Detection of Global State Predicates

Keith Marzullo*
Gil Neiger**

TR 91-1221
November 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This author was supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM, Siemens and Xerox. 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. Author's address: Cornell University Department of Computer Science, Upson Hall, Ithaca, NY 14853-7501, USA.
**This author was supported in part by the National Science Foundation under grants CCR-8909663 aand CCR-9106627. Author's address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 30332-0280, USA.

Detection of Global State Predicates

Keith Marzullo* Gil Neiger†

Abstract

This paper examines algorithms for detecting when a property Φ holds during the execution of a distributed system. The properties we consider are expressed over the state of the system and are not assumed to have properties that facilitate detection, such as stability.

Detection is done by a monitoring process within the system, which cannot perceive an execution of a distributed system as a total order; because of this, we consider two interpretations for "detecting Φ":

1. There is an execution consistent with the observed behavior such that Φ was true at a point in that execution. We refer to this property as *possibly* Φ.

2. For all executions consistent with the observed behavior, there was some point in real time at which the global state of the system satisfied Φ. We refer to this property as *definitely* Φ.

In this paper, we give formal definitions for these two interpretations and present algorithms for them. We give protocols for both asynchronous and synchronous systems and, for synchronous systems, give upper bounds on the time between the occurrence of the property of interest and the time a monitor detects the property.

*This author was supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM, Siemens and Xerox. 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. Author's address: Cornell University Department of Computer Science, Upson Hall, Ithaca, New York 14853-7501, USA.

†This author was supported in part by the National Science Foundation under grants CCR-8909663 and CCR-9106627. Author's address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280, USA.

1 Introduction

A *reactive system* [6] is characterized by a *control program* that interacts with an *environment*. The control program is input-driven: it monitors the environment and reacts to significant events by sending commands to the environment. There are many examples of reactive systems; for example, most embedded real-time systems are reactive systems, in which case the environment is an instrumented physical process. Non-real-time examples of reactive systems includes monitoring and debugging systems [4,13] and tool integration services [5,14].

In the *Meta* project [9,11], we have been developing tools that support the management of distributed applications through the use of a reactive system structure. Using Meta, the distributed application and its supporting services (for example, operating system, network servers, and hardware) can be instrumented with *sensors* that access its state and *actuators* that allow its state to be changed. Meta also provides a distributed interpreter of finite state automata that reference these sensors and actuators. Under Meta, control programs are translated into finite state automata that are executed by this distributed interpreter. Each interpreter executes *guarded atomic commands* of the form $\langle \Phi \rightarrow S \rangle$, meaning execute the action S in a state satisfying the global state predicate Φ.

The problem addressed in this paper arises in the context of Meta: how can a set of processes monitor the state of a distributed application in a consistent manner? For example, consider the simple distributed application shown in Figure 1. Each of the three processes in the application has a light, and the control processes would each like to take an action when some specified subset of the lights are on. The application processes are instrumented with *stubs* that determine when the process turns its light on or off. This information is disseminated to the control processes, each of which then determines when its condition of interest is met.

Meta is built on top of the ISIS toolkit [1], and so we first built the sensor dissemination mechanism using atomic broadcast. Atomic broadcast guarantees that all recipients receive the messages in the same order and that this order is consistent with causality [7]. Unfortunately, the control processes are somewhat limited in what they can deduce when they find that their condition of interest holds.

For example, Figure 2 shows a space-time diagram of an execution of the application shown in Figure 1. In this figure, a process turning its light on is represented by a rectangle and the process turning its light off is rep-

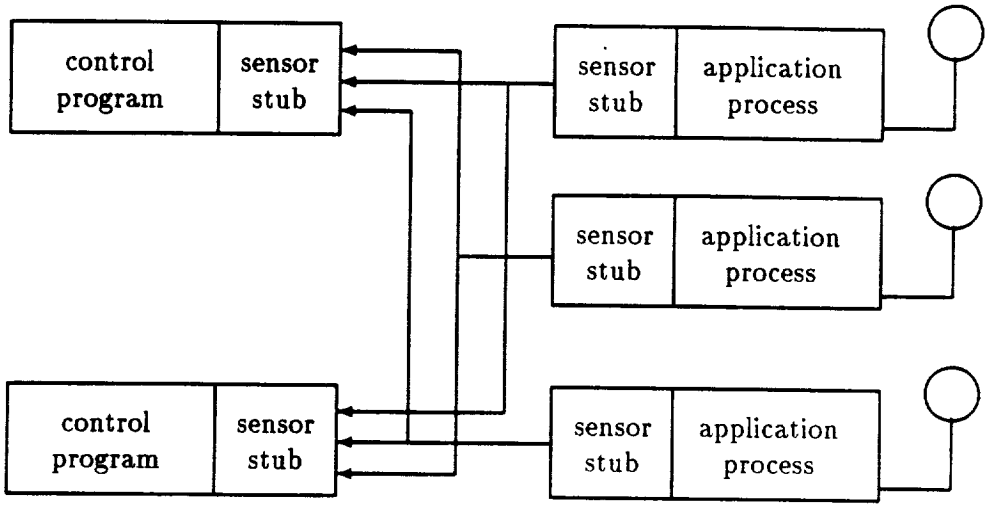

Figure 1: A Monitored Distributed Application

resented by a vertical line. Assume, for the moment, that this system is *asynchronous*, meaning that there is no bound on message passing delays or on the relative speeds of processes. In this case, the only ordering relations between events that can be determined from within the system are those of potential causality. Two events that are not so related are *concurrent*. In Figure 2, the events a and b are concurrent as are a and c, so the control processes could receive these event notifications (as sent by atomic broadcast) in one of these orders: $(a; b; c)$, $(b; a; c)$ or $(b; c; a)$. Thus, the control processes may or may not determine that p_1's and p_2's lights were on simultaneously, but they will reach the same decision. On the other hand, the events a, d and e are causally ordered, so the control processes will determine that p_1's and p_3's lights were on simultaneously.

Given a global property Φ, there are at least two ways that "detecting Φ" can be interpreted:

1. There is an execution (*i.e.* a linear sequence of events) consistent with the observed behavior such that Φ was true at a point in that execution. We will refer to this property as *possibly* Φ. In the space-time diagram shown in Figure 2, the predicate *possibly* (p_1's light on and p_2's light on) holds.

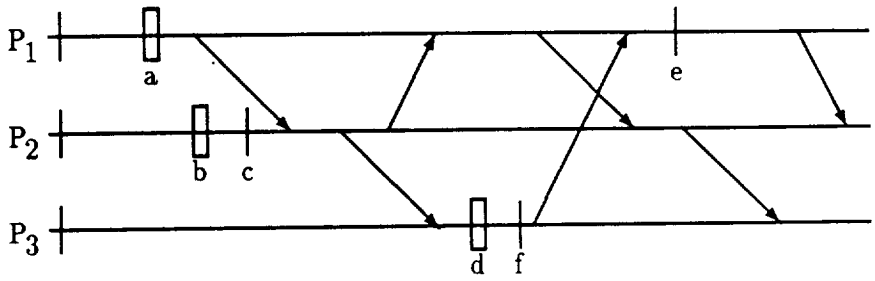

Figure 2: Space-Time Diagram of Application Execution

2. For all executions consistent with the observed behavior, there was some point in real time at which the global state of the system satisfied Φ. We will refer to this property as *definitely* Φ. In the execution shown in Figure 2, the predicate *definitely* (p_1's light on and p_3's light on) holds, since the event of p_3 turning its light on happened between p_1 turning its light on and p_1 turning its light off.

Note that *definitely* Φ is stronger than *possibly* Φ. Hence, we will want to guarantee that if a control program determines *possibly* Φ for a set of local states, then no control program will ever determine *definitely* $\neg\Phi$ for the same states. Note that both of these conditions refer to some past state or states.

In this paper, we give formal definitions for these two interpretations and present protocols for them. We first give the protocols for an asynchronous system. These protocols can take an unbounded amount of time to detect their condition of interest; furthermore, they can have substantial running times because they may have to enumerate may possible global states. However, no better is possible, in general, due to the nature of asynchronous systems. We then modify these protocols for a system with approximately synchronized clocks and bounded message delay. These protocols are more practical, and we give upper bounds on the time between the occurrence of the property of interest and the time a control program detects the property. The existence of such a bound makes these protocols more useful in real systems.

Snapshot protocols for computing global states of a distributed system [2] are related to the protocols described in this paper, but they suffer from a limitation similar to that of the atomic broadcast implementation described

above. In particular, if S is the global state computed by the snapshot, then there exists a legal execution of the system containing S, and so $\Phi(S)$ implies *possibly* Φ. Snapshot protocols are well-suited for detecting *stable* properties, which are those that, once they become true, the remain so. It may be the case that *possibly* Φ holds of an execution, but the snapshot protocol never detects it (this can happen if Φ is not stable).

A recent dissertation by Spezialetti [16] looks at a broader set of issues than those covered in this paper, such as using semantic information (like relative stability) to determine which local events could make a global property true. This dissertation also presents protocols whose specification is similar to ours. However, her protocols that detects event occurrence suffer from the same limitation as snapshots and the atomic-broadcast-based protocol described above. Additionally, Spezialetti's protocols do not take into account the ordering of events established by the underlying system's communication. We have also looked at the problem addressed in this paper when environments are continuous state transition systems [8]. Such systems have the useful property that physical variables can, in many cases, be interpolated forward. By doing so, the monitor can reason about the current state of the physical process rather than a past state, and so *possibly* Φ and *definitely* Φ can be determined for the current state.

2 Definitions

We first define the notion of an execution of a system. A system is composed of *processes*, some of which are part of the application being run and some of which are part of the monitoring control program. Let $\{p_1, \ldots, p_n\}$ be the set of application processes; for the sake of simplicity, we assume that there is only one monitoring process, denoted p_0. Each pair of processes is connected by a point-to-point, reliable, FIFO communication link, and we assume that processes do not exhibit faulty behavior.

Each process p_i has a *local state* s_i, which changes when an *event* occurs at the process. An event may be completely internal to the process, or it may be the sending or receipt of a message (e.g., "send m_1 to p_j" or "receive m_2 from p_k"). For the sake of simplicity, we assume that all message sent in the system are unique. Process p_i's *local history*, denoted h_i is a (possibly infinite) sequence of states and events

$$s_i^0 e_i^1 s_i^1 e_i^2 s_i^2 \cdots.$$

In this case, s_i^0 is p_i's initial state, and the first event it executes is e_i^1.

after which the process's state is s_i^1, etc. A *global state* is a tuple $S = \langle s_0, s_1, \ldots, s_n \rangle$, one for each process. Although the monitor, p_0, is a process in the system, when we refer to a global state, we will usually mean only the global state of the application, $\langle s_1, \ldots, s_n \rangle$. A *global history* (or *history*) is a tuple $H = \langle h_0, h_1, \ldots, h_n \rangle$ of local histories, one per process.

Although a global history does not specify the relative timings of events and states at different processes, it does allow us to draw certain conclusions about these timings. An event e_i^l *happens before* e_j^m (written $e_i^l \rightarrow e_j^m$) if one of the following is true [7]:

- the events are at the same process and occur in the order indicated, that is, if $i = j$ and $l < m$;
- e_i^l is the sending of a message by p_i to p_j and e_j^m is the receipt of that message; or
- there is another event e_k^n such that $e_i^l \rightarrow e_k^n$ and $e_k^n \rightarrow e_j^m$.

The "happens before" relation can be used to reason about the possible executions associated with a global history. We associate with each global history a set of linearizations.

A *linearization* L of a history H is a sequence of global states and local events

$$S^0 e^1 S^1 e^2 S^2 \ldots$$

that contains exactly the events in H such that, if $e^m \rightarrow e^n$ in H, then $m < n$. Notice that no prefix of L contains the receipt of a message whose sending does not appear in that prefix. In synchronous systems (see Section 3.3), there are further constraints on the linearizations of a global history.

(The above definition of linearization assumes that, in the actual execution of a distributed system, no two events can occur simultaneously. This need not be the case; it is possible that events at different processors may occur at exactly the same time. We can easily extend our definition of linearization to include such definitions.)

We use the notion of a cut to represent the global states that could have occurred in the execution. A *cut* [2] of a global history H is a tuple of natural numbers $\langle t_1, \ldots, t_n \rangle$ that represents the state of the system after t_i events have executed at process p_i; that is, the cut represents the global state $\langle s_1^{t_1}, \ldots, s_n^{t_n} \rangle$. Only certain cuts of a global history can truly correspond to global states that took place at some real time. A cut $\langle t_1, \ldots, t_n \rangle$ of H is *consistent* if there is some point in some linearization L of H by which each

process p_i executed exactly t_i events. L is said to *pass through* this cut. We will also refer to the associated tuple of events, $\left\langle e_1^{t_1}, \ldots, e_n^{t_n} \right\rangle$ as a consistent cut.

We want to be able to reason about certain facts (such as "possibly Φ") being true in different global histories. To this end, we introduce the following notation. Let H be some global history of the system. To formally define "*possibly* Φ," we introduce the formulas $?\Phi|C$, where C is a consistent cut. Formally, $?\Phi|C$ holds for history H if $C = \langle t_1, \ldots, t_n \rangle$ is a consistent cut of H and Φ holds for the global state $\langle s_1^{t_1}, \ldots, s_n^{t_n} \rangle$. If $?\Phi|C$ holds for H, then it is possible that Φ held during the execution that generated H since it held at some point in some linearization of H.

To formally define "*definitely* Φ," we introduce the formulas $!\Phi|A$, where A is a finite set of cuts. Formally, $!\Phi|A$ holds for H if A is a finite set of consistent cuts of H, every linearization of H passes through some cut in A, and for all cuts $\langle t_1, \ldots, t_n \rangle \in A$, Φ holds for the global state $\langle s_1^{t_1}, \ldots, s_n^{t_n} \rangle$. If $!\Phi|A$ holds for H, then Φ definitely held at some point in the execution that generated H because it held at some point in all linearizations of H.

Note that the definitions of these formulas satisfy two properties discussed earlier. The definitely operator ! is clearly stronger than the possibly operator ? in the following sense: if $!\Phi|A$ holds for H then for any $C \in A$, $?\Phi|C$ holds for H. Furthermore the two operators are, in a certain sense, dual. If $!\neg\Phi|A$ holds for H, then $?\Phi|C$ cannot hold for H if $C \in A$.

Informally, the control process p_0 detects *possibly* Φ when it can determine that there is a consistent cut of H that satisfies Φ, and p_0 detects *definitely* Φ when it can find a finite set of consistent cuts A such that every linearization of H passes through a member of A and such that Φ holds for every member of A. We are investigating the more formal definition of detection, but we do not present such defintions in this version of the paper.

3 Protocols

As noted above, system consists of $n+1$ processes $\{p_0, p_1, \ldots, p_n\}$ whose only method of interaction is by exchanging messages. The process p_0 monitors the other processes to determine when some state predicate becomes true. This state predicate of interest will be of the form *possibly* Φ or *definitely* Φ, where Φ is a predicate over the states of the processes $p_1, \ldots, p_n$.

Each process p_i will know how to compute Φ and will send a message to p_0 when its local state changes in a way significant with respect to Φ. In particular, a process can determine whether a local event potentially changes

Φ. More formally, let Φ be a predicate expressed over a global state; that is, $\Phi(s_1, \ldots, s_n)$ is true or false. Consider some event e_i^t of process p_i; recall that s_i^{t-1} is the value of s_i before e_i^t executes and s_i^t is the value of s_i after e_i^t executes. Event e_i^t *potentially affirms* Φ if the execution of e_i^t could have made Φ true:

$$\exists s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n : \neg\Phi(s_1, \ldots, s_i^{t-1}, \ldots, s_n) \wedge \Phi(s_1, \ldots, s_i^t, \ldots, s_n).$$

Similarly, event e^i *potentially rejects* Φ if the execution of e_i^t could have made Φ false:

$$\exists s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n : \Phi(s_1, \ldots, s_i^{t-1}, \ldots, s_n) \wedge \neg\Phi(s_1, \ldots, s_i^t, \ldots, s_n).$$

An event *potentially changes* Φ if it potentially affirms or rejects Φ: such an event is also called a *relevant event.*

Note that an event can both potentially affirm and reject Φ. For example, if $n \geq 4$ and Φ is "either two or three processes have their lights on," then when a process turns its light on, this action both potentially affirms and rejects Φ even though it is possible that the value of Φ did not change.

Our detection protocols will have the monitored processes periodically send to the monitor its state relevant to Φ; that is, the message will contain the values of the variables of p_i referred to in Φ. For each process p_i ($1 \leq i \leq n$), process p_0 maintains a sequence Q_i of such messages received from p_i. These messages will also carry information for ordering these states, which is described next.

3.1 Weak Vector Clocks and Enumeration of Global States

Our protocols will have the monitor enumerate possible global states of the system by choosing states from each of the message sequences Q_i. In this section, we describe how this enumeration of global states is performed. We use a slight modification of *vector clocks* [12].

A *logical clock* [7] is a value T that satisfies the *clock condition*: given two events e_1 and e_2 and their associated clock values $T(e_1)$ and $T(e_2)$, if $e_1 \rightarrow e_2$, then $T(e_1) < T(e_2)$. We will find it advantageous to use clocks that also satisfy the converse of the clock condition; that is, clocks that satisfy

$$(e_1 \rightarrow e_2) \Leftrightarrow T(e_1) < T(e_2). \tag{1}$$

In particular, such clocks enable one to determine whether or not two events are concurrent; e_1 and e_2 are *concurrent* if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

Unfortunately, Lamport's logical clocks of [7] (which are implemented using a single counter) do not satisfy Equation 1.

A logical clock that satisfies Equation 1 can be implemented with a vector V of n counters. If V_i is the logical clock associated with process p_i, then $V_i[i]$ is the number of events that have been executed by p_i and $V_i[j]$, $j \neq i$, is the number of events that p_i "knows" p_j has executed. If e_i is an event at process i, then we use $V(e_i)$ to denote the value of V_i *after* e_i executed. Given this definition, one can easily show that $e_i \rightarrow e_j$ if and only if the vector clock of e_j records the fact that e_i has occurred:

$$e_i \rightarrow e_j \Leftrightarrow V(e_i)[i] \leq V(e_j)[i]. \tag{2}$$

Similarly, if e_i and e_j are concurrent, then

$$V(e_i)[i] > V(e_j)[i] \wedge V(e_j)[j] > V(e_i)[j].$$

If the set of processes is static, then vector clocks are not hard to implement. Initially, $V_i[j]$ is set to zero for all i and j. $V_i[i]$ is incremented whenever p_i executes an event. Every message sent by p_i is timestamped with V_i (let $V(m)$ refer to the timestamp on message m). If e_i is the receipt of message m, then each $V_i[k]$ ($k \neq i$) is set to the maximum of $V_i[k]$ and $V(m)[k]$. As an example, Figure 3 shows the values of vector clocks for the events of the execution shown in Figure 2.

We can use vector clocks to determine whether or not a set of local states represents a consistent cut. The set of local states $S = \langle s_1, \ldots, s_n \rangle$ is a (consistent) global state if every pair of local states s_i and s_j is *potentially concurrent*. In terms of vector clocks, s_i and s_j are potentially concurrent if $V(s_i)[i] \geq V(s_j)[i]$ and $V(s_j)[j] \geq V(s_i)[j]$. Thus, the global state S is a consistent cut if and only if

$$\forall i, j : 1 \leq i, j \leq n : V(s_i)[i] \geq V(s_j)[i]. \tag{3}$$

Because we are interested only in the causal relationship of events that potentially change Φ, we can use a slight weakening of vector clocks [10]. With our clocks, process p_i will increment its local counter $V_i[i]$ only when it executes an event that potentially changes Φ. It will send a message to p_0 whenever its vector clock changes—that is, either when it executes a relevant event or when it executes a receive event through which it learns that another process has potentially changed Φ. The message sent from p_i to p_0 will contain p_i's state s_i after such an event is executed and the vector time $V(s_i)$.

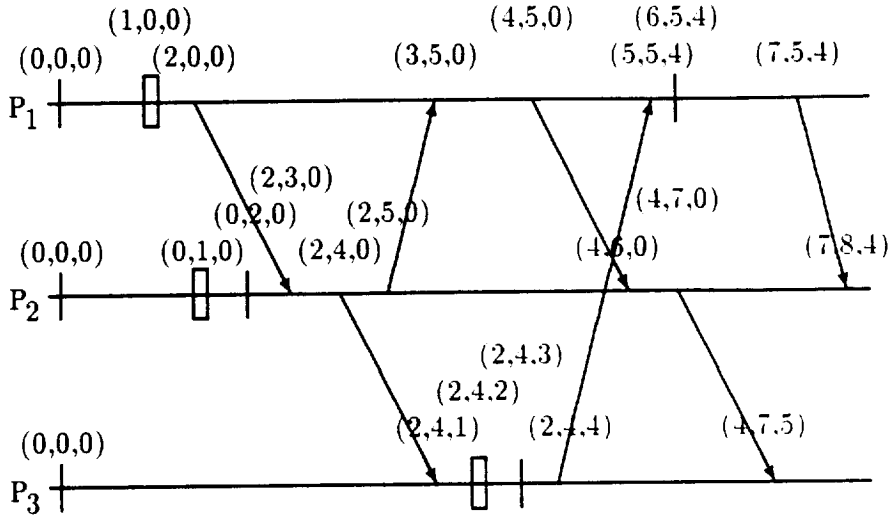

Figure 3: Vector Clocks for Events of Figure 2

Figure 4 illustrates such vector clocks. These clocks are a weakened version of normal vector clocks—for example, if $i = j$, they need not satisfy Equation 2. They do, however, satisfy Equation 3, and this is all that our protocols need. For the sake of simplicity, the remainder of this paper assumes that all events—including send and receive events— are relevant events and thus that our weak vector clocks are true vector clocks.

3.2 Asynchronous Systems

In this section, we assume that processes do not possess local real-time clocks, that there is no global clock, and that there is no upper bound on message delays. We note in advance that there is no way to bound the amount of time between the time a condition becomes true and the time the monitor detects the condition. This is because messages sent to the monitor may be arbitrarily delayed.

The protocols for detecting *possibly* Φ and *definitely* Φ are based on the same data structure: the lattice of consistent global states that correspond to an observed execution. Such a lattice consists of n orthogonal axes, with one axis for each monitored process. A point $\bar{t} = \langle t_1, t_2, \ldots, t_n \rangle$ in this lattice corresponds to a consistent global state in which process p_i has executed t_i

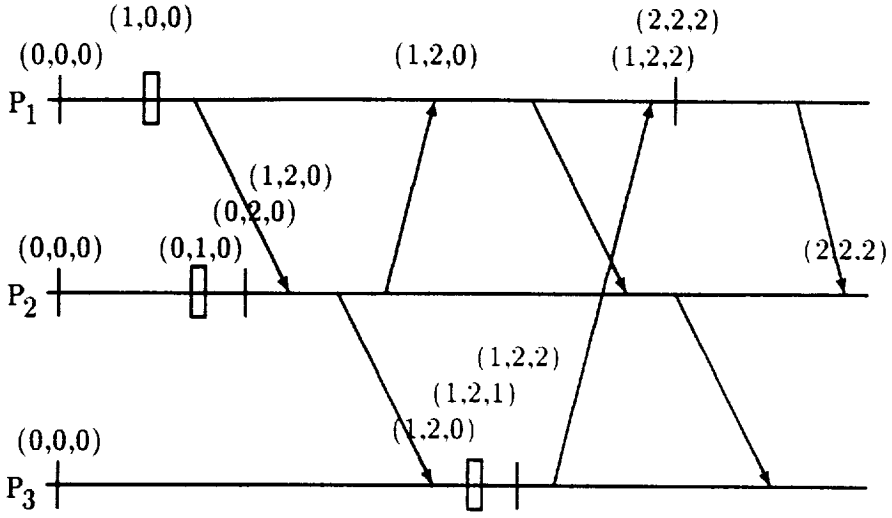

Figure 4: Weak Vector Clocks for Events of Figure 2

events. Of course, not all tuples $\langle t_1, t_2, \ldots, t_n \rangle$ appear in the lattice: this depends on the causal dependencies among the local states of P. Define the *level* of a point $\bar{t}$ to be the sum of its indices $t_1 + t_2 + \ldots + t_n$.

Consider some global history. A linearization of this history is a total order of (consistent) global states in which exactly one process executes one event between adjacent global states In terms of the lattice corresponding to the history, a linearization corresponds to a path in the lattice, where the level of each subsequent point in the path increases by one. A space-time diagram of a two-process system and the corresponding lattice of global states is illustrated in Figure 5. A point S_{ij} 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 corresponds to the sequence of global states

$$S_{00}; S_{01}; S_{11}; S_{21}; S_{22}; S_{23}; S_{33}; S_{43} \cdots.$$

For every point $\bar{t}$ in a lattice, there exists at least one linearization that passes through $\bar{t}$. Hence, if any point in the lattice satisfies Φ, then *possibly* Φ holds. The property *definitely* Φ requires all linearizations to pass through a point that satisfies Φ. For example, suppose in Figure 5 that the points S_{43} and S_{34} both represent states that satisfy Φ; then *definitely* Φ holds. This is

because S_{43} and S_{34} are the only points in level 7 and all linearizations must pass through some point in that level. *Definitely* Φ also holds if instead the states represented by points in the set $\{S_{53}, S_{35}, S_{54}, S_{45}\}$ all satisfy Φ. This is because if a linearization does not pass through S_{53} or S_{35}, then it must pass through S_{44} and hence through either S_{54} or S_{45}.

Figures 6 and 7 give the high-level algorithms that a monitoring process uses to detect *possibly* Φ and *definitely* Φ, respectively, in an n-processor system. Each algorithm begins by having the monitor distribute the predicate Φ to all processes and then construct the initial global state of level 0. (It is assumed that the monitor knows *a priori* each process's initial state relevant to Φ; if this is not the case, the processes begin by performing a two-phase synchronization protocol.)

The *possibly* Φ algorithm is straightforward: using the messages it receives, the monitor iteratively constructs levels of the lattice, using the vector timestamps accompanying each message (see below). If it ever finds a global state in the current level satisfying Φ, then it reports *possibly* Φ and halts. Note that this protocol is not optimal in its reporting time because it always waits for a level to be completely enumerated. This restriction is not necessary and is only done to simplify the presentation of the algorithm and the one that follows.

The *definitely* Φ algorithm also iteratively constructs one level at a time. It attempts to prove that all paths in the lattice pass through a state supporting Φ. To this end, when constructing a new level, it adds only states that do not support Φ; call the resulting level a *reduced level*. If the monitor ever finds an empty reduced level, then the monitor halts and reports *definitely* Φ (in fact, it can report that Φ definitely holds by the time processes execute a total of *lvl* relevant events, where *lvl* is the last level enumerated).

As stated earlier, the implementations of the algorithms in Figures 6 and 7 require a monitored process to send the relevant part of its local state to the monitor whenever its vector clock changes. The monitor maintains sequences of these states, one per process, and assembles them into the necessary global states. Thus, 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 cannot appear in any further global states of interest. To achieve this, we use weak vector timestamps developed in the previous section.

Let Q_i be the sequence of messages that p_0 has received from p_i stored in FIFO order. Each state s_i in a message stored in Q_i is labeled with the weak vector timestamp $V(s_i)$ of the event that generated that state. Equation 3

defines when a set of states $\langle s_1, s_2, \ldots, s_n \rangle$, with s_i from process p_i, comprise a consistent cut. Note that the level of this global state is $\sum_{u=1}^{n} V(s_u)[u]$.

Consider some point $\bar{t} = \langle t_1, \ldots, t_n \rangle$ in the lattice that corresponds to the state $\langle s_1, \ldots, s_n \rangle$. The monitor can enumerate points of the next level in the lattice as follows. For each process p_i, the monitor checks to see if s_i', the state in the $(t_i + 1)^{st}$ message of Q_i, is potentially concurrent with the other s_j's (if there is no such state in Q_i, the monitor cannot complete the next level until it receives that state). Thus, if

$$\forall j : j \neq i : V(s_i')[i] \geq V(s_j)[i] \wedge V(s_j)[j] \geq V(s_i')[j] \qquad (4)$$

then point $\langle t_1, \ldots, t_i + 1, \ldots, t_n \rangle$ is in the lattice at the next level. (Although many such states will have be checked, it should be clear that a state at some level in the lattice may follow from several in the previous level; it only has to be checked once and not for each possible predecessor.)

We can also use vector timestamps to determine when a message containing state s_i can be eliminated, in the interest of saving space, from a queue Q_i. If the last state in each other queue happens after s_i and is not potentially concurrent with it, then no state subsequently received could possibly form a global state with s_i. Thus, the message containing s_i can be removed from Q_i as soon as the following holds:

$$\forall j : j \neq i : V(Q_j.last)[i] > V(s_i)[i],$$

where $Q_j.last$ is the last state in Q_j.

The running time of both detection algorithms are linear in the number of global states. Unfortunately, the number of global states can be exponential in the number of processes. Even worse, the worst-case space complexity is unbounded, since the delivery of a message can be indefinitely delayed in an asynchronous system. While there are heuristics that can be used to limit the number of constructed global states, they are intrusive in that they require some kind of synchronization or limited blocking of the monitored processes. Real-time bounds on communication and the rate of change of local states can also be used, as is discussed in the next section.

3.3 Partially Synchronous Systems

In this section, we assume that each process p_i has a real-time clock C_i, and that these clocks are approximately synchronized: at any given "real" time, the difference between the clocks of two processes is no more than ϵ. We

define this formally by modifying our definition of histories and linearizations slightly. Firstly, all processes (including p_0) execute "tick" events; a process's local time is the number of tick events that it has executed. If e_i is an event at p_i, then $C_i(e_i)$ is the number of tick events that p_i has executed through e_i. If H is a history with approximately synchronized clocks, then L is a linearization of H only if, in addition to the usual requirement, in all prefixes of L and every pair of processes p_i and p_j, the difference in the number of tick events executed by the two processes is at most ϵ.

In addition to approximately synchronized clocks, we assume that there are lower and upper bounds on message transmission times. This means that if process p_i executes "send m to p_j" after it has executed t_s tick events, then when p_j executes "receive m from p_i," it has executed t_r tick events, where $t_s + d_{\mathbf{min}} \leq t_r \leq t_s + d_{\mathbf{max}}$ for constants $d_{\mathbf{min}}$ and $d_{\mathbf{max}}$ (both greater than 0). These bounds will be especially important when considering messages received by the monitor p_0. Approximately synchronized clocks can be used to extend the "happens before" relation to order two events e_i and e_j even when there is no explicit communication between p_i and p_j; thus, we redefine $e_i \rightarrow e_j$:

$$e_i \rightarrow e_j \Leftrightarrow (V(e_i)[i] \leq V(e_j)[i]) \vee (C(e_i) + \epsilon < C(e_j)).^1$$

That is, e_i must happen before e_j either if e_i can causally affect e_j (as measured by vector timestamps) or if the clock times corresponding to the events show that e_i must happen first.

Our protocols will be such that each state s_i sent by a process p_i to the monitoring process p_0 will include the local time $C(s_i)$ at which the event resulting in s_i occurred, as well as the vector timestamp $V(s_i)$. The monitor can then use the vector timestamps and the clock times of these states to enumerate the levels of the lattice. The clock times can be used to further restrict the pairs of events that are potentially concurrent. With each state s_i in Q_i, the monitor can determine the latest local time at which p_i must have been in state s_i (call this $L(s_i)$). If there is a state s_i' after s_i in Q_i, then this is $C(s_i')$; if $s_i = Q_i.last$, then this is $C - d_{\mathbf{max}}$, where C is the monitor's current local time. If p_i had changed its local state between $C(s_i)$ and $C - d_{\mathbf{max}}$, then the monitor would have gotten another message from p_i by its local time C.

[1] There is no need to take the transitive closure of the two relations because, if $d_{\mathbf{min}} \geq 0$, $V(e_i)[i] \leq V(e_j)[i]$ and $C(e_j) + \epsilon < C(e_k)$ then $C(e_i) + \epsilon < C(e_k)$, and if $C(e_i) + \epsilon < C(e_j)$ and $V(e_j)[j] \leq V(e_k)[j]$ then $C(e_i) + \epsilon < C(e_k)$.

We can now say that two states s_i and s_j received by the monitor are potentially concurrent if both the vector time stamps and the the real-time clocks indicate this:

$$\begin{aligned}(V(s_i)[i] \geq V(s_j)[j]) \quad &\wedge \quad (V(s_j)[j] \geq V(s_i)[i]) \\ &\wedge \quad ((C(s_i) - \epsilon) \leq C(s_j) \leq \ L(s_i) + \epsilon \\ &\quad\ \vee (C(s_j) - \epsilon) \leq C(s_i) \leq \ L(s_j) + \epsilon).\end{aligned} \tag{5}$$

Suppose now that the monitor is seeking to extend the state $\langle s_1, \ldots, s_n \rangle$ to the next level by potentially adding a new state s'_i, the $(t_i + 1)^{st}$ state in Q_i. It checks to see if s'_i is potentially concurrent with the other s_j's by using Equation 5 instead of Equation 4. If s'_i is potentially concurrent with all the s_j's, then the state $\langle s_1, \ldots, s'_i, \ldots, s_n \rangle$ is added to the next level of the lattice; otherwise, it is not. An exception to the last point is if $s'_i = Q_i.last$ and s'_i was not deemed to be potentially concurrent because its latest time was too early. For example, suppose $\epsilon = 1$ and

$$C(s'_i) = 3; \qquad L(s'_i) = 4; \qquad C(s_j) = 6; \qquad L(s_j) = 7.$$

Because $s'_i = Q_i.last$, $L(s'_i) = C - d_{\max}$; as time passes on the monitor's clock, $L(s'_i)$ may grow so that the two states would be judged potentially concurrent. In such cases, therefore, the decision about whether to add the state $\langle s_1, \ldots, s'_i, \ldots, s_n \rangle$ to the lattice is postponed until either another message arrives from p_i or the monitor's clock advances to a point where a decision can be made. Until then, the level cannot be completely enumerated.

The conditions *possibly* Φ and *definitely* Φ can now be detected exactly as in the previous section. Each processor sends its state to the monitor whenever its vector clock changes; it includes with this message its vector time and the number of tick events it has executed. The monitor then uses this information to construct levels of the lattice, using the properties of the "potentially concurrent" states discussed above. It then reports "*possibly* Φ" or "*definitely* Φ" exactly as it would in the case of asynchronous systems.

We now argue upper bounds on detection times. Suppose that $S = \langle s_1, \ldots, s_n \rangle$ is a global state such that the last event leading to this state occurs when the monitor's local time is t. No process's local clock is higher than $t + \epsilon$ when one of the events leading to S occurs, so p_0 receives all messages necessary to construct this state by local time $t + \epsilon + d_{\max}$. Local time $t + 2\epsilon$ is the latest that a process could execute an event that could be

potentially concurrent with one leading to S; thus, by time $t + 2\epsilon + d_{\max}$, p_0 will have completed the construction of the level containing S.

Suppose that *possibly* Φ holds of a history; this means that some consistent cut of the history supports Φ. If the last event leading to this cut occur when the monitor's local clock is t, then the monitor will finish enumerating the level of S at its local time $t + 2\epsilon + d_{\max}$, detecting *possibly* Φ at that time (actually, it could detect it at time $t + \epsilon + d_{\max}$ because, as noted earlier, the *possibly* protocol does not need to enumerate an entire level once it finds a state satisfying Φ).

Suppose that *definitely* Φ holds of a history; this means that there is some finite set of consistent cuts, all supporting Φ, through which all linearizations pass. If the last event leading to the last of these occurs when the monitor's local time is t, then the monitor must detect *definitely* Φ by time $t + 2\epsilon + d_{\max}$ on its local clock; this is because the last state in the level of the last cut will be enumerated by that time, and the protocol will halt.

The above discussion does not consider the amount of local computation required by the monitor. In general, this depends on the relation between ϵ and the rate at which processes can potentially change Φ. If clocks are closely synchronized, then the monitor will never have to consider more than a few state changes by any one process. If instead processes potentially change Φ very often, then the monitor may have to do significant local computation.

4 Conclusions

This paper has defined two means (*possibly* and *definitely*) by which global states in an asynchronous or loosely synchronous system can be detected. It presented algorithms by which a monitor can detect these properties in both types of systems. There are other means of detection that are also of interest. For example, we have been investigating a third type of detection, called *currently*, that occurs when the monitor learns a condition actually holds at the time of detection. One can modify our *definitely* algorithms for partially synchronous systems to detect *currently* by requiring that application processes forgo potentially rejecting the condition being detected for a well-defined amount of time. We can obtain *currently* algorithms for asynchronous systems only by forcing application processes to block.

These algorithms may be complex, both in terms of computation and storage. Although we are investigating optimizations of these algorithms, we maintain that significant complexity is required for detection to be complete. In the future, we plan to look at the kinds of information that may

simplify the detection. If the property that is to be detected, Φ, has certain nice properties, then detection may be simplified. If the monitor has some knowledge of the how and when the application program potentially changes the condition to be detected, then this can also simplify detection. We have also been investigating casting the detection problem into temporal and epistomological logics. We believe that such a characterization will aid in finding sets of properties under which detection can be simplified.

Although our original application was towards distributed application management, we have also been investigating the use of these detection protcols in the scope of debugging distributed systems [3]. The constraints of a debugger are slightly different from those that arise in tool integration or distributed application management. For example, invasiveness is traditionally considered untolerable, yet in tool integration, temporarily blocking an application may be acceptable.

The work most similar in spirit to ours are the protocols developed by Spezialetti [16]. 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, the execution in Figure 8 shows such an execution. If the messages in this figure correspond to the messages generated in establishing simultaneous regions [15], then her protocol will not detect $x_1 = x_2$, yet in fact *definitely* $x_1 = x_2$ holds.

References

[1] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

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

[3] Robert C. B. Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173. ACM/ONR, 1991.

[4] C. J. Fidge. Partial orders for parallel debugging. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194. ACM, 1988.

[5] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of the Fourth Symposium on Software Development Environments*, pages 1–10. ACM SIGSOFT, 1990.

[6] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F*, pages 477–498. Springer-Verlag, New York, 1985.

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

[8] Keith Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304, November 1990.

[9] Keith Marzullo, Robert C. B. Cooper, Mark Wood, and Kenneth P. Birman. Tools for distributed application management. *IEEE Computer*, 24(8):42–51, August 1991.

[10] Keith Marzullo and Laura Sabel. Using consistent subcuts for detecting stable properties. In *Proceedings of the Fifth Workshop on Distributed Algorithms and Graphs*, October 1991. To appear.

[11] Keith Marzullo and Mark Wood. Tools for distributed application management. In *Proceedings of the Spring 1991 EurOpen Conference*, pages 477–498, May 1991.

[12] Friedemann Mattern. Time and global states of distributed systems. In Michel Cosnard et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1989.

[13] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 316–323. IEEE, 1988.

[14] Steven P. Reiss. Connecting tools using message passing in the FIELD program development environment. *IEEE Software*, 7(4), July 1990.

[15] M. Spezialetti and J. P. Kearns. Simultaneous 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.

[16] Madelene Spezialetti. *A Generalized Approach to Monitoring Distributed Computations for Event Occurrences*. Ph.D. dissertation. University of Pittsburgh, 1989.

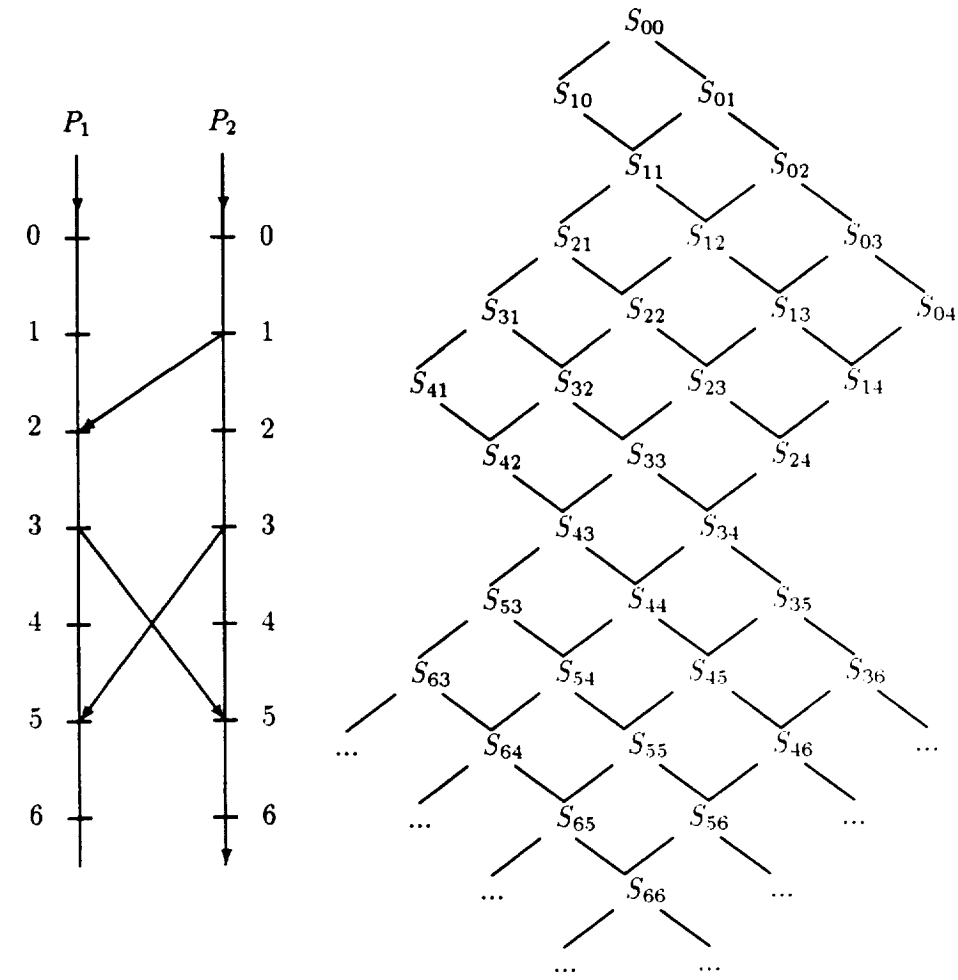

Figure 5: An execution and the corresponding lattice of global states.

```
Possibly(Φ): begin
    current := global state ⟨s_1^0, s_2^0, ..., s_n^0⟩;
    lvl := 0;
    do no state in current satisfies Φ →
        last := current
        lvl := lvl + 1;
        current := states of level lvl reachable from a state in last;
        od
    end;
    report Possibly Φ
```

Figure 6: Algorithm for detecting *Possibly* Φ.

```
Definitely(Φ): begin
    last := global state ⟨s_1^0, s_2^0, ..., s_n^0⟩;
    remove all states in last that satisfy Φ;
    lvl := 1;
    % Invariant: last contains all states of level lvl − 1 that are accessible
    % from ⟨s_1^0, s_2^0, ..., s_n^0⟩ without passing through a state satisfying Φ
    do last ≠ { } →
        current := states of level lvl reachable from a state in last
        remove all states in current that satisfy Φ;
        lvl := lvl + 1; last := current
        od
    end;
    report Definitely Φ
```

Figure 7: Algorithm for detecting *Definitely* Φ.

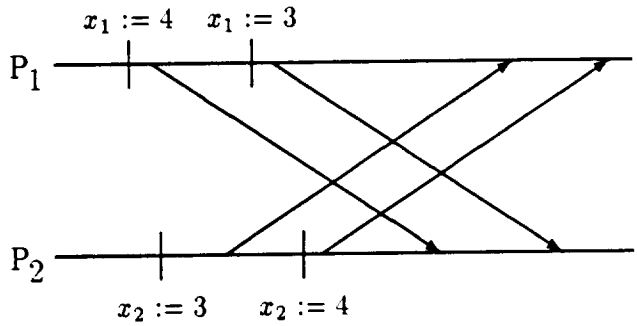

Figure 8: $\Phi \equiv (x_1 = x_2)$