Consistent Detection of Global Predicates*

Robert Cooper Keith Marzullo

Department of Computer Science
Cornell University
Ithaca NY 14853, USA

A fundamental problem in debugging and monitoring is detecting whether the state of a system satisfies some predicate. If the system is distributed, then the resulting uncertainty in the state of the system makes such detection, in general, ill-defined. This paper presents three algorithms for detecting global predicates in a well-defined way. These algorithms do so by interpreting predicates with respect to the communication that has occurred in the system. Briefly, the first algorithm determines that the predicate was *possibly* true at some point in the past; the second determines that the predicate was *definitely* true in the past; while the third algorithm establishes that the predicate is *currently* true, but to do so it may delay the execution of certain processes.

Our approach is in contrast to the considerable body of work that uses *temporal* predicates (i.e., predicates expressed over process histories) for distributed monitoring. Temporal predicates are more powerful, but also more complex to use. In many cases, the condition that the programmer wishes to monitor is simply and intuitively viewed as a predicate over the "instantaneous" state of the system. Using the possibly/definitely/currently interpretation such a predicate becomes well-defined, without requiring it to be recast using temporal formulas. Further, our algorithms may be more efficient than techniques that use a notion of explicit time or process histories. Section 1 specifies the protocols and Section 2 gives an outline of their operation.

This work arose as part of *Meta*, a toolkit supporting distributed system monitoring and control. The architecture addressed by Meta is more general than that needed for debugging, in that we are also concerned with several monitoring components reacting in a consistent and fault-tolerant manner. Section 3 discusses how the algorithms of this paper can be used to provide a breakpoint and tracepoint facility for Meta.

1 Uncertainty in Distributed Systems

Suppose we have a set of n processes $P = \{p_1, p_2, \ldots, p_n\}$ that communicate only by message passing, and we have a *monitoring process* p_0 that wishes to determine if the state of P satisfies a global state property Φ. In general, the uncertainty in the state of the system makes such determination impossible. For example, if each process has an integer-valued variable x and Φ is "the median value of x is greater than 10", then p_0 would have to know the simultaneous value of at least half of the variables to determine whether Φ holds.

Following are two ways that Φ can be changed so that it is meaningful in the face of uncertainty:

GP1: There is an execution of P consistent with its observed behavior such that Φ was true at a point in that execution. We will refer to this property as *possibly* Φ.

GP2: For *all* executions of P consistent with its observed behavior, Φ was true at some point. We will refer to this property as *definitely* Φ.

Both of these conditions refer to some past state or states of P; a third property can be detected if it is acceptable to temporarily block processes:

GP3: Φ currently holds, and there is a logical execution of the unblocked system such that Φ holds. We will refer to this property as *currently* Φ.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, Contract N00140-87-C-8904, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

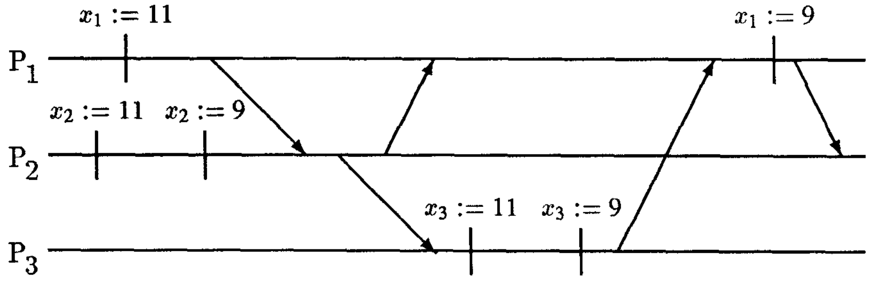

Figure 1: Φ = (at least two of the three x variables are greater than 10).

Our protocols are based on each process in P informing p_0 when a local event changes some local state on which Φ depends. We say that such events potentially change Φ. Then, p_0 constructs the partial order of events, represented compactly as vectors of timestamps [7], in order to determine whether *possibly* Φ or *definitely* Φ held. The first two conditions are illustrated in Figure 1. In this space-time diagram, three processes p_1, p_2, p_3 have variables x_1, x_2, x_3 respectively. *Possibly* $(x_1 > 10) \wedge (x_2 > 10)$ and *definitely* $(x_1 > 10) \wedge (x_3 > 10)$ hold in this execution.

These three specifications provide different ways to manage the uncertainty of debugging in a distributed environment. The distinction between *possibly* and *definitely* arises from the relative nature of time in a distributed system [4]. To an omniscient observer with a global clock, two events that we regard as concurrent (i.e. $a \not\rightarrow b \wedge b \not\rightarrow a$) will have some total order. We cannot determine this total order in any practical way, but determining that some predicate was *possibly* true provides us with valuable information about the behavior of the program. For instance, if Φ identifies some erroneous state of the program, *possibly* Φ holding almost certainly indicates a bug, even if the predicate was not true during this execution as seen by our omniscient observer. Further, for GP2, note that our specification does not include *all* cases in which the predicate holds, but only those cases detectable by observing the messages sent by each process in the system. In particular, if no communication takes place, no predicate will be detected.

Additional uncertainty arises from the delays in monitoring, and effect of monitoring on the computation at hand. When detecting a global predicate, there is in general an unavoidable delay between local predicates becoming potentially true and the entire global predicate becoming true. Thus we can either detect the predicate in the past (GP1 and GP2) or we must block the computation at critical points. How much do these restrictions confound the debugging process? The definitions GP1 and GP2 imply that the predicate of interest may no longer be true when detection is reported, e.g. when the program is halted in order to examine local state. This is so if the predicate is not stable [1]. Thus one must either record relevant program state and include it in the messages sent to the monitoring process, or rely on some program replay or reversible execution technique to recover the state of interest.

With definition GP3, we report the predicate when it is currently true in the global program state. However to achieve this we have to delay execution of some processes. This is can be a serious impediment to debugging. By blocking some processes when the predicate becomes potentially true, we may make the predicate either more or less likely to occur. For example, a predicate may be less likely to occur if processes "communicate" using timeouts or some other uncontrolled form of communication. The latter is a particular problem when processes are multithreaded; that is, consisting of multiple, independently schedulable threads of control which may communicate through shared memory. It is rarely practical to monitor such communication when debugging without hardware or language support.

2 Algorithms

2.1 Detecting *Possibly* Φ and *Definitely* Φ

The algorithms for detecting *possibly* Φ and *definitely* Φ are based on the same data structure: the lattice of consistent global states that correspond with an observed execution. Such a lattice consists of n orthogonal axes, with one axis for each monitored process. A point $\overline{x} = (x_1, x_2, \ldots, x_n)$ in this lattice corresponds to a global state in which process p_i has taken x_i steps. Of course, not all values of $(x_1, x_2, \ldots, x_n)$ denote a point in the lattice, depending on the causal dependencies among the local states of P. Define the *level* of a point $\overline{x}$ to be the sum of its indices $x_1 + x_2 + \ldots + x_n$.

Consider an observed behaviour of a system. A possible execution of this system is a total order of (consistent) global states in which exactly one process takes a step be-

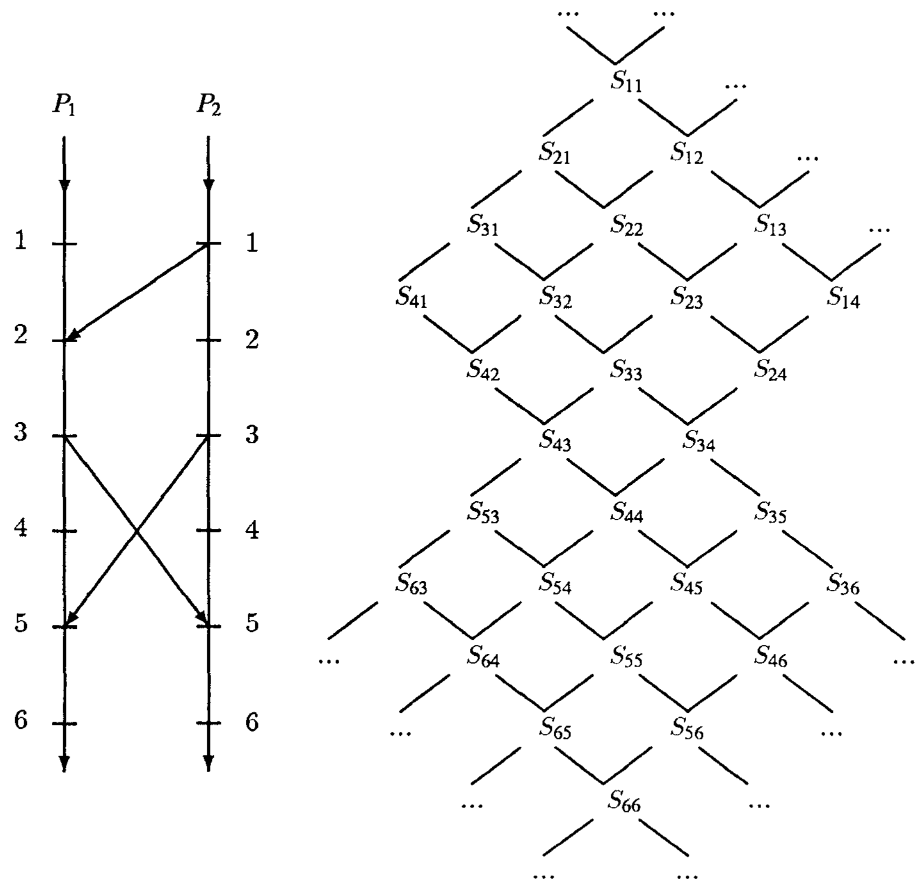

Figure 2: An execution and the corresponding lattice of global states.

tween adjacent global states. In terms of the lattice corresponding with the observed behavior, a possible execution is a path in this lattice where the level of each subsequent point in the path increases by one. A space-time diagram of a two-processes system and the corresponding lattice of global states is illustrated in Figure 2. A point $S_{i,j}$ represents a state in which process p_1 is in its i^{th} state and process p_2 is in its j^{th} state. From the lattice, it is easy to see that one possible execution is the sequence of global states

$$S_{1,1};\ S_{2,1};\ S_{2,2};\ S_{2,3};\ S_{3,3};\ S_{4,3}\ldots$$

For every point $\overline{x}$ in a lattice, there exists an execution that passes through $\overline{x}$. Hence, if any point in the lattice satisfies Φ, then by **GP1** *possibly* Φ holds. The property *definitely* Φ requires all possible executions to pass through a point that satisfies Φ. For example, in Figure 2 if the points $\{S_{4,3}, S_{3,4}\}$ both represent states that satisfy Φ then *definitely* Φ holds. This is because $\{S_{4,3}, S_{3,4}\}$ are the only points with a level of 7 and all executions must include exactly one point with any given level. *Definitely* Φ also holds if the states represented by points $\{S_{5,3}, S_{3,5}, S_{6,4}, S_{5,5}, S_{4,6}\}$ all satisfy Φ. This is because if the execution does not pass through $S_{5,3}$ or $S_{3,5}$, then it must pass through $S_{4,4}$ and hence through one of $\{S_{6,4}, S_{5,5}, S_{4,6}\}$.

Figures 3 and 4 give high-level algorithms used by the monitoring process p_0 to detect *possibly* Φ and *definitely* Φ respectively. The *definitely* Φ algorithm iteratively constructs the set of global states that have a level *lvl* and are reachable from the initial state without passing through a global state that satisfies Φ. If there are no such states, then *definitely* Φ holds by the level *lvl*.

In order to implement the algorithms in Figures 3 and 4, the monitored processes send their local states to the monitor p_0. p_0 maintains sequences of these states, one sequence per process, and assembles them into the necessary global states. The monitor must be able to determine when it can assemble all the reachable global states of a given level and when it can drop a local state from its sequence because the local state will not appear in any further global states of interest. To achieve this, we use vector time stamps.

Let Q_i be the sequence of states that p_0 maintains for p_i

```
Possibly($\Phi$): begin
    % Synchronize processes and distribute $\Phi$
    send $\Phi$ to all processes;
    current := global state $S(0,0,\ldots,0)$;
    release processes;
    lvl := 0;
    do no state in current satisfies $\Phi$ $\rightarrow$
        last := current
        lvl := lvl + 1;
        current := states of level lvl reachable from a state in last;
        od
    end;
    report Possibly $\Phi$
```

Figure 3: Algorithm for detecting *Possibly* Φ.

```
Definitely($\Phi$): begin
    % Synchronize processes and distribute $\Phi$
    send $\Phi$ to all processes;
    last := global state $S(0,0,\ldots,0)$;
    release processes;
    remove all states in last that satisfy $\Phi$;
    lvl := 1;
    % Invariant: last contains all states of level lvl − 1 that are accessable
    % from $S(0,0,\ldots,0)$ without passing through a state satisfying $\Phi$.
    do last $\neq$ { } $\rightarrow$
        current := states of level lvl reachable from a state in last
        remove all states in current that satisfy $\Phi$;
        lvl := lvl + 1; last := current
        od
    end;
    report Definitely $\Phi$
```

Figure 4: Algorithm for detecting *Definitely* Φ.

stored in FIFO order. Each state e_i in Q_i is labeled with a vector time stamp $V(e_i)$ where $V(e_i)[i]$ is the event number of p_i that resulted in this state and $V(e_i)[j], j \neq i$ is the number of the latest event of p_j that p_i is aware of. For example, in Figure 2, the vector timestamp of the fourth event of process p_1 is $(4, 1)$. A set of local events $\{e_1, e_2, \ldots, e_n\}$ with e_i from process p_i comprise a valid global state iff

$$\forall i, j : 1 \leq i, j \leq n : V(e_i)[i] \geq V(e_j)[i]$$

Intuitively, this condition states that a message cannot be received before it is sent. Note that the level of this global state is $\sum_{u=1}^{n} V(e_u)[u]$.

For every event e_i of process p_i there exists a *minimum* global state $S_{min}(e_i)$ that contains e_i and a *maximum* global state $S_{max}(e_i)$ that contains e_i. These global states are:

$$S_{min}(e_i) = (e_1, e_2, \ldots, e_n) : V(e_j)[j] = V(e_i)[j]$$

and

$$\begin{aligned} S_{max}(e_i) = (e_1, e_2, \ldots, e_n) : \\ V(e_j)[i] \leq V(e_i)[i] \wedge \\ \forall e'_j : e_j \rightarrow e'_j : V(e'_j)[i] > V(e_i)[i] \end{aligned}$$

These two global states limit the levels in which e_i occurs. The minimum level containing e_i is particularly easy to compute: it is the sum of components of the vector time stamp $V(e_i)$. So, p_0 can construct the set of states with level *lvl* when, for each sequence Q_i, the sum of the components of the vector time stamp of the last element of Q_i is at least *lvl*. And, p_0 can remove event e_i from Q_i when *lvl* is greater than the level of the global state $S_{max}(e_i)$.

The two detection algorithms are linear in the number of global states, but unfortunately the number of global states is $\Omega(k^n)$ where k is the maximum number events a monitored process has executed. There are techniques

```
Currently($\Phi$): begin
    $M$: array $[1..n]$ of state := all empty;
    $a$: array $[1..n]$ of integer := all 0;
    send $\Phi$ to all processes;
    do $M$ does not satisfy $\Phi \rightarrow$
        receive message $m$ from process $i$;
        if $m$ is a state message then $M[i] := m$
        else if $m$ is a block message then
            $a[i] := n - 1$;
            send ack($i$) to all processes except $i$
        else if $m$ is an ack($j$) message then
            $a[j] := a[j] - 1$;
            if $a[j] = 0$ then send unblock to process $j$; $M[i] :=$ empty;
        od
    end;
    report Currently $\Phi$
```

Figure 5: Monitor protocol for detecting *Currently* Φ.

```
Client: cobegin
    receive $\Phi$;
    for each event $e$ that will potentially change $\Phi$ do
        if $e$ potentially rejects $\Phi$ then
            send block to monitor;
            receive unblock;
            execute $e$ and send state to monitor
        else execute $e$ and send state to monitor
        od
    []
        do true $\rightarrow$
            receive ack($j$);
            send ack($j$) to monitor
        od
    coend
```

Figure 6: Client protocol for detecting *Currently* Φ.

that can be used to limit the number of constructed global states. For example, a process p_i need only send a message to p_0 when p_i potentially changes Φ or when p_i learns that p_j has potentially changed Φ. Another technique is for p_i to send an empty message to all other processes when p_i potentially changes Φ. These, and other techniques for limiting the number of global states are discussed in [5].

2.2 Detecting *Currently* Φ

In contrast to the previous two algorithms, detecting *Currently* Φ is computationally cheap but may block the monitored program, as discussed in Section 1. Figures 5 and 6 give a protocol that detects *Currently* Φ for conditions that do not reference the number of messages sent by a process[1].

This protocol is very simple; the main idea is to ensure that the condition holds long enough for the monitor to react. Whenever a process executes an event that potentially changes Φ, it sends its relevant state to p_0, which maintains the latest received state for each process in P. If a process is about to execute an event that could make Φ false, however, it first notifies p_0 and blocks before executing this event. The monitor then flushes all of the links into p_0, thereby examining all of the local process states of P up to the time that the first process blocked. If Φ is not found to hold by the time all of the channels are flushed then p_0 releases the blocked process.

An example of the execution of this protocol is shown in

[1] More precisely, a process's local state with respect to Φ does not change when that process sends a message.

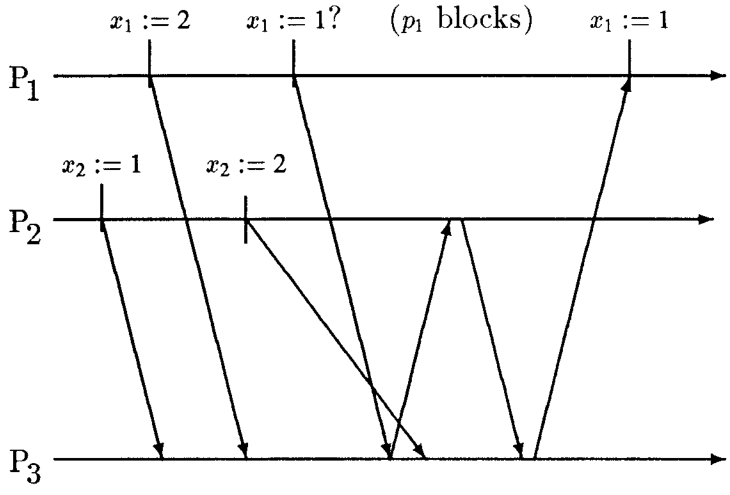

Figure 7: $\Phi = $ (*Currently* $x_1 + x_2 \geq 5$).

Figure 7 in which the condition $\Phi =$ *Currently* $x_1 + x_2 \geq 5$ is being monitored. Each process can increment its value of x without blocking since doing so can only potentially make Φ true. Process p_1, however, before decrementing its value of x since doing so potentially makes Φ false. The monitor therefore can consider the state $x_1 = 2, x_2 = 2$ before Φ could become false.

2.3 Comparison with Other Algorithms

The work most similar in spirit to ours are the protocols developed by Spezialetti [8]. In particular, her *event holding* condition is the same specification as our protocol for detecting *Currently* Φ, and the specification of her *event occurrence* condition is similar to the specification of our *Possibly* Φ algorithm. However, her protocols for non–local event detection are incomplete, in that they can miss conditions that in fact held. For example, Figure 8 shows such an execution. If the messages in this figure correspond to the messages generated in establishing simultaneous regions [9], then her protocol will not detect $x_1 = x_2$, yet in fact the condition *Definitely* $x_1 = x_2$ holds.

Our notion of *possibly* and *definitely* is closely related to the problem of access anomaly detection in parallel debugging and program analysis [10]. An important difference is that we wish to detect predicates over the states of the processes, while the access anomaly conditions are predicates over *sequences* of states (typically these are the sequences of instructions between synchronization events). Thus to apply our methods to access anomaly detection, we would have to augment process states with information on the *read* and *write* sets of global variables in a given instruction sequence. This is in fact how dynamic methods for detecting anomalies operate.

Recent algorithms for dynamically detecting access anomalies have also employed vector timestamps [2,3]. The Flowback Analysis technique [2], computes *before* and *after* vectors for each synchronization event in a program execution. These are directly related to our minimum and maximum global states, $S_{min}(e_i)$ and $S_{max}(e_i)$, for an event e_i.

The Flowback Analysis technique first gathers traces of event sequences and determines event orderings offline. Then various analyses are possible, of which access anomaly detection is just one. Our technique has two potential advantages: first, it does not require complete process traces to be recorded, and second, processes need not report all synchronization events, but only those that might change the predicate being detected. Of course the disadvantage is the the predicate must be specified before execution of the program.

3 Breakpoints and Tracepoints in Meta

The Meta system [6] is a toolkit that provides the basic primitives needed to build a non-real-time reactive system. It finds application in areas such as distributed application management, performance monitoring, load balancing, and distributed debugging.

A Meta computation consists of a *control program* and the *environment* which it monitors and controls (see Figure 9). In the case of debugging, the control program is the distributed debugger, and the environment is the distributed debuggee program. Using the Meta toolkit, the

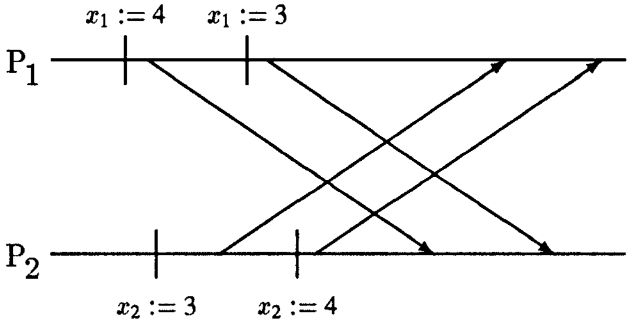

Figure 8: $\Phi = (x_1 = x_2)$.

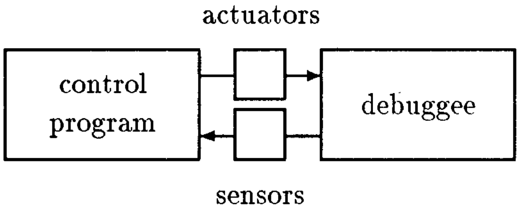

Figure 9: The Meta computation model

environment can be instrumented with sensors and actuators in order to expose its state. The control program monitors and controls the environment through the use of guarded commands (i.e. of the form **when** *condition* **do** *action*) that reference the sensors and actuators of the instrumented environment. These guarded commands are compiled into a low-level postfix language that is executed by interpreters that reside in *stubs* co-located with the components of the distributed environment (in our case the processes of the debuggee program). This architecture facilitates fast notification and reaction. Each condition is a proposition on the state of system, and the action portion is a sequence of actuator invocations that is executed atomically. References to sensors or actuators may be both local (within the entity to which the stub is attached) or nonlocal, allowing one to write distributed control programs. The toolkit includes other features, for describing the system structure, fault tolerance and concurrency control, that do not concern us here.

To make Meta useful for distributed debugging, we wish to instrument the debuggee program to provide the breakpoint and tracepoint facilities familiar to most programmers. In rule-based debuggers, breakpoints are often modeled as **when** *condition* **do** *stop program*, with the (often unstated) assumption that the state in which the program is stopped is (the first) one in which *condition* became true. As we have seen, when the condition is a global predicate, we must relax these requirements and detect the condition some time after it has happened, or perturb the computation being monitored. Also, since guarded commands are evaluated on the same computers as a the processes that they are monitoring, their impact on the performance of the debuggee is a concern.

However there is a practical problem with the Meta implementation even with strictly local predicates. If a local predicate and action such as

when *in_procedure(proc3)* **do** *stop*

is implemented as an interpreted guarded command, there will be a small but significant delay between detection of the condition and stopping the program. To avoid this we can model such a breakpoint as a modified kind of sensor that first halts the program, and then reports to the interpreter that the condition has become true and that the program is halted. In this view, Meta is monitoring a modified program that consists of the original program and the primitive breakpoints. We find utility in this concept for other applications of Meta. For instance when controlling a pressure vessel in an industrial automation project, one may prefer to use a pressure sensor that is directly connected to a safety shutoff valve. Such a sensor would report a potentially unsafe pressure reading *after* having shut off the valve, rather than relying on timely response from the Meta system to turn off the valve.

We are now incorporating these algorithms into the Meta implementation for use in distributed debugging and other application areas.

Acknowledgements We would like to acknowledge Gil Neiger, with whom we developed the protocols, and Raphael Yahalom who read an earlier draft of this paper.

References

[1] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] J.-D. Choi, B. P. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. Technical Report 786, University of Wisconsin-Madison, Computer Sciences Department, Aug. 1988.

[3] A. Dinning and E. Schonberg. The task recycling technique for detecting access anomalies on–the–fly. Technical Report RC 15385 (68453), IBM T.J.Watson Research Center, Jan. 1990.

[4] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] K. Marzullo and G. Neiger. Detection of global state predicates. In preparation.

[6] K. Marzullo and M. Wood. Tools for monitoring and controlling distributed applications. Technical Report TR 91-1187, Cornell University, Jan. 1991. To appear in *EurOpen 1991*.

[7] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers, Amsterdam, 1989.

[8] M. Spezialetti. *A Generalized Approach to Monitoring Distributed Computationsfor Event Occurences.* PhD thesis, University of Pittsburgh, 1989.

[9] M. Spezialetti and J. P. Kearns. Simultanous regions: A framework for the consistent monitoring of distributed computations. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 61–68. IEEE, 1989.

[10] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(5):265–278, May 1980.