. Monitoring systems with limited hardware support can provide different degrees of intrusion and a continuum of such systems with properties between these extremes can be envisioned and built.

The degree of intrusion that can be tolerated depends upon the nature of the application and upon the desired results of monitoring. To determine this level, it is important to understand that an intrusive monitor perturbs the execution of the parallel program by altering in an arbitrary manner the timing of events in the multiple threads of control being monitored. Multiple threads of control are defined as the case in which multiple processors are executing (potentially) independent instructions. Altering the timing of events may:

(a) lead to incorrect results,
(b) create (or mask) deadlock situations when the order of events in different threads of control is affected,
(c) cause a real-time program to fail to meet its deadlines,
(d) increase drastically the execution time of the program being monitored,
(e) make the debugging of a parallel program a difficult task.

As an example of (e), consider two events $E_1$ and $E_2$ occurring in different processing elements (PEs) such that the order of the two events is important for debugging. If, depending upon the input data, an intrusive monitor some-
times alters the order of the events and sometimes does not, debugging of the program becomes impossible.

Whenever an intrusive monitor leads to any combination of the conditions (a) to (e) listed above that cannot be tolerated for the monitoring goals, then it is necessary to decrease the level of intrusion, typically by augmenting the hardware support for instrumentation.

Instrumentation of conventional serial computing systems for performance monitoring and debugging is a fairly well understood subject [37], however, instrumentation for multiprocessor systems is far more difficult. Serial programs can be monitored and debugged using solely software probes to define breakpoints and to analyze the program status at these breakpoints [3]. It is virtually impossible to extend these techniques to the parallel case because in parallel programs there are multiple independent threads of control. A software probe in one thread of control can reliably supply only local information from that thread. Moreover, a software probe in one thread of control cannot affect other threads of control, e.g., stop them in order to examine system state. Without any hardware support a monitoring system generally cannot handle nonlocal information (i.e., information about events defined across multiple threads of control).

Two examples are now presented to illustrate the limitations of intrusive monitoring. The first example shows that intrusive monitoring provides only aggregate measures of performance; it does not allow refined measurements. For example, consider two processes running on two PEs, PEi and PEj of a hypercube multiprocessor. Assume that the process running on PEi needs partial results produced by the process running on PEj and at some moment of time issues a read command and then waits for a message. An intrusive monitor will report the interval between READ-BEGIN and READ-END pairs as communication time, while a refined measurement will break this interval into synchronization time $\tau_1$ (waiting time), connection time $\tau_2$ (propagation time and communication protocol delay), and data transmission time $\tau_3$, as shown in Fig. 1. The refined measurements provide valuable information to tune the application. By modifying the algorithm to transmit the message earlier, the synchronization time may be reduced or even eliminated. If the connection time is large compared with the data transmission time, the program may benefit from transmitting large messages.

The intrusive monitor consists of code embedded into each process that creates a trace record with a timestamp every time one of several local events occur, e.g., whenever a read or a write is initiated or terminated. Because the local clocks of the two PEs cannot be synchronized perfectly, even a cross-reference of local events can only give approximate results. In contrast, a nonintrusive monitor supported in hardware allows ordering of such nonlocal events. This is done by distributing a global clock signal to monitoring hardware at each node. The two local hardware monitor nodes of PEi and PEj observe the local events, and then insert timestamps and send messages through an independent interconnection network to inform a central monitoring node about their occurrence. In this case the results are available on-line as opposed to the intrusive monitoring case when only a cross-reference performed off-line is possible.

Without any hardware support a monitoring system is not capable of handling nonlocal information. For example, it is virtually impossible to define in PEi the nonlocal event WRITE-BEGIN in PEj in the previous example. Also it could not perform actions affecting more than one thread of control.

As a second example, consider a pair of asynchronous processes as illustrated by the two parallel threads P1 and P2 in Fig. 2. In this example, there are two resources to which mutually exclusive access may be necessary. These are represented by semaphore variables S1 and S2. Note that a possible deadlock-free execution sequence of the P and V operations shown is 1, 2, 3, 4, 6, 7, 8. However, if the P on line 5 is reached prior to the P on line 2, deadlock will occur. Assume that in a system without any monitoring facilities that the lengths of code blocks CB1, CB2, and CB3 obey the relation CB2 < CB1 + CB3. In this case, deadlock will occur after the partial execution sequence 1, 5. However, if intrusive monitoring lengthens CB2 such that CB2 > CB1 + CB3, then deadlock will not occur. Clearly, if the goal of monitoring is to detect and locate timing-sensitive problems such as this example of deadlock, any amount of intrusion will be unacceptable.

As early as 1975, McDaniel proposed a kernel instrumentation for distributed environments [25]. The work on some of the distributed systems designed in the early 1980s includes some performance monitoring facility, usually a software monitor, e.g., the work of Cheriton [9] for the V kernel and Powell and Miller [29] for DEMOS/MP. Systems for interactive debugging of a distributed computational environment were considered as a separate issue [32]. Several monitors for distributed systems based on an Ethernet or a Hyperchannel are being built, e.g., the work
of Watson at Livermore and of Ogle and Schwan on the Real-Time monitoring systems at Ohio State [39]. Miller at Wisconsin is currently developing iPSC2, based upon his experience with the Distributed Program Monitor [27]. Work in progress is reported also at HP Labs (Spin and Ratti) at AT&T (Jordan). Instrumentation for parallel systems is carried out at NBS where TRAMS, a software measurement system, and REMS, a resource monitoring system, are under development. LeBlanc is working on debugging tools based upon Instant Replay [18]. Bates at Massachusetts [1, 2] has developed a sophisticated environment for High Level Debugging, based upon the EBBA paradigm (Event Based Behavioral Abstraction). A language-based approach to debugging is discussed in [38]. Dongarra and Sorensen at Argonne National Lab have developed a package, SCHEDULE, for the analysis of parallel FORTRAN programs using dependency graphs [12]. Another set of useful tools developed at Argonne consists of MAP1 and MAPA that use trace files [6]. SEECUBE is a package developed by Couch that allows the programmer of a parallel computer with a hypercube communication geometry to visualize communication within parallel programs [10]. Instrumentation for iPSC/2 was developed at Illinois [24, 30]. Tools have also been developed that aid the programmer in visualizing the execution using Petri nets [14].

This work differs from previous work in that the goal is to develop a complete model of the monitoring process based on the event–action paradigm and an associated layered architecture model for a monitoring system. These models can then be used to classify existing monitoring systems as well as to design new systems ranging from software-only intrusive monitors to completely nonintrusive hardware monitors. In this paper, a formal model for the event–action paradigm is developed at the process level. The relevant components of monitored and monitoring/debugging systems are described in terms of a set of primitives. These primitives are then used to define the interactions between the monitored and monitoring/debugging system. Concepts such as intrusive and nonintrusive monitoring and observability are examined using this model. In addition, these concepts are mapped to monitoring imple-

mentations by way of a architectural model of the monitoring system.

While the paper addresses fundamental models for parallel and distributed systems monitoring tools for software development, this work is motivated by the need for practical monitoring systems. In our research on developing instrumentation for the 30-processor PASM prototype [20, 21], we found a need for defining the theory underlying the monitoring systems. Thus, one purpose of the theoretical study presented here is to provide a basis upon which practical monitoring systems may be designed.

Section 2 introduces a process level formal model for the event–action paradigm. The way in which timing is considered with respect to the event–action paradigm introduced is discussed in Section 3. Finally, Section 4 moves the concepts of the event–action model to an architectural model and a proposed implementation.

2. A FORMAL MODEL FOR THE EVENT–ACTION PARADIGM

The event–action paradigm provides a general framework for the description of an entire class of concurrent systems in which a well-defined causality relationship exists [8]. In such systems, when any event associated with a well-defined subset of state changes occurs, predefined actions must be enabled. Monitoring and debugging systems [19], kernels of operating systems, as well as different control systems [22], all operate according to this paradigm.

A formal model (at the process level) that describes the event–action paradigm used for monitoring and debugging functions associated with a distributed or parallel system is introduced in this section. Such a formal model is necessary to identify the primitive components of performance monitoring and debugging. Using these elements, it is then possible to establish the interactions among these primitive activities, to define the relationships between the monitoring activities and the rest of the system, and to construct an architectural model for performance monitoring and debugging. The architectural model defines layers that group together primitive activities supporting the event–action paradigm, and makes explicit the interfaces between layers. This architectural model may then be used for implementation of monitoring and debugging tools for parallel and distributed systems software development.

The model introduced in this paper identifies two subsystems at the process level. Informally, these are the target subsystem that consists of the application processes and a monitoring subsystem that provides the monitoring and debugging functions needed for software development support and performance analysis.

Consider a message-oriented system \( \mathcal{P} \) consisting of a finite set of processes, \( \mathcal{P} \), and a finite set of message channels,
A process is informally defined as a program in execution. A message is a data type used to send information from one process to another. The precise format of a message is not defined at this stage except that it is assumed a message has a unique message identifier. A message channel is an abstract structure that identifies the destination of a message. A message channel is associated with a physical communication channel. In this model it is assumed that a communication channel is error-free, delivers messages in order, and has arbitrary, but finite, message delay. The state of a channel is the sequence of messages presently being sent along the channel.

In this model, the set of processes is partitioned into two disjoint subsets: \( \mathcal{A} \), the subset of active processes, and \( \mathcal{R} \), the subset of reactive processes. Thus, \( \mathcal{P} = \mathcal{A} \cup \mathcal{R} \).

An active process \( p \in \mathcal{A} \) is defined by a set of states, one of which is denoted as the initial state, and a sequence of events. The state of an active process is determined by the value of all its variables, including its program instruction counter (pure code is assumed). Following Chandy and Lamport [7], an event \( e \) in an (active) process \( p \) is defined as an atomic entity that reflects a change of state of \( p \) or of a channel adjacent to \( p \). An event \( e \) is characterized by a five-tuple \( (p, s, s', c, M) \) with

- \( p \), the active process in which the event occurs,
- the state \( s \) of \( p \) immediately prior to the occurrence of the event,
- the state \( s' \) of \( p \) immediately after the event,
- the channel \( c \) whose state may be altered by the event,
- the message \( M \) sent along \( c \).

The model introduced in this paper defines a reactive process \( m \in \mathcal{R} \) by:

- the subset \( \mathcal{A}_m \subseteq \mathcal{A} \) of active processes \( m \) is supervising, \( \mathcal{A}_m = \{ p_m, p_m_1, \ldots, p_m_n \} \),
- the set of events \( E_m \) that \( m \) recognizes \( E_m = \{ e_m_1, e_m_2, \ldots, e_m_n \} \),
- the set of actions \( A_m \) that \( m \) is capable of performing \( A_m = \{ a_m, a_m_1, \ldots, a_m_n \} \).

A reactive process performs one or more actions in response to one or more events. A reactive process is initially in a suspended state; it is activated by the occurrence of the events in the active processes it supervises. After identifying an event \( e \), the reactive process \( m \) determines what action \( a \in A_m \) has to be performed, performs action \( a \), and then returns to the suspended state.

In general, only a distinguished subset of events has actions associated with it and is called events of interest. An equivalent way of presenting this issue is to consider that a null action, \( \alpha_0 \), is associated with any event that does not belong to the set of events of interest. Here, however, the former convention will be used.

The set of channels is partitioned into three disjoint subsets \( \mathcal{C} = \mathcal{C}_a \cup \mathcal{C}_r \cup \mathcal{C}_{ar} \). Channels \( c \in \mathcal{C}_a \) connect only active processes, channels \( c \in \mathcal{C}_r \) connect only reactive processes, and channels \( c \in \mathcal{C}_{ar} \) connect active with reactive processes.

With these definitions, two subsystems of the system \( \mathcal{S} \) can be identified: a target subsystem \( \mathcal{T} = \{ \mathcal{A}, \mathcal{C}_a \} \), consisting of active processes and channels, and a monitoring subsystem \( \mathcal{M} = \{ \mathcal{R}, \mathcal{C}_r \} \), consisting of reactive processes and channels. The two subsystems are interconnected by means of channels in \( \mathcal{C}_{ar} \). Hence, \( \mathcal{S} = \{ \mathcal{T}, \mathcal{M}, \mathcal{C}_{ar} \} \).

Two basic types of interrelationships between a monitoring and a target subsystem are possible depending, in part, upon the feedback from the monitoring subsystem.

(a) A monitoring subsystem \( \mathcal{M} \) is intrusive if it induces changes of state of \( \mathcal{T} \), or if it alters the timing or ordering of events in \( \mathcal{T} \).

(b) A monitoring subsystem \( \mathcal{M} \) is nonintrusive if it does not induce changes of state of \( \mathcal{T} \) and if it does not alter the timing or ordering of events in \( \mathcal{T} \).

The intrusion may occur at the time of event detection and/or as the result of an action. Event detection may alter the timing of events, their ordering, both the timing and the ordering of events, or neither of them. The first three cases correspond to intrusive event detection and the last one to nonintrusive event detection. Actions may also be intrusive or nonintrusive. An intrusive action alters the timing, ordering of events, the state of the active process, or all three, while a nonintrusive action does not alter any of these. As examples, a nonintrusive action might be simple event counting, and an intrusive action could change the state of \( \mathcal{A} \) as part of the debugging process.

Consequently, a monitoring subsystem is nonintrusive if event detection and all actions are nonintrusive. The monitoring subsystem is intrusive if either event detection or actions are intrusive.

Intrusion can also be discussed in terms of information flow between \( \mathcal{T} \) and \( \mathcal{M} \). If the flow of information (on \( \mathcal{C}_{ar} \)) between the two subsystems is unidirectional from \( \mathcal{T} \) to \( \mathcal{M} \), both intrusive and nonintrusive event detections are possible, and only nonintrusive actions can occur. In a unidirectional channel information flows only in one direction, however, this does not preclude acknowledgments in the other direction. This implies that the receiver may delay the sender as a form of flow control. If the information flow between \( \mathcal{T} \) and \( \mathcal{M} \) is truly bidirectional, intrusive actions can occur.

Consider an example of a monitoring subsystem with intrusive event detection and unidirectional information flow. When an event occurs in an active process \( p \), the process is suspended due to nonreceipt of an acknowledgment, and it resumes execution only after event detection in the reactive process completes. In this case \( \mathcal{M} \) exercises a form of flow control upon \( \mathcal{T} \). The timing of events in \( p \) is
changed, and possibly the global ordering of events in \( T \) is altered (i.e., the ordering of the suspended processes with respect to other processes in \( T \)).

A typical operating system on a uniprocessor involves intrusive event detection and intrusive actions, but with a more general goal. In this case, the events are hardware and software interrupts, the active processes are user processes, and the reactive processes are interrupt handling and resource management routines of the operating system.

Debugging tools [18] generally require that \( M \) exercise some form of control over \( T \). Hence, they may be associated with intrusive monitoring. A difficulty in using such tools, however, arises from the fact that the intrusion causes changes in timing or ordering of events, and thus these tools have limitations for debugging timing-sensitive and real-time software. Most real-time control systems perform one form or another of nonintrusive monitoring [23].

Performance measuring systems can be intrusive or nonintrusive as well. Whenever it is required that the measuring system not perturb the system being measured, the nonintrusive monitoring solution is mandatory. In this case, nonintrusive event detection is a minimum requirement. Unfortunately, nonintrusive monitoring raises difficult conceptual and practical problems, and a compromise is usually reached to minimize the effects the intrusive monitoring subsystem has upon the target subsystem. Most existing monitoring systems for parallel architectures perform one form or another of intrusive monitoring. For example, the SEECCUBE package [10], the SCHEDULE package [11], the MAPI and MAPA tools [6], and the CAPS execution and debugging environment for PASM [21].

The model described above indicates some of the major difficulties encountered in the nonintrusive monitoring of software behavior. Nonintrusive monitoring requires that the target subsystem, \( T \), and the monitoring subsystem, \( M \), not share any system resources other than the channels in \( E_{ar} \). Furthermore, \( E_{ar} \) must be unidirectional with no acknowledgments from \( M \) to \( T \). This is a very strict requirement and implies that nonintrusive monitoring is simply not possible without hardware support consisting of dedicated processors for reactive processes and dedicated communication channels for communication among reactive processes.

3. TIME AND THE EVENT-ACTION PARADIGM

Section 2 defined the basic elements of a model of monitoring and monitored parallel/distributed systems. In this section, the dynamic behavior of these elements is studied and the model made complete through an examination of the relationship between time and the elements of the model. Some of the terminology used here has appeared elsewhere in a different context [35]. Although the context, and often the details of the definitions, differ, some of the same terms are used because they describe related concepts. The context here is a complete formal model for a monitoring system for parallel and distributed processing.

Time plays a central role in the event-action paradigm. For any event-action pair \((e, a)\), one can identify on an arbitrary time scale available to an observer external to the system the following sequence of steps.

- Event \((e)\) occurs \((o)\) at time \( t_{eo} \). As mentioned earlier, the occurrence of the event is associated with a change of state of active process \( p \). This change of state will also trigger sending a message from active process \( p \) to a reactive process \( m \), if \( e \in E_m \).

- Event \((e)\) recognition \((r)\) at time \( t_{er} \). The event is recognized when the reactive process \( m \) receives the message sent at time \( t_{eo} \).

- Initiation \((i)\) of action \((a)\) at time \( t_{ea} \). The reactive process \( m \) determines the action \((a)\) to be performed, where \( a \in A_m \).

- Termination \((t)\) of action \((a)\) at time \( t'_{ea} \) by the reactive process.

The following terminology is introduced.

\[
\Delta_f = t_{er} - t_{eo} \text{ is called the event recognition latency.}
\]

\[
\Delta_{ea} = t'_{ea} - t_{ea} \text{ is called the action enabling latency.}
\]

\[
\Delta_a = t'_{ea} - t_{ea} \text{ is the duration of the action.}
\]

\[
\Delta_t = \Delta_f + \Delta_{ea} + \Delta_a = t'_{ea} - t_{eo} \text{ is called the event processing time.}
\]

The holding time of a state \( s \) of process \( p \) is defined as the time interval between two consecutive state changes corresponding to events of interest. If \( e_1 \) and \( e_2 \) are consecutive events of interest, when \( e_1 \) is triggered by the state change \((s_1, s_2)\) and \( e_2 \) is triggered by the state change \((s_2, s_3)\), then the holding time of state \( s_2 \) is defined as \( H_s(s_2) = t_{eo}^2 - t_{eo}^1 \).

To ensure the correctness of a monitoring activity it is necessary that all events of interest generated by the target subsystem be processed by the monitoring subsystem. A subsystem with this property will be called observable. Intuitively, observability means that there is enough time to perform the action before the next event of interest. It is assumed that, in general, two or more actions associated with the same active process cannot be performed concurrently because such action is related to the process state reached as a result of the occurrence of the corresponding event and the next event may change salient aspects of the state. In other words, in general, there must be a one-to-one correspondence between a state and an action. For the same reason, requests for actions cannot be queued.

Clearly, an intrusive system with respect to flow control (i.e., one that can suspend an active process) can always be observable. A sufficient condition for observability of a nonintrusive system is that the processing time of any event of interest \( e \), performed by the reactive process \( r \), be smaller than the holding time of the state \( s \) reached as a result of the occurrence of event \( e \): \( \Delta_e \leq H_s(s) \).
The expected rate $\lambda_e$, i.e., the expected number of events occurring per unit of time, is $\lambda_e = 1/E(H_i(s))$. The expected event processing rate $\mu_e$, i.e., the expected number of events processed by a reactive process per unit of time, is $\mu_e = 1/E(\Delta_e)$. A nonintrusive system that is not observable is statistically observable if $\lambda_e < \mu_e$. Statistical observability is a weaker property than observability. It guarantees some form of stability of the monitoring subsystem in the sense that the number of events missed by the monitoring subsystem does not grow without bound (in a queueing theoretical sense). In this paper, the concern is with pure observability, not with statistical observability.

In this case of detection-intrusive monitoring with respect to flow control, the target subsystem may be suspended immediately after the occurrence of event $e$ and this suspension lasts no longer than the time necessary to process the event, namely an interval equal to $\Delta_e$. In this case, a time-dilation effect is observed. Consider an observation interval starting with event $e_1$ and terminating with the occurrence of event $e_2$, and denote the corresponding time intervals as $T_0$ when monitoring is not enabled and as $T_m$ with monitoring enabled. Then $T = T_m - T_0 > 1$. Monitoring is not enabled if no active process sends messages across channels in $\mathcal{E}_{ar}$.

It has been tacitly assumed that it is possible to associate, in a simple and consistent manner, a time with each event. Time-stamping an event corresponds to assigning a number to it, and that number is determined by a physical clock and represents the time at which the event has occurred. Because physical clocks cannot be perfectly synchronized and of infinite resolution it follows that in the model presented here it is sometimes impossible to decide which one of two events has occurred first. Following Lamport [17], the relation "happened-before" is denoted by $\rightarrow$ and is defined on the set of all events in a system to be the smallest ordering of events. Time-stamping with the local clock will be used for ordering of events within a process. This will provide a total ordering of local events. Ordering of events occurring in different active processes will be performed by a reactive process based upon the ordering of messages received from the active processes. If event $e_1$ occurs in process $p_1$ and $e_2$ in $p_2$, the reactive process $r$ will decide that $e_1 \rightarrow e_2$ if the message received from $p_1$, $msg_{e_1}$, arrived before the message from $p_2$, $msg_{e_2}$.

Determining the ordering of events is also complicated by the event recognition latency ($\Delta_e'$). For example, assume the system is observable. Clearly, two reactive processes $r_1$ and $r_2$ may reach opposite conclusions concerning the order of any two events $e_1$ and $e_2$, simply because the event recognition latencies for $r_1$ and $r_2$ are different. Event recognition latency is, in general, associated with communication delay, the time required to send a message from an active process to a reactive process. Call $d(p, r)$ the communication delay from $p$ to $r$. Suppose that two events, $e_1$, and $e_2$, occur simultaneously in active processes $p_1$ and $p_2$, respectively, and two reactive processes, $r_1$ and $r_2$, monitor both processes. If $d(p_1, r_1) < d(p_2, r_1)$, then $r_1$ decides that $e_1 \rightarrow e_2$. If $d(p_2, r_1) > d(p_2, r_2)$, then $r_2$ decides that $e_2 \rightarrow e_1$.

To continue examining the timing attributes of the event-action paradigm new terminology is defined throughout the rest of this section. A primitive event is associated with a change of state of an active process. Compound events are combinations of primitive and compound events, created by using various boolean valued functions, and the intersection, union, and sequence of primitive or compound events. A primitive event has two states, occurred and nonoccurred. A compound event has an additional transient state, representing that the compound event is in the process of occurring. When a compound event is defined as a sequence of primitive events, e.g., $e = e_1e_2e_3 \ldots e_n$, then it is required that

- the component events occur precisely in the specified order,
- no other events of interest occur between any pair of consecutive events in the list.

In general, a compound event can be a combination of primitive events, which occur in some arbitrary orders $e = \mathcal{E}(e_1, e_2, \ldots, e_n)$. If $e_{first}$ is the first component event to occur and $e_{last}$ the last one, then the lifetime of the transient state of the compound event $e$ is $\tau_e = l_{e_{last}}^e - l_{e_{first}}^e$ with $l_{e_{first}}^e$ the corresponding occurrence time. For compound events, $l_{e_{last}}^e$ is used instead of $l_{e_{first}}^e$ in calculating $\Delta_e$.

Recognition of a compound event that is not a sequence of primitive events may be challenging and difficult to implement because the compound event can be in the transient state for a long period of time. During this transient period, the reactive process that will recognize the event has to store the state history since $l_{e_{first}}^e$. From the practical standpoint, it may be necessary to limit the lifetime of a transient state, and if $\tau_e$ exceeds a given value, $\tau_{max}$, the compound event will go undetected.

Another attribute of an event is its range. Following Spezialetti and Kearns [35], three types of events can be defined: local, nonlocal, and global. All the events occurring in the same active process are local events of that process. A local
event can be either a primitive or a compound event. Non-
local events are always compound events consisting of
events occurring in different known active processes. An ex-
treme example of a nonlocal event is a compound event
that consists of events associated with every active process
in the system. Finally, for a global event, unlike local and
nonlocal events, the location of the event of interest is not
known; i.e., the location of the event is not included in the
predicate. A global event usually corresponds to some as-
pect of the overall system state.

Any event of interest (e) has a predicate (p_e) associated
with it. The recognition of the event means that the predi-
cate is true. Examples of predicates are: \( A = 5 \); \( A < 2 \); \( A
> B \); \( A \) is read; \( A = 1 \) and \( B = 2 \) and \( C > D \). Another impor-
tant characteristic of an event is the holding time of the
defining predicate. The time interval over which the predicate
remains true is called the predicate holding time, \( H_p(p_e) \).
An event is called either monotonic or nonmonotonic de-
pending upon the holding time of the associated predicate.
If the event’s predicate holding time corresponds to the
entire lifetime of the monitoring period, then the event is
called monotonic, otherwise the event is called nonmono-
tonic. In order to ensure consistency of the event-action
paradigm, it is necessary that the holding time of a predicate
associated with a nonmonotonic event satisfy the following
condition \( H_p(p_e) > \Delta_e \). When this condition is met the
predicate is referred to as stable. If this condition is violated,
the predicate is referred to as unstable, and the results of the
action (a) associated with event (e) may not be as intended.
Whether stability is required depends upon the action. An
example when stability is not required is when the action is
counting the number of events. An example when stability
is important is when the action is printing the values of \( A
\) greater than 20; i.e., \( A \) is changed from 25 to 5 before the
actual printing takes place.

From this brief description of the attributes of an event,
follows that one should expect difficulties in handling
compound events and especially nonlocal ones [26]. In ad-
dition, special precautions are needed for actions associated
with nonmonotonic events. Methods for specifying events
and actions are discussed in [20].

From the software engineering standpoint, a monitoring
tool hides the details of event detection from the end-user
while actions can be user-defined, based upon a set of primi-
tive actions supported by the system. In general, a monitor-
ing subsystem cannot impose limitations upon the duration
of a user-defined action. In the case of a system without flow
control intrusions, actions of arbitrary duration may lead
to a violation of its observability.

4. AN ARCHITECTURAL MODEL.

This section illustrates the concepts of the event-action
paradigm through a proposed architectural implementa-
tion. A layered model of the monitoring process is devel-
oped and a working definition of intrusion is presented. Fi-
ally, a proposed implementation sketch is given for a
(detection) nonintrusive monitoring subsystem.

4.1. A Layered Model for Monitoring

A layered model, shown in Fig. 3, is the basis for the map-
ing of the event-action paradigm to a monitoring imple-
mentation. Each layer is defined only by its function and
interfaces to the layers above and below. No layer has
knowledge concerning the function of the layers above or
below and the net effect of the function of each layer is the
filtering and compression of event and/or action informa-
tion for the layer above.

The lowest level (level 0) is strictly a part of the active
process, while levels 1–5 represent functions of the reactive
process. Level 0, Event Occurrence, is any change in state
of an active process. Event Recognition (level 1) represents
identification and recording of a change in state and is the
lowest layer of the reactive process.

The Event Filtering layer (level 2) represents the first
level of data compression. It will determine which events
are of interest and pass information about these to the Ac-
tion Enabling layer (level 3). The Action Enabling layer
is responsible for the mapping of events to actions and will,
turn, initiate actions associated with specific events.

The last two layers, the Presentation and the Application
(levels 4 and 5), are named in accordance with the OSI
model of layered network protocols [16]. In this scheme,
the Presentation layer is largely dependent on the Appliance
layer. The Presentation layer will be built up from a
standard library of functions to support the current Appli-
cation layer, while the Application layer will be more flexi-
ble providing any desired functionality to the user. This is
clarified below.

4.2. Intrusive versus Nonintrusive Monitoring

In this section the definition of intrusion provided in Sec-
tion 2 is refined in architectural terms and two possible

<table>
<thead>
<tr>
<th>Level</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Application</td>
</tr>
<tr>
<td>4</td>
<td>Presentation</td>
</tr>
<tr>
<td>3</td>
<td>Action Enabling</td>
</tr>
<tr>
<td>2</td>
<td>Event Filtering</td>
</tr>
<tr>
<td>1</td>
<td>Event Recogntion</td>
</tr>
<tr>
<td>0</td>
<td>Event Occurrence</td>
</tr>
</tbody>
</table>

FIG. 3. Layered model.
mappings of implementations onto the layered model are given: a nonintrusive system, and an intrusive system demonstrating detection intrusion. A distinction between detection intrusion and action intrusion is made.

Intrusion detection is the direct result of the need to incorporate levels of the reactive process within the same processor that is supporting the active process. This form of intrusion may be an undesirable result of monitoring and is illustrated in Section 4.2.2. Intrusive actions, on the other hand, are associated with deliberate steps taken by a reactive process to alter the state of an active process thereby affecting an event of interest.

For the remainder of this paper, intrusive (nonintrusive) monitoring refers to monitoring exhibiting intrusive (nonintrusive) detection. Note that any monitoring system (either detection intrusive or detection nonintrusive) may intrude as the result of an action, i.e., affecting timing, ordering, or state.

In addition, a formal distinction between two classes of resources is now made. The first is all resources used by the target subsystem $T$ and is denoted $r$. $r$ is the hardware being instrumented or monitored. $r$ may, in fact, also be used by $M$, the monitoring subsystem, as is seen. The second class of resources, denoted $\mu$, is those resources used exclusively by $M$ and never by $T$.

4.2.1. Possible nonintrusive layering. Nonintrusive detection, as stated earlier, requires dedicated hardware instrumentation to carry out the passive monitoring of the target subsystem. Figure 4a shows the mapping of a possible nonintrusive implementation onto the layered model.

In this case $\mathcal{E}_{a_p}$ would be the CPU busses of $r$. The change of state of these bus signals constitutes a change of processor state and represents the message $M$. This is to be clearly distinguished from the state of the active process. The event-action paradigm defines events to be changes of state of the active process at a higher level. These changes of processor state may occur without affecting the state of the active processes; e.g., the change of the state of the processor bus does not necessarily constitute the change of any CPU registers or memory locations. A state change of the active process will typically consist of a number of processor state changes. The first step taken by the reactive process is the latching and possible queueing of these processor state changes. Next, the signal patterns are compared against a list of patterns of interest to determine if there was a change of state of the process and if the event corresponding to that process state change is an event of interest.

If an event of interest has occurred, it is mapped to an action that is initiated. Typical actions include the logging of the event occurrence or enabling or disabling the mapping of other events. In the former case, the Mapping and Initiation layer would signal the layer above to log the event and in the latter, the Mapping and Initiation layer may carry out the action itself. Thus this layer embodies the realization of the event-action models' binding of actions to events.

The Presentation layer again depends heavily on the application and in this example it is responsible for logging and communicating Event Occurrence to the Application layer. It could directly log the occurrence on secondary memory or initiate the update of a user interface in the Application layer. An example of the Application layer could be a window-oriented graphic user interface.

The last points to note about Fig. 4a are the columns Process and Processor. These make clear the functional and physical location of each layer. In the Process column, $R$ represents the Reactive processes and $\mathcal{A}$ represents the Active processes. The Processor column shows in which resources the layer is physically executed. The Process column shows that Event Occurrence is associated with active processes, while all the other layers are part of the reactive processes. In addition, the Processor column shows that each layer of the reactive processes uses only the resources of the monitoring hardware ($\mu$); however, this will not be the case for intrusive monitoring.

4.2.2 Possible intrusive layering. Figure 4b shows one possible mapping of an intrusive implementation onto the layered model, where the detection and filtering of events are done without any special hardware. This type of layering is typical of software-instrumented systems that exist today, e.g., [6, 10, 11, 27].

In this example, the upper three layers are identical to the layers presented previously. However, the detection intrusion is evident from the physical location of the lower layers. From the Processor column, it can be seen that in this implementation the Event Filtering and Event Recognition layers, which are parts of the reactive processes and, hence, part of $M$, use the same resources ($r$) as the Event Occurrence layer, which is the active process and, hence, part of $T$. An example would be source level statements added to the program that delays the active process each time the variable "$A$ is assigned a value in order to evaluate the predicate "$A = 5." Upon occurrence of an event of interest, these statements will execute, thus communicating the occurrence to the monitoring hardware. This form of instru-

![FIG. 4. Possible (a) nonintrusive and (b) intrusive layerings.](image-url)
mentation and its drawbacks are well known, and methods for doing such instrumentation exist, e.g., [11].

4.3. Current Intrusive Implementation—CAPS

CAPS (Coding Aid for the PASM System) [21] is the current generation of monitoring hardware and software for the PASM (Partitionable SIMD/MIMD) parallel processor prototype designed and constructed at Purdue University [33, 34]. This includes specialized hardware added to the PASM prototype and software servers, running on PASM and the user’s workstation, to facilitate the transfer of information. CAPS is currently used to assist development of application and system software for PASM, as well as in experimental system evaluation [4, 5, 13].

CAPS illustrates the intrusive layering model shown in Fig. 4b. In the figure τ corresponds to the PASM CPUs and μ corresponds to the other hardware resources shown later in the block diagram in Fig. 5. In CAPS the recognition of events of interest uses the same resources as the active processes, namely PASM CPU cycles.

CAPS consists of a set of dedicated I/O channels and associated hardware and software that facilitate bidirectional information flow between the individual nodes of PASM and a workstation providing the user interface. The information is sent by code added to the user’s program that transmits messages through the dedicated I/O channels. Once the data are sent from the nodes they are combined into a single stream that is sent through a Local Area Network (LAN) to the workstation where they are used to debug and analyze the execution of programs. Currently, only ASCII data are sent and the information is presented to the user in a textual form. However, work is underway to develop various useful graphical user interfaces. The monitoring data are sent by code added to the user’s program, so this system is clearly an intrusive one.

Some degree of hardware instrumentation is necessary to keep intrusion to a reasonable level, however. CAPS was designed to keep the hardware enhancements to the architecture to a minimum cost. The techniques used can be applied to a broad class of parallel machines that includes the PASM parallel processing system, and any system capable of executing in either SIMD mode, MIMD mode, or both. The individual processors of these machines must have the capability to transmit debug/trace information over an I/O channel to a location where the information can be collected and forwarded to a remote site.

4.3.1. Software environment for CAPS. The process of monitoring a program’s execution begins with adding monitoring code statements to the user’s source code. These statements send messages through a dedicated I/O channel to the monitoring system. The code to send the messages may be added either manually by the programmer to gain information on specific aspects of the program or by the system to obtain information on more global issues relating to program execution. The data sent from the nodes are collected and can be presented to the user in a number of forms including textual or graphical. The display can provide information one each node or on the system as a whole.

The graphics workstation in the current CAPS implementation is a Sun 3 [36] running X-windows [31]. These windows allow interactive I/O between the workstation and any processor on the PASM system. Each PASM processor can have its own window on the workstation. Each window can be adjusted to any size and windows may overlap if necessary. A window can act as a terminal allowing access to the processor’s resident monitors that are based on Motorola’s MVMEBUG software [28]. Monitor features include printing memory and register contents, setting program breakpoints, disassembly of segments of memory, and other debugging functions. The programmer may also send displayed information to disk or review information that has scrolled past the screen using standard X-window utilities.

In a typical debugging session, the CAPS server on a Sun opens windows to PASM’s System Control Unit, the host where the parallel program to be executed is being developed (usually a dual processor Vax 11/780 [15]), and to the processors of interest on the PASM system. The window to the host machine is opened through a standard Unix remote login procedure. CAPS is invoked on the Sun workstation and opens windows to the desired PASM processors. The System Control Unit window can be opened via a remote login from the Sun or it can be opened through CAPS and provides access to Unix System V on the System Control Unit. In this configuration, the programmer is able to execute programs on PASM and use program output to help debug the software. The PE’s resident monitors can also help detect errors. The monitoring information can be used to make changes in the source code on the host; then the program can be recompiled assembled, reloaded, and reexecuted on PASM, all without leaving the CAPS environment. This procedure can be carried out on any workstation capable of running X-windows attached to the LAN, as well as remotely from any Internet site with the same capabilities.

The interface is rather flexible and can easily be tailored to specific debugging tasks. During the programming and debugging session, each window can be moved, resized, and iconized (and restored) depending on the needs of the user. Also, any number of windows can be supported.

4.3.2. Architectural support for CAPS. A block diagram showing how the architectural support for CAPS is integrated into PASM is shown in Fig. 5. The CAPS system on the PASM prototype functions as described below.

Each of the CPU boards in the PASM prototype has a serial port intended for terminal I/O with the CPU’s resident monitor to allow for program debugging and control. CAPS uses these I/O channels. Each serial port in the pro-

1Unix is a trademark of AT&T Bell Laboratories.
FIG. 5. Block diagram of the architectural support for CAPS.

totype is connected to the System Monitoring Module, which is controlled by the I/O Processor. The I/O Processor constantly monitors each of the serial ports of the System Monitoring Module for incoming data from any of the PASM CPUs. The System Monitoring Module and I/O Processor together act as the data concentrator. Once a PASM CPU sends a character out its own serial port, the associated port on the System Monitoring Module receives the character and stores the character. The I/O Processor reads the PASM CPU's transmitted character and forms a two-byte packet. The first byte of the packet contains information indicating which of the PASM CPUs sent the character. The second byte of the packet is the seven-bit ASCII character sent. The I/O Processor sends this packet to the System Control Unit via the I/O Processor-System Control Unit parallel port connection. A process running on the System Control Unit reads the packets from its parallel port connection and sends the packets out onto the Ethernet channel to the Sun workstation. Data input through the windows on the Sun is packetized and returned to the appropriate PASM CPU in a similar manner, i.e., Sun to System Control Unit, System Control Unit to I/O Processor, I/O Processor to System Monitoring Module, System Monitoring Module to PASM CPU.

The data concentrator (System Monitoring Module-I/O Processor pair) is necessary because no other component of PASM, e.g., System Control Unit or I/O Processor, has the number of ports required to bring all CPU serial connections together. The I/O Processor controls the System Monitoring Module rather than the System Control Unit because the ports must be serviced in real-time to avoid loss of data. The System Control Units, running Unix V, is not able to service that number of ports without neglecting its other activities or losing data. However, the System Control Unit is capable of handling the single stream of packetized data from the I/O Processor. When the System Control Unit is unable to service the I/O Processor System Control Unit parallel port, the I/O Processor buffers packets in its local memory.

The limitations of the current intrusive implementation are discussed in [21]. These limitations include the event rates that can be sustained, the type of events that can be specified, and the ability to correlate events in different processors.

4.4. Nonintrusive Implementation Sketch

This section shows a possible hardware implementation of each layer specified in Section 4.2.1 for a nonintrusive system. At the global system level, the monitoring subsystem $M$ is implemented with physically distributed monitoring hardware ($\mu$). $\mu$ includes special purpose hardware that is replicated for each node in $\tau$, on which the target subsystem $T$ is executing. This structure is shown in Fig. 6. The only point of contact between $\tau$ and $M$ is at the physical level of the processor's bus or buses (Node Bus). This connection is the realization of $E_{ax}$, the channel through which the active and reactive processes communicate.

Several other aspects of the event–action model can be seen from Fig. 6. The set of channels $E$ are: the active processors' interconnection, $E_{ax}$, connection between the active and reactive processes, $E_{ar}$, and the interconnect for the monitoring subsystem, $E_{m}$. $E_{m}$ is the interconnection network for the target subsystem and is typically accessed by the processing nodes through a network interface, represented by the NI block. For some interconnections it may
be necessary to monitor the state of \( \mathcal{E}_a \) if it cannot be determined through software simulation or if the interconnection exhibits nondeterministic behavior. This capability is provided through the Network Monitor block labeled NM (one per system). The Node Bus, which in this case is also \( \mathcal{E}_{a,r} \), connects the processor to its I/O, memory, and network interface, and may in fact, be multiple buses. Finally, \( \mathcal{E}_r \), consists of a communication media (e.g., Ethernet). In addition there is hardware synchronization support for the Special Purpose Hardware Monitor, SPHM, units (one per node) and the Central Monitoring Facility, CMF (one per system). This synchronization support will include a clock line to provide global time.

The CMF provides control of the SPHM units, helps provide coordination between them, and supports the user interface to the monitoring system. Through supporting the user interface, the CMF provides some of the functionality of the Presentation and Application layers. In addition, in order to evaluate event predicates spanning a number of nodes, the SPHM units must have some concept of nonlocal state and some form of cooperation to handle nonlocal actions. Through providing coordination between SPHM units, the CMF provides support for nonlocal events and actions and, thereby, some of the functionality of the Event Filtering and Action Enabling layers. The CMF will also control the downloading of the event and action information or "program" for the SPHMs.

At the core of this design, however, is the SPHM. The SPHM is responsible for the identification of events, mapping of events onto actions, and carrying out some actions. Due to concurrent events and actions, the SPHM units must be capable of working in concert through \( \mathcal{E}_r \), possibly with the aid of the CMF. Figure 7 shows a simplified functional model of the SPHM unit.

The **Front End** can be thought of as fast parallel comparison logic which holds a large list of events in an **Event Memory**. Upon matching a change in state of the active process with an event in the Event Memory, the **Back End** is notified. The Back End includes a processor of equal or lesser power than a node of the target subsystem being monitored. Once the event has been identified, the Back End carries out the action specified in the **Action Memory**. The Back End is also responsible for communication with other SPHM units and the CMF. The relationship between the Front and Back Ends is governed by \( H_1(s) > \Delta_r \); the event processing...
time must be less than the event holding time to avoid queueing or missing events. To assure this, it is intended that the Front End process events fast enough and provide a high enough degree of data compression so that the Back End needs process as little data as possible. Obviously some maximum event rate must be tolerated by the user to prevent the cost of the monitoring hardware from exceeding some “reasonable” limit.

A more detailed functional block diagram of the proposed SPHM unit that maps the Front End/Back End model onto hardware is shown in Fig. 8.

As bus signals are latched, they are queued (in Q) for comparison against patterns in an associative memory MP (Pattern Memory). The elements of Q represent processor state changes and are shifted through Q to be compared either as single patterns or as groups. Single patterns that may represent events of interest would include, for example, entry into or exit from a code segment, i.e., a read from a certain memory location. A message signifying the occurrence of these single pattern events would be sent to the SPHM Control. A typical state change at the process level, however, will consist of several processor state changes. Hence the Match circuit must be capable of checking for groups of patterns of interest shifting through Q. An example of a group of patterns which represent an event of interest would be the setting up and use of a communication channel in ea. Again the occurrence of these groups of processor state changes is sent to the SPHM Control and to the State and Event Controller, K.

The State and Event Controller, with the aid of the SPHM Control, is responsible for the identification of any event involving the evaluation of a predicate. These predicates specify more complex process level state changes and involve some concept of the current state of the process and even the state of nonlocal nodes. The predicates are held in ME, Complex Events Memory. As processor state changes occur, which are components of predicates, the information is kept in the partial state memory MVS (i.e., the transient of the event is held in MVS in order to evaluate the predicates). An example would be the change of value of one element of an array where a predicate has been defined that depends on all elements of the array. In addition, MVS holds a subset of the local and nonlocal state of the active process updated by the SPHM Control. Finally, MA, the action memory, specifies the actions to be taken by the SPHM control on the identification of events of interest.

The arrows at the top of Fig. 8 delineate the levels of the monitoring hierarchy within this design. Level 0, Event Occurrence, is shown on C_a,. Level 1 is the latching and queueing of the signals by Q. Level 2, Event Filtering, is shown spanning the Match circuit, and the State and Event Controller K to represent the identification of both processor state and process state changes. Levels 3 and 4 are both shown residing within the SPHM Control with level 4 also including other SPHM units and the CMF. This is because the SPHM is responsible for carrying out some actions and because the Presentation layer needs to include the CMF and all other SPHM units. The existence of nonlocal events also creates the need to have the functionality of the Event Recognition and Action Enabling layers with regard to concurrent events and actions also residing across SPHM units and on the CMF. Finally, not shown in the figure, is level 5 of the Application layer. This layer will reside solely within the CMF.

5. CONCLUSIONS

This paper presented several aspects of the multidimensional problem of providing monitoring support for debugging and performance analysis of software for distributed and parallel systems. A formal event-action model at the process level and a layered architectural model were introduced. The event–action model was applied in the development of abstract layered models for both intrusive and nonintrusive systems.

Currently, an intrusive monitor similar to the structure presented in Sections 4.2.2 and 4.3 is available on the PASM system prototype at Purdue. Continuing efforts are focused on the implementation of a nonintrusive monitor for PASM. This monitor has the structure shown in Sections 4.2.1 and 4.4.

The relation of the existing intrusive and proposed nonintrusive monitoring schemes to the event–action and layered architecture models was discussed. This demonstrated how the models could be used in the development of actual software development tools.

ACKNOWLEDGMENTS

A preliminary version of portions of this paper was presented at Compusac 89. With respect to the current implementation of the CAPS system, the authors gratefully acknowledge many useful discussions with, and contributions from, Samuel A. Fineberg, Wayne G. Nation, Edward C. Bronson, Pierre H. Pero, Thomas Schwederski, and Henry G. Dietz.
REFERENCES


DAN C. MARINESCU received an M.S. degree in electrical engineering and computer science in 1969 from U.C. Berkeley, and an M.S. degree in
electrical engineering in 1965 and the Ph.D in electrical engineering and computer science in 1976, both from the Polytechnic Institute Bucharest, Romania. Dr. Marinescu has been an associate professor in the Computer Science Department at Purdue University since 1984. He is conducting research in distributed systems and networking, performance evaluation, real-time systems, and parallel computing, and has published over 40 papers in these areas. Dr. Marinescu is a member of the IEEE Computer and Communication Societies and a member of the Association of Computing Machinery (ACM).

JAMES E. LUMPP, JR., received his B.S. and M.S. degrees in electrical engineering from Purdue University, West Lafayette, Indiana, in 1988 and 1989, respectively, and is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Iowa, Iowa City, Iowa. His research interests include parallel processing, computer architecture, operating systems, and performance monitoring. Mr. Lumpp is a member of Eta Kappa Nu, the IEEE Computer Society, and the Association of Computing Machinery (ACM).

THOMAS L. CASAVANT received the B.S. degree in computer science in 1982, the M.S. degree in electrical and computer engineering in 1983, and the Ph.D. degree in electrical and computer engineering in 1986, all from the University of Iowa. From 1986 to 1989, Dr. Casavant was on the faculty of the School of Electrical Engineering at Purdue University, where he also served as Director of the Parallel Processing Laboratory. He is currently an assistant professor on the faculty of the Department of Electrical and Computer Engineering at the University of Iowa. His research interests include parallel processing, computer architecture, programming environments for parallel computers, and performance analysis. Dr. Casavant is a member of the IEEE Computer Society and the Association of Computer Machinery (ACM).

HOWARD JAY SIEGEL received two B.S. degrees from the Massachusetts Institute of Technology (MIT), and M.A., M.S.E., and Ph.D. degrees from Princeton University. He is a professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue University. He has coauthored over 125 technical papers, coedited four volumes, authored one book (Interconnection Networks for Large-Scale Parallel Processing), consulted, given tutorials, and prepared videotape courses, all on parallel processing networks and systems. He was General Chairman of the "3rd Int'l Conf. on Distributed Computing Systems" (1987), Program Co-chairman of the "1983 Int'l Conf. on Parallel Processing," and General Chairman of the "15th Ann. Int'l Symp. on Computer Architecture" (1988). He is a Fellow of the IEEE and a member of the Association of Computing Machinery (ACM).