1988

An Event-Action Model and Associated Architecture for Monitoring Parallel and Distributed Systems

Dan C. Marinescu
James E. Lumpp
Thomas L. Casavant
Howard Jhay Siegel

Report Number:
88-817


This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
AN EVENT-ACTION MODEL AND ASSOCIATED ARCHITECTURE FOR MONITORING PARALLEL AND DISTRIBUTED SYSTEMS

Dan C. Marinescu
James E. Lumpp, Jr.
Thomas L. Casavant
Howard Jay Siegel

Computer Sciences Department
Purdue University
CSD-TR-817
October 1988
AN EVENT-ACTION MODEL AND ASSOCIATED ARCHITECTURE
FOR MONITORING PARALLEL AND DISTRIBUTED SYSTEMS

Dan C. Marinescu*
James E. Lumpp, Jr.†
Thomas L. Casavant†
Howard Jay Siegel†

*Computer Science Department
†School of Electrical Engineering, Parallel Processing Laboratory
Purdue University
West Lafayette, IN 47907

October 1988

Abstract

Several aspects of the multidimensional problem of providing monitoring support for
debugging and performance analysis of distributed and parallel systems and parallel
applications are presented in this paper. The key elements of the solution to the
problem presented are a formal event-action model at the process level and a layered
architectural model. The application of the event-action model to the development of
the abstract layered architectural model, and the use of of this abstract layered archi­
tectural model in the specification of possible intrusive and non-intrusive implementa­
tion of instrumentation are shown.
1. INTRODUCTION TO MONITORING AND DEBUGGING OF PARALLEL AND DISTRIBUTED SYSTEMS

Instrumentation of conventional multiuser computing systems for performance monitoring and debugging is a fairly well understood subject. The operating system has to allocate different resources (e.g., CPU(s), memory, I/O bandwidth) according to pre-established algorithms, hence it has to monitor the status of each component of the system and the amount of each resource consumed by any activity. At the foundation, we have an event/action (E/A) paradigm. A typical event is the interrupt which triggers activation of the scheduler (e.g., time-slice completion) and the typical actions are counting (e.g., update the CPU usage counter of the process to be suspended) and dispatching the process with the highest priority from the ready to run queue. The data collected by the performance monitoring environment is used for accounting and by a system programmer for tuning (e.g., re-adjust the static priority), for bottleneck analysis (e.g., determine that the system needs more real memory, because paging is excessive), for system reconfiguration (e.g., redistribute system files which are frequently accessed over multiple disks), and so on. Occasionally, in such a system, a user defines events (e.g., tracing specific operations like system calls or branch instructions) for debugging purposes.

As early as 1975, McDaniel has proposed a kernel instrumentation for distributed environments [McD75]. The work on some of the distributed systems designed in the early 1980's include some performance monitoring facility, usually a software monitor, see, for example, Cheriton [CHZ85] for the V kernel and Powell and Miller [PoM83] for DEMOS/MP. Systems for interactive debugging of a distributed computational environment were considered as a separate issue [Sch81]. Several monitors for distributed systems based on an Ethernet or a Hyperchannel are being built. We note here the work of W.B. Watson at Livermore and the Real-Time monitoring systems at Ohio State (P. Ogle and K. Schwan) [Van87]. Miller at the University of Wisconsin is currently developing IPS2, based upon his experience with the Distributed Program Monitor [MiM86]. Work in progress is reported also at HP Labs (Spinn and Ratii) and 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 [LeM87]. Bates at University of Massachusetts [Bat88, BaW83] has developed a sophisticated environment for High Level Debugging, based upon the EBBA paradigm (Event Based Behavioral Abstraction). Dongarra and Sorensen at Argonne National Laboratory have developed a
package, SCHEDULE for the analysis of parallel FORTRAN programs using dependency graphs [DoS87]. Another set of useful tools developed at Argonne consists of MAPI and MAPA that use trace files [BrD88]. 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 program [Cou87].

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 non-intrusive monitoring and observability are examined using this model. The challenges and the major concepts related to monitoring of such systems are discussed. In addition these concepts are mapped to monitoring implementations by way of a hierarchical model of the monitoring system.

While the paper addresses fundamental models for parallel and distributed systems monitoring, this work is motivated by the need for practical monitoring systems. Specifically, in our research on developing instrumentation for the 3D-processor PASM prototype [FiC88, SiS81] we found a need for defining the theory underlaying the monitoring systems. Thus one purpose of the theoretical study presented here is to provide a basis upon which a practical monitoring system may be designed.

Section 2 introduces a process level formal model for the event-action paradigm. The way in which timing is considered in 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 [ChM88]. 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 [LeR85], kernels of operating systems, as well as different control systems [MaB84], 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 which 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.

The model introduced in this paper identifies two subsystems, a target subsystem which provides the basic computational functionality of the system for applications, and a monitoring subsystem which supports the monitoring and debugging functions needed for application support and system-oriented performance analysis.

Consider a message oriented system \( A \) consisting of a finite set of processes, \( \mathcal{P} \), and a finite set of message channels, \( \mathcal{C} \). 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 it is assumed that 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 being sent presently along the channel.

In our 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 [ChL85], 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 \( \langle p, s, s', M, c \rangle \) 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 $\mathcal{A}_m = \{p_{m_1}, p_{m_2}, \ldots, p_{m_p}\}$,
- the set of events $E_m$ that $m$ recognizes $E_m = \{e_{m_1}, e_{m_2}, \ldots, e_{m_e}\}$,
- the set of actions $A_m$ that $m$ is capable of performing $A_m = \{a_{m_1}, a_{m_2}, \ldots, a_{m_a}\}$.

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 $e_i$, the reactive process $m$ determines what action $a_i \in A_m$ has to be performed, performs action $a_i$ and then returns to the suspended state.

In general, only a distinguished subset of events have actions associated with them. Such events are called events of interest. An equivalent way of presenting this issue is to consider that a null action, $a_0$, is associated with any event that does not belong to the set of events of interest. However, in this paper the former convention will be used.

The set of channels is partitioned into three disjoint subsets $\mathcal{C} = \mathcal{C}_a \cup \mathcal{C}_r \cup \mathcal{C}_{a,r}$. Channels $c \in \mathcal{C}_a$ connect only active processes, channels $c \in \mathcal{C}_r$ connect only reactive processes, and channels $c \in \mathcal{C}_{a,r}$ 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 $\mathcal{C}_{a,r}$. Hence, $\mathcal{S} = (\mathcal{T}, \mathcal{M}, \mathcal{C}_{a,r})$.

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

(a) A monitoring system $\mathcal{M}$ 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 system \( \mathcal{M} \) is non-intrusive if it does not induce changes of state of \( \mathcal{F} \) and if it does not alter the timing or ordering of events in \( \mathcal{F} \).

The interactions between \( \mathcal{F} \) and \( \mathcal{M} \) occur at the time of event detection and possibly during 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 non-intrusive event detection. Actions may also be intrusive or non-intrusive. An intrusive action may alter the timing, ordering of events, the state of the active process, or all three. An example of non-intrusive action might be simple event counting.

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

If the flow of information (on \( G_{\mathcal{F},\mathcal{M}} \)) between the two subsystems is unidirectional from \( \mathcal{F} \) to \( \mathcal{M} \), it is possible to experience either intrusive or non-intrusive event detection, and non-intrusive actions. Note that 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. If the flow is bidirectional, then actions can be intrusive.

An example of a monitoring subsystem with intrusive event detection and unidirectional information flow is presented next. When an event occurs in an active process \( p \), the process is suspended due to non-receipt 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 \( \mathcal{F} \). The timing of events in \( p \) is changed, and possibly the global ordering of events in \( \mathcal{F} \) is altered (i.e., the ordering of the suspended processes with respect to other processes in \( \mathcal{F} \)).

A typical operating system performs this form of intrusive monitoring, 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.

Most real-time control systems perform one form or another of non-intrusive monitoring \([\text{Mar87}]\). If action \( a_i \) is associated with event \( e_i \), and if \( e_i \) and \( e_{i+1} \) are two consecutive events, then in a real-time system the duration of action \( a_i \), \( d(a_i) \), and the inter-event time, \((t_{e_{i+1}} - t_{e_i})\), are typically related by the following condition \( d(a_i) < t_{e_{i+1}} - t_{e_i} \).
Debugging tools [LeM87] generally require that \( \mathcal{M} \) exercises some form of control over \( \mathcal{J} \). 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 have limitations for debugging real-time applications.

Performance measuring systems can be intrusive or non-intrusive too. Whenever it is required that the measuring system does not perturb the system being measured, the non-intrusive monitoring solution is mandatory. In this case, non-intrusive event detection is a minimum requirement. Unfortunately, non-intrusive 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 SEEUCUBE package [Cou87], the SCHEDULE package [DoS86], the MAPI and MAPA tools [BrD88], and the Execution and Debugging Environment for PASM [NaF88].

The model described above indicates some of the major difficulties encountered in non-intrusive monitoring. Non-intrusive monitoring requires that the target system, \( \mathcal{J} \), and the monitoring system, \( \mathcal{M} \), do not share any system resources other than the channels in \( C_{a,r} \). Furthermore, \( C_{a,r} \) must be unidirectional with no acknowledgments from \( \mathcal{M} \) to \( \mathcal{J} \). This is a very strict requirement and implies that non-intrusive 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: BASIC ASSUMPTIONS

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 \( t_{e}^{o} \). 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 \( t_{e}^{r} \). The event \( (e) \) is recognized when the reactive process \( m \) receives the previous message.

- **Initiation** \( (i) \) of action \( (a) \) at time \( t_{e,a}^{i} \). The reactive process \( m \) determines the action \( (a) \) to be performed, where \( a \in A_{m} \).
- Termination \((t)\) of action \((a)\) at time \(t_{e,a}\) by the reactive process.

The following terminology is introduced.

\[
\Delta_e^r = t_e^r - t_e^o
\]

is called the event recognition latency.

\[
\Delta_{e,a}^l = t_{e,a}^l - t_e^r
\]

is called the action enabling latency.

\[
\Delta_{e,a}^d = t_{e,a}^d - t_{e,a}^l
\]

is the duration of the action.

\[
\Delta_e = \Delta_e^r + \Delta_{e,a}^l + \Delta_{e,a}^d = t_{e,a}^l - t_e^o
\]

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 \(H(s_2) = t_{e_2}^o - t_{e_1}^o\).

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 system 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. 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 non-intrusive 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 after the occurrence of event \(e\): \(\Delta_e \leq H(s)\).

The expected event rate \(\lambda_e\) - i.e., the expected number of events occurring per unit of time, is \(\lambda_e = \frac{1}{E[H(s)]}\). The expected event processing rate \(\mu_e\) - i.e., the expected number of events processed by a reactive process per unit of time, is \(\mu_e = \frac{1}{E[\Delta_e]}\). A non-intrusive system that is not observable is \textit{statistically observable} if \(\lambda_e < \mu_e\). Statistical observability is a weaker property than observability. It guarantees some form of stability of the monitoring system in the sense that the number of events missed by the monitoring system does not grow without bound (in a queueing theoretical sense). In this paper, the concern will be with pure observability, not with statistical observability.

In the case of detection intrusive monitoring with respect to flow control, the target system may be suspended immediately after the occurrence of event \(e\) and this suspension lasts no
longer than the time to process the event, namely an interval equal to $\Delta_t$. 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 $T_m$ with monitoring enabled. Then $\theta = \frac{T_m}{T_0} > 1$. Monitoring is not enabled if no active process sends messages across channels in $C_{a,r}$.

So far 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 an event, 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 [Lam78] the relation "happened-before" is denoted by "$\rightarrow$" and is defined on the set of all events in a system to be the smallest relation satisfying three conditions.

1. If $e_1$ and $e_2$ are events in the same process $p$, and event $e_1$ happens before $e_2$ (according to the local clock of process $p$), then $e_1 \rightarrow e_2$.
2. If the event $e_1$ is associated with sending of a message from process $p$, and event $e_2$ is associated with receiving the message by process $q$, then $e_1 \rightarrow e_2$.
3. The "$\rightarrow$" relation is transitive.

Two events such that $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_1$ are said to be concurrent.

In general, reactive processes are concerned with 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 $p_2$, $msg_{e_2}$.

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 \( d(p_1,r_2) > d(p_2,r_2) \), 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**, as defined in Section 2, is associated with a change of state of an active process. **Compound events** are combinations of primitive 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 non-occurred. 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., \( e = e_1 e_2 e_3 \cdots e_q \) 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 order \( e = E(e_1,e_2,e_3,\ldots,e_q) \). If \( e_{\text{first}} \) is the first component event to occur and \( e_{\text{last}} \) the last one, then the lifetime of the transient state of the compound event \( e \) is \( \tau_e = t_{\text{last}}^e - t_{\text{first}}^e \) with \( t^0 \) the corresponding occurrence time. For compound events \( t_{\text{last}}^e \) is used instead of \( t_e^0 \) in calculating \( \Delta_e \).

Recognition of a compound event which 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 which will recognize the event, has to store the state history since \( t_{e_{\text{first}}}^0 \). From the practical standpoint, it may be necessary to limit the lifetime of a transient state, and if \( \tau_e \) exceeds a given value, \( \tau_{\text{max}} \), the compound event will go undetected.

Another attribute of an event \( e \) is its **range**. Two types of events can be defined: local and non-local. 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 active processes. An extreme example of a non-local event is a global event that consists of events associated with every active process in the system.
Any event of interest \((e)\) has a predicate \((p_e)\) associated with it. The recognition of the event means that the predicate is true. Examples of predicates are: \(A = 5\); \(A < 2\); \(A > B\); \(A = 1\) and \(B = 2\) and \(C > D\). It follows that an event has another important characteristic, namely, its stability. An event can be either monotonic or non-monotonic depending upon the holding time of the associated predicate. If the predicate remains true for the entire lifetime of the monitoring period, then the compound event is monotonic. If the predicate remains true only for a predicate holding time, \(H_p(p_e)\), and then becomes false, then the event \((e)\) is non-monotonic. In order to ensure consistency of the event-action paradigm, it is necessary that the holding time of a predicate associated with a non-monotonic event satisfies the following condition \(H_p(p_e) > \Delta_e\). If this condition is violated, 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 necessary is when the action is printing the values of \(A\) greater than 20, i.e., \(A\) is changed from 20 to 5 before the actual print takes place.

From this brief description of the attributes of an event, it follows that one should expect difficulties in handling compound events and especially non-local compound events [MiC88]. Special precautions are needed for actions associated with non-monotonic events.

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 primitive actions supported by the system. In general, a monitoring system 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 violation of its observability as well as to inconsistent states of \(S\).

4. AN ARCHITECTURE MODEL

This section will illustrate the concepts of the event-action paradigm through a proposed architectural implementation. A hierarchical model of the monitoring process is developed and a working definition of intrusion is presented. Finally, a proposed implementation sketch is given for a non-intrusive (with respect to detection) monitoring system.
4.1 A Monitoring Hierarchy

A hierarchical model is the basis for the mapping of the event-action paradigm to a monitoring implementation. Figure 1 shows this hierarchy.

![Pure Monitoring Hierarchy](image)

**Figure 1. Pure Monitoring Hierarchy**

Information in this model flows strictly upward with each layer 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 information 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. Level 1, Event Recognition, 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 Action Enabling layer (level 3). The Action Enabling layer is responsible for the mapping of events to actions and will, in turn, initiate actions associated with specific events.
The last two layers of the hierarchy, the Presentation layer and the Application layer (levels 4 and 5) are named in accordance with the OSI model of layered network protocols [ISO83]. In the present hierarchy, the Presentation layer is largely dependent on the Application layer. The Presentation layer will be built up from a standard library of functions to support the current Application layer, while the Application layer will be more flexible providing any desired functionality to the user. The functions of the these layers are clarified below.

4.2 Intrusive Versus Non-Intrusive Monitoring

In this section the definition of intrusion provided in Section 2 is refined in architectural terms and two possible mappings of the monitoring hierarchy onto implementations are given: a non-intrusive system, and an intrusive system demonstrating detection intrusion. In addition, a distinction between detection intrusion and action intrusion is now made.

Intrusion detection is the direct result of the need to incorporate levels of the reactive process within the processor supporting the active process. That is if the activities of the reactive process require the resources of the active process. This form of intrusion may be an undesirable result of monitoring and will be 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 monitoring will refer to monitoring exhibiting intrusive detection. Note that any monitoring system (either detection intrusive or detection non-intrusive) may intrude as the result of an action, i.e., affecting timing, ordering, or state.

4.2.1 Possible non-intrusive layering

Non-intrusive detection, as stated earlier, requires dedicated hardware instrumentation to carry out the passive monitoring of the target system. Figure 2 shows the mapping of the monitoring hierarchy onto a possible non-intrusive implementation demonstrating how this hardware can be layered.

The change of state of the bus signals of the CPU upon which the active process is running constitutes a change of processor state. 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 process, 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 possibly queueing of these processor state
changes. Next, the signal patterns are compared against a list of events of interest to determine
if there was a change of state of the process and if the event generated by that process change
of state is an event of interest.

<table>
<thead>
<tr>
<th>Process</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Graphic Interface</td>
</tr>
<tr>
<td>R</td>
<td>Logging and/or Communicating Occurrence</td>
</tr>
<tr>
<td>R</td>
<td>Event Mapping and Action Initiation</td>
</tr>
<tr>
<td>R</td>
<td>Match Circuit</td>
</tr>
<tr>
<td>R</td>
<td>Latch Signals</td>
</tr>
<tr>
<td>A</td>
<td>Bus Signals Change</td>
</tr>
</tbody>
</table>

Figure 2. Possible Non-Intrusive Layering

If the event is of interest, it is mapped to an action that is initiated. Typical actions could
be the logging of the event occurrence or enabling or disabling the mapping of other events. In
the former case, the Mapping and Initiate layer would signal the layer above to log the event
and in the latter, the Mapping and Initiate layer may carry out the action itself. Thus this
layer embodies the realization of the event-action model’s 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 inter-
face in the Application layer. An example of the Application layer could be a window-oriented
graphic user interface.
The last points to note about Figure 2 are the columns Process and Processor. These make clear the functional and physical location of each layer. In each column, R represents the Reactive process/processor and A represents the Active process/processor. The Process column shows that Event Occurrence is associated with the active process, while all the other layers are activities of the monitoring process. In addition, the Processor column shows that each layer of the reactive process lies within the reactive system, however, this will not be the case for intrusive monitoring.

4.2.2 Possible intrusive layering

Figure 3 shows one possible mapping of the monitoring hierarchy onto an intrusive implementation where the detection and filtering of events is done without any special hardware. This type of layering is typical of software-instrumented systems, which exist today [BrD88, Cou87, DoS86, NaF88].

![Figure 3. Possible Intrusive Layering](image)

In this example, the upper three layers will be 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 strictly activities of the reactive process, lie within the...
active processor. An example would be source level statements added to the program which defines the active process after each time the variable "A" is assigned a value in order to recognize the event of interest, "A=5." Upon occurrence of an event of interest, these statements will execute, thus communicating the occurrence to the monitoring hardware.

This form of instrumentation and its drawbacks are well known, and methods for doing such instrumentation exist, e.g. [DoS86]. The more ambitious goal, which is now examined in detail, is the design of a detection non-intrusive system.

4.3 Non-Intrusive Implementation Sketch

This section shows a way for each layer specified in Section 4.2.1, for a non-intrusive system, to be mapped to hardware. At the global system level, there is monitoring hardware replicated for each node in the target parallel system. This structure is shown in Figure 4. The only point of contact between $\mathcal{F}$ and $\mathcal{M}$ is at the physical level of the processor's bus or buses (Node Bus). This connection is the realization of $\mathcal{C}_{a,r}$, the channel through which the active and reactive processes communicate.

The set of channels $\mathcal{C}$ are: the active processors' interconnection $\mathcal{C}_a$, connection between the active and reactive processes $\mathcal{C}_{a,r}$, and the interconnect for the monitoring system $\mathcal{C}_r$. $\mathcal{C}_a$ is the interconnection network for the target system and is typically accessed by the processing nodes through a network interface, represented by the Network block. For some interconnections it may be necessary to monitor the state of $\mathcal{C}_a$ if it cannot be determined through software simulation. This capability is provided through the block labeled NM (Network Monitor). The Node Bus connects the processor to its I/O, memory and network interface, and may, in fact, be multiple buses. Finally, $\mathcal{C}_r$ consists of a high bandwidth communication media (e.g. an Ethernet), in addition to hardware synchronization support for the Special Purpose Hardware Monitor, SPHM boards and the Central Monitoring Facility, CMF.

The user interface is supplied through the CMF, that is also responsible for coordinating the SPHM boards. In doing so, the CMF provides parts of the roles of the Presentation and Application layers. However, the CMF could also be involved in the Event Filtering and Action Enable layers with respect to non-local actions. A number of event predicates can be defined that span a number of nodes of the target system and thereby multiple SPHM boards.
Figure 4. Non-Intrusive Monitoring System Overview
The CMF may intervene to identify these concurrent events, possibly by providing storage for non-local state, and to assist in carrying out concurrent actions.

At the core of this design is the SPHM. This board 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 boards must be capable of working in concert through $C_r$, possibly with the aid of the CMF. Figure 5 shows a simplified functional model of the SPHM board.

![Figure 5. Functional Diagram of SPHM Board](image)

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 system 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 boards and the CMF. The relationship between the front and back ends is governed by $H_s(s) > \Delta_e$; the event processing time must be less than the event holding time to avoid queueing of events or missing of events. To assure this, it is intended that the Front End be fast enough and capable of a
high enough degree of data compression that the Back End can work with 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 the cost of a node of the target system.

A more detailed functional block diagram of the proposed SPHM board which moves the Front End/Back End model onto hardware is shown in Figure 6.

As bus signals are latched, they are queued for comparison against patterns in an associative memory $M_P$ (Pattern Memory). Some of these processor state changes correspond to events of interest and are passed directly to the SPHM Control for Action Enabling. Such events would include the entry and exit from code segments. A typical state change at the process level, however, will consist of several processor state changes and may also involve the
evaluation of a predicate. All of this is provided by the State and Event Controller, K and the SPHM Control.

The identification of these process level state changes (events of interest) is accomplished through another comparison of groups of patterns. Event-defining groups of processor state changes are held in the $M_E$ (Event Memory). As prerequisite processor state changes occur, the information is kept in the partial state memory $M_{VS}$ (i.e., the transient of the event is held in $M_{VS}$ in order to evaluate the predicates). In addition, $M_{VS}$ holds a subset of the state of the active process. This information is used in the recognition of more complex events that need to be processed by the SPHM Control.

The top of Figure 6 shows the locations of the levels of the monitoring hierarchy within this design. Level 0 Event Occurrence is shown on $C_{a,r}$. Level 1 is the latching and queuing of the signals by $Q$. Level 2, the 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 boards and the CMF. This is the case because the Presentation layer needs to be spread across SPHM boards. The existence of non-local 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 boards 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 distributed and parallel systems and applications. The key element of the solution to the problem presented are a formal event-action model at the process level, and a layered architectural model. We have shown applications of the event-action model in the development of the abstract layered model. We have shown also the application of the layered architecture model for the specification of possible intrusive and non-intrusive implementations.

The most useful result is the structured approach to constructing non-intrusive monitoring systems. This characteristics is an absolute requirement for debugging of real-time applications, and highly desirable for tightly coupled, highly parallel applications in which intrusion
could cause excess serialization. Currently, an intrusive monitor with the structure presented in Section 4.2.2 is available on the PASM system prototype at Purdue. Continuing efforts are focused on the implementation of a non-intrusive monitoring system for PASM. This monitor has the structure shown in Sections 4.2.1 and 4.3.

REFERENCES


