Detecting Global Predicates in Distributed Systems with Clocks*

Scott D. Stoller†

January 13, 1999

Abstract

This paper proposes a framework for detecting global state predicates in systems of processes with approximately-synchronized real-time clocks. Timestamps from these clocks are used to define two orderings on events: "definitely occurred before" and "possibly occurred before". These orderings lead naturally to definitions of 3 distinct *detection modalities*, *i.e.*, 3 meanings of "predicate Φ held during a computation", namely: $\mathbf{Poss}^{\overset{db}{\rightarrow}}\,\Phi$ ("Φ possibly held"), $\mathbf{Def}^{\overset{db}{\rightarrow}}\,\Phi$ ("Φ definitely held"), and $\mathbf{Inst}\,\Phi$ ("Φ definitely held in a specific global state"). This paper defines these modalities and gives efficient algorithms for detecting them. The algorithms are based on algorithms of Garg and Waldecker, Alagar and Venkatesan, Cooper and Marzullo, and Fromentin and Raynal. Complexity analysis shows that under reasonable assumptions, these real-time-clock-based detection algorithms are less expensive than detection algorithms based on Lamport's happened-before ordering. Sample applications are given to illustrate the benefits of this approach.

Index terms: global predicate detection, consistent global states, partially-synchronous systems, distributed debugging, real-time monitoring

1 Introduction

A *history* of a distributed system can be modeled as a sequence of events. Since execution of a particular sequence of events leaves the system in a well-defined global state, a history uniquely determines the sequence of global states through which the system has passed. Unfortunately, in a distributed system without perfect clock synchronization, it is, in general,

*A preliminary description of this work appeared in [30].

†Scott D. Stoller (stoller@cs.indiana.edu, www.cs.indiana.edu/~stoller/) is with the Department of Computer Science, Indiana University, Bloomington, IN.

impossible for a process to determine the order in which events on different processors actually occurred. Therefore, no process can determine the sequence of global states through which the system passed. This leads to an obvious difficulty for detecting whether a global state predicate (hereafter simply called a "predicate") held.

Cooper and Marzullo proposed a solution for asynchronous distributed systems [6]. Their solution involves two modalities, which we denote by $\mathbf{Poss}^{\stackrel{hb}{\rightarrow}}$ ("possibly") and $\mathbf{Def}^{\stackrel{hb}{\rightarrow}}$ ("definitely"). These modalities are based on logical time [18] as embodied in the *happened-before* relation $\stackrel{hb}{\rightarrow}$, a partial ordering[1] of events that reflects potential causal dependencies. Happened-before is not a total order, so it does not uniquely determine the history, but it does restrict the possibilities. Given a predicate Φ, a computation satisfies $\mathbf{Poss}^{\stackrel{hb}{\rightarrow}}\Phi$ iff there is *some* interleaving of events that is consistent with happened-before and in which the system passes through a global state satisfying Φ. A computation satisfies $\mathbf{Def}^{\stackrel{hb}{\rightarrow}}\Phi$ iff for *every* interleaving of events that is consistent with happened-before, the system passes through a global state satisfying Φ.

Cooper and Marzullo's definitions of these modalities established an important conceptual framework for predicate detection in asynchronous systems, which has been the basis for considerable research [8, 13, 4, 17, 32, 14, 5, 12]. In practice, though, $\mathbf{Poss}^{\stackrel{hb}{\rightarrow}}$, $\mathbf{Def}^{\stackrel{hb}{\rightarrow}}$, and other modalities based on happened-before have significant drawbacks in many cases. First, in many systems, it is difficult to determine the happened-before relation. Happened-before can be determined if each process maintains a vector clock. This requires that a vector timestamp with $O(N)$ components be attached to every message, where N is the number of processes in the system, and imposes computational overhead of $O(N)$ operations per message received (to update the vector clock). Generating code that inserts and removes the vector timestamps without changing the existing types in the programs (which would open a can of worms) or copying entire messages (which is inefficient) can be difficult. If the programs use a stream-oriented communication protocol that does not provide message boundaries, such as TCP, the difficulty is significantly compounded, since a receiver might receive a fragment of a "message" or several "messages" in a single receive event. Furthermore, piggybacking vector timestamps requires changing all communication statements in the application, even if the predicate of interest involves the state of only one module. If source code is not available for part of the system, this might be impossible. Happened-before can be determined without vector clocks, if all processes inform the monitor of all send events and receive events and provide the monitor with enough information to determine the correspondence between send and receive events (*i.e.*, for each receive event, the monitor can determine which send event sent the received message). However, this method

[1] In this paper, all partial orderings are irreflexive unless specified otherwise.

often has significant drawbacks, too. In general, determining the correspondence between send and receive events requires piggybacking an identifier (*e.g.*, a sequence number) on each message; this involves the same difficulties as piggybacking a vector timestamp.[2] An additional drawback of this method is that the monitor must be informed of all send events and receive events. With vector timestamps (or if the happened-before relation is not needed), it suffices to inform the monitor only of events that might change the truth value of the predicate of interest;[3] this can significantly reduce the amount of information sent to the monitor.

A second drawback of detecting $\mathbf{Poss}^{\overset{hb}{\to}}\,\Phi$ or $\mathbf{Def}^{\overset{hb}{\to}}\,\Phi$ is the computational cost: the worst-case time complexity is $\Omega(E^N)$, where E is the maximum number of events executed by any process. The worst case occurs when there is little or no communication and hence few causal dependencies, so that many interleavings must be explored. For example, this exponential cost was seen in two sample applications considered in [31], namely, a coherence protocol and a spanning-tree algorithm. A third drawback is that, in systems with *hidden channels* [2] (*i.e.*, means of communication other than messages), happened-before does not accurately capture causal relationships, so $\mathbf{Poss}^{\overset{hb}{\to}}\,\Phi$ and $\mathbf{Def}^{\overset{hb}{\to}}\,\Phi$ do not accurately capture the meanings of "possibly held" and "definitely held".

This paper proposes a framework for predicate detection in systems with approximately-synchronized real-time clocks. Timestamps from these clocks can be used to define two orderings on events: $\overset{db}{\to}$ ("definitely occurred before") and $\overset{pb}{\to}$ ("possibly occurred before"). By (roughly speaking) substituting each of these orderings for happened-before in the definitions of $\mathbf{Poss}^{\overset{hb}{\to}}$ and $\mathbf{Def}^{\overset{hb}{\to}}$, we obtain definitions of four new modalities. The two modalities based on $\overset{db}{\to}$ are closely analogous to $\mathbf{Poss}^{\overset{hb}{\to}}$ and $\mathbf{Def}^{\overset{hb}{\to}}$, so we denote them by $\mathbf{Poss}^{\overset{db}{\to}}$ and $\mathbf{Def}^{\overset{db}{\to}}$. We obtain algorithms for detecting $\mathbf{Poss}^{\overset{db}{\to}}$ and $\mathbf{Def}^{\overset{db}{\to}}$ by adapting algorithms of Garg and Waldecker [13, 14], Alagar and Venkatesan [1], and Cooper and Marzullo [6]. Modalities based on $\overset{pb}{\to}$ are quite different, because $\overset{pb}{\to}$ (unlike $\overset{hb}{\to}$ and $\overset{db}{\to}$) is not a partial ordering. In fact, $\overset{pb}{\to}$ yields a degenerate case, in which the analogues of $\mathbf{Poss}^{\overset{hb}{\to}}$ and $\mathbf{Def}^{\overset{hb}{\to}}$ are equivalent. We show that this single modality, which we denote by **Inst**, is closely related to Fromentin and Raynal's concept of $\mathbf{Properly}^{\overset{hb}{\to}}$ [9, 10], and we adapt for detecting **Inst** an algorithm of theirs for detecting $\mathbf{Properly}^{\overset{hb}{\to}}$.

Our detection framework is applicable to a wide range of systems, since it does not require that clocks be synchronized to within a fixed bound. However, the quality of clock synchronization does affect the two event orderings just described and therefore the results

[2] Even if the underlying communication protocol uses sequence numbers, operating-system protection mechanisms may prevent the monitoring system from accessing them.

[3] This optimization is not explicitly incorporated in our algorithms, but that is easily done.

of detection. For example, consider **Inst** Φ. Informally, a computation satisfies **Inst** Φ iff the timestamps imply that there was an instant during the computation when predicate Φ held, *i.e.*, iff there is some collection of local states that form a global state satisfying Φ and that, based on the timestamps, definitely overlapped in time. Suppose Φ actually holds in a global state g that persists for time δ. Whether **Inst** Φ holds depends on the quality of synchronization. Roughly, if the maximum difference between clocks is known to be less than δ, then **Inst** Φ holds; otherwise, there is in some cases no way to determine whether the local states in g actually overlapped in time, so **Inst** Φ might not hold.

The quality of clock synchronization affects also the cost of detection. For example, consider $\mathbf{Poss}^{\overset{db}{\to}}\,\Phi$. Informally, a computation satisfies $\mathbf{Poss}^{\overset{db}{\to}}\,\Phi$ iff there is some collection of local states that form a global state satisfying Φ and that, based on the timestamps, possibly overlapped in time. The larger the error in clock synchronization, the more combinations of local states possibly overlap. In general, Φ must be evaluated in each such combination of local states. Thus, the larger this error, the more expensive the detection. If this error is bounded relative to the mean interval between relevant events (*i.e.*, events that potentially truthify or falsify Φ), then the number of global states that must be checked is linear in E. In the asynchronous case, the number of global states that must be checked is $O(E^N)$.

The above condition on the error in clock synchronization holds in many systems. In most local-area distributed systems, protocols like NTP can efficiently maintain synchronization of clocks to within a few milliseconds [26]. Even in extremely wide-area distributed systems like the Internet, clock synchronization can usually be maintained to within a few tens of milliseconds [24, 26]. The detection framework and algorithms proposed here are designed to provide a basis for monitoring and debugging applications in such systems. Some sample applications are described in Section 7, including applications in which timers provide a hidden channel, causing detection based on happened-before to be less appropriate.

Directions for future work include: implementing the detection algorithms described above; developing efficient algorithms for detecting global properties that depend explicitly on time; and investigating clock-based detection of sequences of global states, perhaps along the lines of temporal modalities based on happened-before [16, 3, 11].

2 Related Work

Marzullo and Neiger [23] discuss global property detection in partially-synchronous systems in which a fixed bound ϵ on the error between clocks is known. In the notation of this paper, they define modalities $\mathbf{Poss}^{\overset{hd}{\to}}$ and $\mathbf{Def}^{\overset{hd}{\to}}$, where $\overset{hd}{\to} \triangleq \overset{db}{\to} \cup \overset{hb}{\to}$, and give detection algorithms for these two modalities. Combining happened-before and real-time ordering exploits more

information about the computation and hence is certainly desirable whenever it is feasible. Modifying the algorithms in this paper to take happened-before into account is a straightforward exercise, and the resulting algorithms would be appropriate for monitoring some systems. This paper presents algorithms that do not use happened-before for three reasons. First and most important, as discussed in Section 1, it is often difficult in practice to modify a system so that the monitor can determine the happened-before relation; consequently, detection algorithms that depend on happened-before have limited applicability.[4] Second, not using happened-before enables some optimizations (specifically, those involving priority queues) that are impossible if causal ordering is also used. Third, incorporating happened-before would have obscured the presentation and complexity analysis of the real-time-based parts of the algorithms, which are the novel parts.

Contributions of this paper relative to [23] include: detection algorithms based purely on real-time clocks; more efficient detection algorithms; and definition of and algorithm for **Inst**. [23] does not consider any modality analogous to **Inst**. Also, [23] assumes a fixed bound on the error in clock synchronization. Our framework allows that bound to vary over time; this supports tighter bounds hence more accurate monitoring results.

An attractive feature of $\mathbf{Properly}^{\overset{hb}{\to}}$ [9, 10] and **Inst** is that the monitor can report a single global state g satisfying Φ that the system actually passed through. **Def** does not have this feature. However, $\mathbf{Properly}^{\overset{hb}{\to}}$ gives useful information only about systems that perform global (i.e., system-wide) barrier synchronization. Such synchronization is expensive and rarely used. In contrast, assuming reasonably good clock synchronization, **Inst** is informative even in the absence of barrier synchronization. Since **Inst**, like $\mathbf{Properly}^{\overset{hb}{\to}}$, can be detected efficiently for arbitrary predicates, it appears to be a useful modality.

The "possibly occurred before" relation $\overset{pb}{\to}$ is reminiscent of Lamport's "can affect" relation for concurrent systems [20, 21]. Both relations may contain cycles because of overlap: for $\overset{pb}{\to}$, overlap of interval timestamps; for "can affect", overlap of non-atomic events. Our framework assumes events are atomic; this is appropriate for systems with message-based communication.

Veríssimo [33] discusses the uncertainty in event orderings caused by the granularity[5] and imperfect synchronization of digital real-time clocks, analyzes the conditions under which this uncertainty is significant for an application, and describes a synchronization technique, suitable for certain applications, that masks this uncertainty. However, [33] does not aim for a general approach to detecting global properties in the presence of this uncertainty.

[4] Our algorithms apply directly even to programs that use high-level communication libraries (*e.g.*, a distributed shared memory (DSM) library) for which source code is not available; detecting happened-before in such cases would be difficult.

[5] Our framework accommodates the granularity of digital clocks by using $\leq$ instead of $<$ in TS1 and TS2.

3 Background

A computation of a single process is called a *local computation* and is represented as a finite or infinite sequence of local states and events. Thus, a local computation has the form

$$e_1,\ s_1,\ e_2,\ s_2,\ e_3,\ s_3,\ \ldots \tag{1}$$

where the e_α are events, and the s_α are local states. By convention, e_1 corresponds to creation of the process. If the sequence is finite, it ends with an event that corresponds (by convention) to the termination of the process.

A *computation* of a distributed system is a collection of local computations, one per process. A computation is represented as a function from process names to local computations. We use integers $1, 2, \ldots, N$ as process names; thus, for a computation c, the local computation of process i is $c(i)$. Variables i and j always range over process names. We use $Ev(c)$ and $St(c)$ to denote the sets of all events and all local states, respectively, in a computation c. For convenience, we assume that all events and local states in a computation are distinct.

The following functions are implicitly parameterized by a computation; the computation being considered should be evident from the context. For an event e, $pr(e)$ denotes the process on which e occurs. For a local state s, $pr(s)$ denotes the process that passes through s, and $\mathcal{S}(s)$ and $\mathcal{T}(s)$ denote the start event and terminal event, respectively, of s. For example, for a computation containing local computation (1), $\mathcal{S}(s_2)$ is e_2, and $\mathcal{T}(s_2)$ is e_3.

A *global state* of a distributed system is a collection of local states, one per process, represented as a function from process names to local states. The set of global states of a computation c is denoted $GS(c)$; thus, g is in $GS(c)$ iff for each process i, $g(i)$ is a local state in $c(i)$. We define a reflexive partial ordering $\preceq$ on global states by:

$$g \preceq g' \ \triangleq\ (\forall i : g(i) = g'(i) \ \vee\ g(i) \text{ occurs before } g'(i)). \tag{2}$$

All of the orderings defined in this paper, including $\preceq$, are implicitly parameterized by a computation; the computation being considered should be evident from the context.

Each event e has an *interval timestamp* $C(e)$, which is an interval with lower endpoint $C_1(e)$ and upper endpoint $C_2(e)$. We model events as being atomic and instantaneous; the width of the interval timestamp depends only on the quality of clock synchronization when the event occurs. We assume that the interval timestamps are non-decreasing and are consistent with the order of events; more precisely, we assume:

TS1. For every event e, $C_1(e) \leq C_2(e)$.

TS2. For every event e_1 with an immediately succeeding event e_2 on the same process,

$C_1(e_1) \leq C_1(e_2)$ and $C_2(e_1) \leq C_2(e_2)$.

TS3. For every event e_1 and every event e_2, if e_1 occurred before e_2, then $C_1(e_1) \leq C_2(e_2)$.

There are various ways of satisfying these assumptions, depending on the underlying clock synchronization mechanism. As a simple (yet realistic) example, if the clock synchronization algorithm never decreases the value of a clock, and if all of the machines (more precisely, all of the machines relevant to the predicate being detected) synchronize to a single time server, then TS1–TS3 hold if timestamps are chosen such that the value of the time server's clock was in the interval $C(e)$ when event e occurred. Thus, a machine can take $C_1(e) = t - \varepsilon$ and $C_2(e) = t + \varepsilon$, where t is the value of its local clock when e occurred, and ε is a bound on the difference between its clock and the time server's clock when e occurred. In systems in which time servers are organized in peer groups or in a client-server hierarchy, $C(e)$ can be determined from an appropriate combination of the bounds on the errors between the relevant clocks. In either case, the information needed to construct interval timestamps can be obtained from standard clock synchronization subsystems, such as NTP [25, 26] or the Distributed Time Service in OSF DCE [27].

4 Generic Theory of Consistent Global States

Predicate detection in asynchronous systems is based on the theory of consistent global states (CGSs) [2]. Informally, a global state is consistent if it could have occurred during the computation. It is convenient to define "consistent" in terms of ideals. Recall that an *ideal* of a partial order $\langle S, \prec \rangle$ is a set $I \subseteq S$ such that $(\forall x \in I : \forall y \in S : y \prec x \Rightarrow y \in I)$. Ideals of $\langle Ev(c), \xrightarrow{hb} \rangle$ are called *consistent cuts* [2]. Recall that for any partial order, the set of its ideals ordered by inclusion ($\subseteq$) forms a lattice [8]. Furthermore, the set of CGSs ordered by $\preceq$ forms a lattice that is isomorphic to the lattice of consistent cuts [28, 2]. This isomorphism has an important consequence for detection algorithms: it implies that a minimal increase with respect to $\preceq$ corresponds to advancing one process by one event (because adjacent ideals of $\langle Ev(c), \xrightarrow{hb} \rangle$ differ by exactly one event) and hence that the lattice of CGSs can be explored by repeatedly advancing one process by one event. This principle underlies detection algorithms of Cooper and Marzullo [6], Garg and Waldecker [13, 14], and Alagar and Venkatesan [1].

In this section, we show that the above theory is not specific to the happened-before relation but rather applies to any partial ordering $\hookrightarrow$ on events, provided $\hookrightarrow$ is *process-wise-total*, *i.e.*, for any two events e_1 and e_2 on the same process, if e_1 occurred before e_2, then $e_1 \hookrightarrow e_2$. This generalized theory underlies the detection algorithms in Sections 5 and 6.

Definition of CGSs. Let c be a computation, and let $\hookrightarrow$ be a relation on $Ev(c)$. We define a relation $\rightsquigarrow^{\hookrightarrow}$ on $St(c)$, with the informal interpretation: $s \rightsquigarrow^{\hookrightarrow} s'$ if s ends before s' starts. Formally,

$$s \rightsquigarrow^{\hookrightarrow} s' \;\triangleq\; \begin{cases} \mathcal{S}(s) \hookrightarrow \mathcal{S}(s') & \text{if } pr(s) = pr(s') \\ \mathcal{T}(s) \hookrightarrow \mathcal{S}(s') & \text{if } pr(s) \neq pr(s'). \end{cases} \tag{3}$$

Two local states are *concurrent* with respect to $\hookrightarrow$ if they are not related by $\rightsquigarrow^{\hookrightarrow}$. A global state is *consistent* with respect to $\hookrightarrow$ if its constituent local states are pairwise concurrent:

$$\text{consis}^{\hookrightarrow}(g) \;\triangleq\; (\forall i, j : i \neq j \Rightarrow \neg(g(i) \rightsquigarrow^{\hookrightarrow} g(j))). \tag{4}$$

Thus, the set of CGSs of computation c with respect to $\hookrightarrow$ is

$$CGS^{\hookrightarrow}(c) = \{g \in GS(c) \mid \text{consis}^{\hookrightarrow}(g)\}. \tag{5}$$

Note that $CGS^{\stackrel{hb}{\to}}$ is the usual notion of CGSs.

Definitions of Poss and Def. The modalities $\mathbf{Poss}^{\stackrel{hb}{\to}}$ and $\mathbf{Def}^{\stackrel{hb}{\to}}$ for asynchronous systems are defined in terms of the lattice $\langle CGS^{\stackrel{hb}{\to}}(c), \preceq\rangle$. We generalize them as follows.

> A computation c satisfies $\mathbf{Poss}^{\hookrightarrow}\,\Phi$ iff $CGS^{\hookrightarrow}(c)$ contains a global state satisfying Φ.

$\mathbf{Def}^{\hookrightarrow}$ is defined in terms of paths. A *path* through a partial order $\langle S, \preceq\rangle$ is a finite or infinite sequence[6] σ of distinct elements of S such that: (*i*) $\sigma[1]$ is minimal with respect to $\preceq$; (*ii*) for all $\alpha \in [1..|\sigma| - 1]$, $\sigma[\alpha + 1]$ is an immediate successor[7] of $\sigma[\alpha]$; and (*iii*) if σ is finite, then $\sigma[|\sigma|]$ is maximal with respect to $\preceq$. Informally, each path through $\langle CGS^{\hookrightarrow}(c), \preceq\rangle$ corresponds to an order in which the events in the computation could have occurred.

> A computation c satisfies $\mathbf{Def}^{\hookrightarrow}\,\Phi$ iff every path through $\langle CGS^{\hookrightarrow}(c), \preceq\rangle$ contains a global state satisfying Φ.

CGSs and Ideals. When $\hookrightarrow$ is a process-wise-total partial ordering, there is a natural correspondence between $CGS^{\hookrightarrow}(c)$ and ideals of $\langle Ev(c), \hookrightarrow\rangle$. One can think of an ideal I as the set of events that have occurred. Executing a set I of events leaves each process i in the local state immediately following the last event of process i in I. Thus, ideal I corresponds

[6] We use 1-based indexing for sequences.

[7] For a reflexive or irreflexive partial order $\langle S, \prec\rangle$ and elements $x \in S$ and $y \in S$, y is an *immediate successor* of x iff $x \neq y \;\wedge\; x \prec y \;\wedge\; \neg(\exists\, z \in S \setminus \{x, y\} : x \prec z \wedge z \prec y)$.

to the global state g such that for all i, $\mathcal{S}(g(i))$ is the maximal element of $\{e \in I \mid pr(e) = i\}$. This correspondence is an isomorphism.

Theorem 1. For every process-wise-total partial ordering $\hookrightarrow$ on $Ev(c)$, the partial order $\langle CGS^{\hookrightarrow}(c), \preceq \rangle$ is a lattice and is isomorphic to the lattice of ideals of $\langle Ev(c), \hookrightarrow \rangle$.

Proof. This is true for the same reasons as in the standard theory based on happened-before [28, 2, 8]. The proof is straightforward. ∎

The following corollary underlies the detection algorithms in Sections 5 and 6.

Corollary 2. For any process-wise-total partial ordering $\hookrightarrow$, if global state g' is an immediate successor of g in $\langle CGS^{\hookrightarrow}(c), \preceq \rangle$, then the ideal corresponding to g' contains exactly one more event than the ideal corresponding to g.

Proof. This follows from Theorem 1 and the fact that for any partial order S, if one ideal of S is an immediate successor of another ideal of S, then those two ideals differ by exactly one element. ∎

5 Detection Based on a Strong Event Ordering: Poss$^{\stackrel{db}{\rightarrow}}$ and Def$^{\stackrel{db}{\rightarrow}}$

We instantiate the generic theory in Section 4 with the partial ordering $\stackrel{db}{\rightarrow}$ ("definitely occurred before"), defined by:

$$e_1 \stackrel{db}{\rightarrow} e_2 \triangleq \begin{cases} e_1 \text{ occurs before } e_2 & \text{if } pr(e_1) = pr(e_2) \\ C_2(e_1) < C_1(e_2) & \text{if } pr(e_1) \neq pr(e_2). \end{cases} \tag{6}$$

This ordering cannot be defined solely in terms of the timestamps $C(e_1)$ and $C(e_2)$, because TS1 and TS2 allow consecutive events on a process to have identical timestamps. Therefore, we assume that a process records a local sequence number as well as an interval timestamp for each event.

Theorem 3. For every computation c, $\stackrel{db}{\rightarrow}$ is a process-wise-total partial ordering on $Ev(c)$.

Proof. See Appendix. ∎

By the discussion in Section 4, $\stackrel{db}{\rightarrow}$ induces a notion $CGS^{\stackrel{db}{\rightarrow}}$ of CGSs. If $g \in CGS^{\stackrel{db}{\rightarrow}}(c)$, then the local states in g possibly overlapped in time. For example, Figure 1 shows a computation c_1 and the lattice $\langle CGS^{\stackrel{db}{\rightarrow}}(c_1)), \preceq \rangle$. The pair of arcs enclosing each event show the endpoints of the interval timestamp. In the lattice, a node labeled i, j represents the global state in which process 1 is local state s^1_i and process 2 is in local state s^2_j.

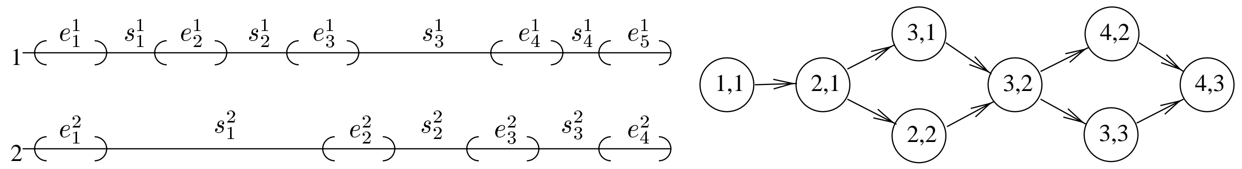

Figure 1: Left: A computation c_1. Right: The lattice $\langle CGS^{\stackrel{db}{\to}}(c_1), \preceq \rangle$.

We consider in this paper only detection algorithms with a passive monitor. Each process in the original system sends its timestamped local states to a new process, called the *monitor*. More specifically, when a process executes an event, thereby terminating its current local state s, the process sends to the monitor a message containing s and the timestamps $C(\mathcal{S}(s))$ and $C(\mathcal{T}(s))$.[8]

We consider only *on-line* detection, in which the monitor detects the property as soon as possible. Algorithms for *off-line* detection, in which the monitor waits until the computation has terminated before checking whether a property holds, can be obtained as special cases. We consider first algorithms for detecting $\mathbf{Poss}^{\stackrel{db}{\to}}$ for a restricted class of predicates and then consider general algorithms for detecting $\mathbf{Poss}^{\stackrel{db}{\to}}$ and $\mathbf{Def}^{\stackrel{db}{\to}}$.

5.1 Algorithms for $\mathbf{Poss}^{\stackrel{db}{\to}}$ and $\mathbf{Def}^{\stackrel{db}{\to}}$ for Conjunctive Predicates

Garg and Waldecker [13, 14] have developed efficient algorithms for detecting $\mathbf{Poss}^{\stackrel{hb}{\to}} \Phi$ and $\mathbf{Def}^{\stackrel{hb}{\to}} \Phi$ for conjunctive predicates Φ. A predicate is *conjunctive* if it is a conjunction of predicates that each depend on the local state of one process. For example, if x_i is a local variable of process i, then the predicate $x_1 > 0 \wedge x_2 < 0$ is conjunctive, and the predicate $x_1 > x_2$ is not conjunctive. Their algorithms can be adapted in a straightforward way to detect $\mathbf{Poss}^{\stackrel{db}{\to}}$ and $\mathbf{Def}^{\stackrel{db}{\to}}$, by (roughly) replacing comparisons based on happened-before with comparisons based on $\stackrel{db}{\to}$. This yields detection algorithms with worst-case time complexity $O(N^2E)$, where E is the maximum number of events executed by any process. The worst-case time complexity of both algorithms can be reduced to $O((N \log N)E)$ by exploiting the total ordering on numbers.

We start by reviewing Garg and Waldecker's algorithm for detecting $\mathbf{Poss}^{\stackrel{hb}{\to}} \Phi$ for conjunctive predicates. Suppose the predicate of interest is $\Phi = \bigwedge_{i=1}^{N} \phi_i$, where ϕ_i depends on

[8]Several straightforward optimizations are possible. For example, each message might describe only the differences between consecutive reported local states, rather than repeating the entire local state. Also, except for the initial local state, it suffices to include with local state s only the timestamp $C(\mathcal{T}(s))$, since $C(\mathcal{S}(s))$ was sent in the previous message to the monitor. Also, for a given predicate Φ, events that cannot possibly truthify or falsify Φ can be ignored.

the local state of process i. Each process i sends to the monitor timestamped local states satisfying ϕ_i; local states not satisfying ϕ_i are not reported. For each process i, the monitor maintains a queue q_i and adds each timestamped local state received from process i to the end of q_i. Let head(q) denote the head of a non-empty queue q. If for some i and j, $\text{head}(q_i) \stackrel{hb}{\to} \text{head}(q_j)$, then head$(q_i)$ is removed from q_i. The heads of the queues are repeatedly compared and sometimes removed in this way, until the heads of the non-empty queues are pairwise concurrent. At that point, if all the queues are non-empty, then the heads of the queues form a CGS satisfying Φ, so the algorithm returns that CGS (thereby indicating that $\mathbf{Poss}^{\stackrel{hb}{\to}} \Phi$ holds); if some queue is empty, then the monitor waits to receive more local states and then repeats the procedure just described. The worst-case time complexity is $O(N^2E)$, because there are $O(NE)$ local states, and each time a local state is removed from q_i, the new head of q_i is compared with the heads of the other $O(N)$ queues.

For detection of $\mathbf{Poss}^{\stackrel{db}{\to}} \bigwedge_{i=1}^{N} \phi_i$, the number of comparisons can be reduced as follows. Expanding the definition of $CGS^{\stackrel{db}{\to}}(c)$, a global state g is consistent iff

$$(\forall i, j : i \neq j \;\Rightarrow\; C_2(\mathcal{T}(g(i))) \geq C_1(\mathcal{S}(g(j)))). \tag{7}$$

Using the fact that for all i, $C_2(\mathcal{T}(\text{head}(g(i)))) \geq C_1(\mathcal{S}(\text{head}(g(i))))$, which follows from TS1 and TS2, one can show that (7) is equivalent to

$$\min_i(C_2(\mathcal{T}(\text{head}(g(i))))) \;\geq\; \max_i(C_1(\mathcal{S}(\text{head}(g(i))))). \tag{8}$$

To evaluate (8) efficiently, we maintain two priority queues p_1 and p_2, whose contents are determined by the invariants:

I1: For each process i such that q_i is non-empty, p_1 contains a record with key $C_1(\mathcal{S}(\text{head}(q_i)))$ and satellite data i. p_1 contains no other records.

I2: For each process i such that q_i is non-empty, p_2 contains a record with key $C_2(\mathcal{T}(\text{head}(q_i)))$ and satellite data $\langle i, ptr\rangle$, where ptr is a pointer to the record with satellite data i in p_1. p_2 contains no other records.

Recall that the operations on a priority queue p include getMin(p), which returns a record $\langle k, d\rangle$ with key k and satellite data d such that k is minimal, and extractMin(p), which removes and returns such a record. We also use priority queues with analogous operations based on maximal key values. Thus, (8) is equivalent to

$$\text{key}(\text{getMin}(p_2)) \geq \text{key}(\text{getMax}(p_1)), \tag{9}$$

where $\text{key}(\langle k, d\rangle) = k$. The negation of (9) is used in the **while** loop in Figure 2 to check the heads of the non-empty queues are concurrent. Let PossConjAlg denote the algorithm in Figure 2. If a computation satisfies $\mathbf{Poss}^{\overset{db}{\to}} \Phi$, PossConjAlg($\Phi$) returns a CGS satisfying Φ.

To analyze the time complexity, recall that an operation on a priority queue containing n records takes $O(\log n)$ time. A constant number of such operations are performed for each local state, so the worst-case time complexity of the algorithm in Figure 2 is $O(EN \log N)$. Note that the time complexity is independent of the quality of clock synchronization.

The algorithm in [14] for detecting $\mathbf{Def}^{\overset{hb}{\to}} \Phi$ for conjunctive Φ can be adapted in a similar way to detect $\mathbf{Def}^{\overset{db}{\to}} \Phi$ for such predicates.

```
On receiving x from process i:
append(q_i, x);
if head(q_i) = x then
   add records for i to p_1 and p_2, to maintain invariants I1 and I2;
   while ¬empty(p_1) ∧ key(getMin(p_2)) < key(getMax(p_1))
      ⟨k, ⟨i, ptr⟩⟩ := extractMin(p_2);
      remove record for i (i.e., record *ptr) from p_1;
      removeHead(q_i);
      if ¬empty(q_i) then
         add records for i to p_1 and p_2, to maintain invariants I1 and I2;
      endif
   endwhile
   if (∀i : ¬empty(q_i)) then
      return the CGS ⟨head(q_1), ..., head(q_N)⟩
   endif
endif
```

Figure 2: Algorithm PossConjAlg(Φ) for detecting $\mathbf{Poss}^{\overset{db}{\to}} \Phi$ for conjunctive predicates. Process i sends to the monitor only local states satisfying its local predicate.

5.2 General Algorithm for $\mathbf{Poss}^{\overset{db}{\to}}$

We develop an on-line detection algorithm for $\mathbf{Poss}^{\overset{db}{\to}} \Phi$ by adapting Alagar and Venkatesan's algorithm for detecting $\mathbf{Poss}^{\overset{hb}{\to}} \Phi$ in non-terminating (i.e., infinite) computations [1]. Their algorithm is based on their procedure for depth-first search of a lattice of CGSs. A depth-first exploration of the lattice of CGSs for an infinite computation would never backtrack and thus would never visit some CGSs near the beginning of the lattice. So, in their algorithm,

the lattice is divided into a sequence of sublattices $L_0, L_1, L_2, \ldots$, corresponding to increasing prefixes of the computation, and depth-first search is used to explore each sublattice $L_{i+1} - L_i$. The following paragraphs describe how to adapt their algorithm to on-line detection of $\mathbf{Poss}^{\overset{db}{\rightarrow}} \Phi$.

Finding the initial CGS. In the asynchronous setting, the initial CGS simply contains the initial local state of each process. In the timed setting, that global state might not be consistent, since the processes might have been started at different times.

Theorem 4. For every computation c, if $CGS^{\hookrightarrow}(c)$ is not empty, then $\langle CGS^{\hookrightarrow}(c), \preceq \rangle$ contains a unique minimal element, *i.e.*, c has a unique initial CGS.

Proof. The existence of minimal elements in the lattice of CGSs follows immediately from non-emptiness and the absence of infinite descending chains in $\preceq$ [7, chapter 2]. We prove by contradiction that the lattice of CGSs has a unique minimal element. Suppose CGSs g_1 and g_2 are minimal and $g_1 \neq g_2$. Let $\wedge_G$ denote the meet operation of the lattice of CGSs. Note that $g_1 \not\preceq g_2$ (by the assumptions that g_2 is minimal and $g_1 \neq g_2$) and $(g_1 \wedge_G g_2) \preceq g_2$ (by the definition of $\wedge_G$), so $(g_1 \wedge_G g_2) \neq g_1$. By definition of $\wedge_G$, $(g_1 \wedge_G g_2) \preceq g_1$, which together with $(g_1 \wedge_G g_2) \neq g_1$ contradicts the assumed minimality of g_1. ∎

To find the initial CGS, we exploit the fact that for every conjunctive predicate Φ, if a computation satisfies $\mathbf{Poss}^{\overset{db}{\rightarrow}} \Phi$, then PossConjAlg($\Phi$) finds and returns the unique minimal CGS satisfying Φ; the proof of this is closely analogous to the proof of the corresponding property of Garg and Waldecker's algorithm [14]. A corollary is: if $CGS^{\overset{db}{\rightarrow}}(c)$ is not empty, then PossConjAlg(*true*) returns the initial CGS (otherwise, PossConjAlg(*true*) never calls **return**).

Choosing the sequence of sublattices. To avoid delays in detection, when the monitor receives a timestamped local state, it constructs the largest CGS g_2 that can be constructed from the local states it has received so far; this is done by the **while** loop in Figure 4. This CGS implicitly defines the next sublattice L_{i+1}: L_{i+1} contains exactly the CGSs g such that $g \preceq g_2$. Let g_1 denote the CGS constructed when the previous local state was received, *i.e.*, the CGS corresponding to sublattice L_i. After constructing g_2, the monitor does a depth-first search of the sublattice $L_{i+1} - L_i$, which (by definition) contains CGSs g such that $g_1 \preceq g \preceq g_2$.

Exploration of a sublattice. There are two main steps in the exploration of the sublattice of CGSs between a CGS g_1 and a larger CGS g_2:

- Use a procedure $initStates(g_1, g_2)$ to compute the set S of minimal (with respect to $\preceq$) CGSs in that sublattice.
- For each CGS g in S, use a procedure $depthFirstSearch(g, g_1, g_2)$ to do a depth-first search starting from g of a fragment of that sublattice. These searches together explore the entire sublattice.

Alagar and Venkatesan observed that *initStates* can be computed efficiently as follows. For a local state s, let $minstate(s)$ be the unique minimal CGS containing s, and let $\mathrm{succ}(s)$ be the local state that occurs immediately after s on the same process (if there is no such local state, then $\mathrm{succ}(s)$ is undefined). Then $initStates(g_1, g_2)$ is given by [1]:

```
procedure initStates(g1, g2)                                        (10)
var i, S;
S := ∅
for i := 1 to N
   if g1(i) ≠ g2(i) ∧ ¬(∃g ∈ S : g ⪯ minstate(succ(g1(i)))) then
      insert(S, minstate(succ(g1(i))))
   endif
rof;
return S
```

Computing *minstate*. Our algorithm for computing *minstate* is similar to our algorithm for computing the initial CGS. It relies on the following property of PossConjAlg: if PossConjAlg(Φ) is started from a global state g (*i.e.*, for all i, local states of process i that occur before $g(i)$ are ignored), and if the remainder of the computation satisfies $\mathbf{Poss}^{\xrightarrow{db}} \Phi$, then PossConjAlg($\Phi$) finds the unique minimal CGS greater than g and satisfying Φ. For a global state g and a local state s, let $g[i \mapsto s]$ denote the global state that is the same as g except that the local state of process i is s. A simple way to compute $minstate(s)$ is to call PossConjAlg($true$) starting from the global state $g_0[pr(s) \mapsto s]$, where g_0 is the initial CGS. An optimization is sometimes possible. Consider a call $minstate(s_2)$. If *minstate* has not previously been called with a local state of $pr(s_2)$ as argument, then the optimization does not apply, and *minstate* is computed as described above. Otherwise, let s_1 be the argument in the previous call to *minstate* on a local state of $pr(s_2)$. Observe that s_1 occurred before s_2, because: (1) *minstate* is called only from *initStates*, and *initStates* is called on a non-decreasing chain of CGSs (this is a property of the algorithm in Figure 4 below), and (2) (assuming short-circuiting evaluation of $\wedge$ in *initStates*) $initStates(g_1, g_2)$ calls $minstate(\mathrm{succ}(g_1(i)))$ only if $g_1(i) \neq g_2(i)$. Since s_1 occurred before s_2, $minstate(s_1) \preceq minstate(s_2)$. So, we can start PossConjAlg($true$) from global state $minstate(s_1)[pr(s) \mapsto s]$ instead of $g_0[pr(s) \mapsto s]$. This

leads to the following algorithm. For each i, $old(i)$ contains the result of the previous call to *minstate* on a local state of process i; initially, $old(i)$ is set to g_0.

procedure $minstate(s)$
$old(pr(s))$:= the CGS returned by PossConjAlg($true$) started from $old(pr(s))[pr(s) \mapsto s]$;
return $old(pr(s))$

Depth-first search of fragment of sublattice. Since a CGS may have multiple predecessors in the lattice of CGSs, the search algorithm needs (for efficiency) some mechanism to ensure that each CGS is explored only once. A straightforward approach is to maintain a set containing the CGSs that have been visited so far. However, this may be expensive in both space and time. Alagar and Venkatesan [1] proposed the following clever alternative. Introduce a total ordering $<_{idx}$ on CGSs, defined by: $g_1 <_{idx} g_2$ if index(g_1) is lexicographically smaller than index(g_2), where for a global state g, index(g) is the tuple $\langle k_1, \ldots, k_n \rangle$ such that $g(i)$ is the k_i'th local state of process i. During the depth-first search, explore a CGS g only from the immediate predecessor (with respect to $\preceq$) of g that is maximal (among the immediate predecessors of g) with respect to $<_{idx}$. This leads to the algorithm in Figure 3, where for a CGS g, pred(g) is the set of immediate predecessors of g in the lattice of CGSs. To compute pred(g): for each process i, check whether moving process i back by one local state yields a CGS, and if so, include the resulting CGS in the set.

Putting these pieces together yields the on-line detection algorithm in Figure 4. All local states "after" g_0 (*i.e.*, local states s such that $g_0(pr(s))$ occurs before s) are handled by the "On receiving s from process i" statement, even though some of these local states might have been received before the initial CGS was found.

Recall that a process sends a local state to the monitor when that local state ends. This is natural (because $\stackrel{db}{\leadsto}$ depends on when local states end) but can delay detection. One approach to bounding and reducing this delay is for a process that has not reported an event to the monitor recently to send a message to the monitor to report that it is still in the same local state (as if reporting execution of **skip**). Another approach, described in [23], requires a bound on message latency: at each instant, the monitor can use its own local clock and this bound to determine a lower bound on the ending time of the last local state it received from a process.

5.2.1 Complexity

To analyze the time complexity, we consider separately the cost of all invocations of *minstate* and the cost of all other operations. In effect, for each process, the calls to *minstate* cause PossConjAlg algorithm to be executed N times (once for each process) on (at worst) the

```
procedure depthFirstSearch(g, g_1, g_2)
  if Φ(g) then
    return(true)
  else
    for i := 1 to n
      if g(i) ≠ g_2(i) then
        g' := g[i ↦ succ(g(i))];
        if consis^{db→}(g') then
          S := {p ∈ pred(g') | g_1 ⪯ p};
          if g is maximal in ⟨S, <_idx⟩ then
            if depthFirstSearch(g', g_1, g_2) then
              return(true)
            endif
      endif
    rof
  endif;
  return(false)
```

Figure 3: Algorithm for depth-first search.

entire computation. Thus, the total cost of calls to PossConjAlg from *minstate*, and hence the total cost of calls to *minstate*, is $O(EN^2 \log N)$. The total cost of all executions of the **while** loop in Figure 4 is $O(EN^3)$, since: (1) evaluating the loop condition takes time $O(N^2)$, and the condition is evaluated at most once for each of the $O(NE)$ local states; (2) the loop body is executed at most once for each of the $O(NE)$ local states, and each execution takes constant time. The total cost of all executions of the body of the **for** loop in Figure 4 is $O(|CGS^{\stackrel{db}{\to}}(c)|N^2)$, since the depth-first search takes $O(N^2)$ time per CGS, since evaluating pred takes $O(N^2)$ time. Each call to *initStates* takes $O(N^3)$ time (excluding the cost of calls to *minstate*), because: (1) evaluating $\preceq$ takes $O(N)$ time, and (2) $\preceq$ may be evaluated $O(N^2)$ times, because of the **for** loop and the existential quantifier. The total cost of all calls to *initStates* is $O(N^4E)$, since *initStates* is called at most once per local state. Summing these contributions, we conclude that the worst-case time complexity of the algorithm is $O(|CGS^{\stackrel{db}{\to}}(c)|N^2 + EN^4)$.

$|CGS^{\stackrel{db}{\to}}(c)|$ depends on the rate at which events occur relative to the error between clocks. To simplify the complexity analysis, suppose: (1) the interval between consecutive events at a process is at least τ, (2) the error between clocks is known to be at most ε, and (3) the interval timestamp on an event e is given by $C_1(e) = t - \varepsilon$ and $C_2(e) = t + \varepsilon$, where t is the value of the local clock of machine $pr(e)$ when e occurred. Then, for every event

```
Initialization Phase :
g0 := the CGS returned by PossConjAlg(true);      (* g0 is the initial CGS *)
g2 := g0;

For all local states s such that g0(pr(s)) occurs before s:
On receiving s from process i:
append(q_i, x);
g1 := g2;
while (∃j : ¬empty(q_j) ∧ consis^{→db}(g2[j ↦ head(q_j)]))      (* construct largest CGS *)
   g2(j) := head(q_j);
   removeHead(q_j)
endwhile;
for s in initStates(g1, g2)
   if depthFirstSearch(g, g1, g2) then
      report Poss^{→db} Φ and exit
   endif
rof
```

Figure 4: Algorithm for detecting $\mathbf{Poss}^{\overset{db}{\to}} \Phi$.

e, $C_2(e) - C_1(e) = 2\varepsilon$. If $\tau > 2\varepsilon$, then each local state is concurrent with at most 3 local states of each other process, so each local state is in at most $O(3^{N-1})$ CGSs, so there are $O(3^N E)$ CGSs, so the worst-case time complexity of the algorithm is $O(3^N E N^2)$. If $\tau \leq 2\varepsilon$, then each local state is concurrent with at most $\lceil (4\varepsilon + \tau)/\tau \rceil + 1$ local states of each other process, so there are $O((\lceil 4\varepsilon/\tau \rceil + 2)^{N-1} E)$ CGSs, so the worst-case time complexity of the algorithm is $O((\lceil 4\varepsilon/\tau \rceil + 2)^{N-1} E N^2)$. In both cases, the worst-case time complexity of detecting $\mathbf{Poss}^{\overset{db}{\to}}$ is linear in E, which is normally much larger than N; in contrast, the worst-case time complexity of general algorithms for detecting $\mathbf{Poss}^{\overset{hb}{\to}}$ and $\mathbf{Def}^{\overset{hb}{\to}}$ is $\Omega(E^N)$.

A more realistic complexity analysis requires considering distributions of inter-event times, rather than simply fixing a minimum value. Specifically, we consider distributed computations with inter-event times selected from a normal (*i.e.*, Gaussian) distribution with mean μ and standard deviation $\sqrt{\mu}$ (negative numbers selected from the distribution were ignored). For simplicity, we continue to assume a fixed bound ε on the error between clocks. The number of CGSs then depends on N, E, and the ratio μ/ε. As in the cases analyzed above, the number of CGSs scales linearly with E; this is illustrated by the graph in Figure 5. Figure 6 plots the number of CGSs *vs.* μ/ε and N. One can see that when μ/ε is large, the number of CGSs increases slowly (roughly linearly) with N; when μ/ε is small, the number of CGSs increases exponentially with N.

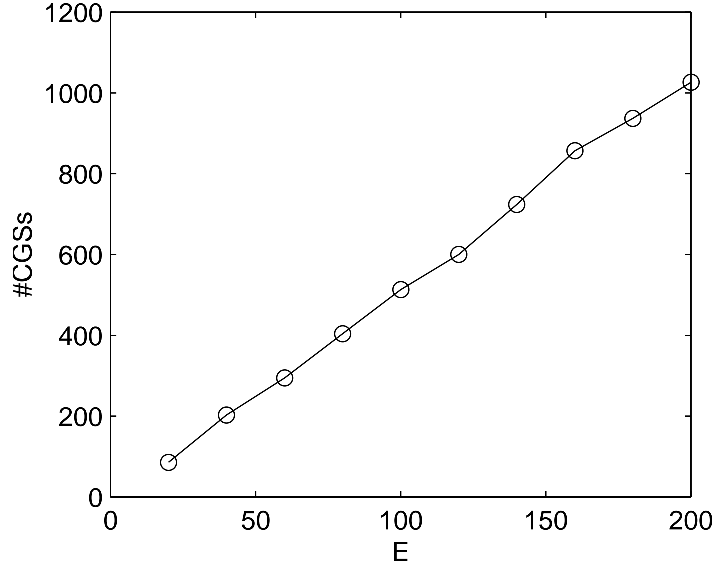

Figure 5: Number of CGSs *vs.* E, for $\mu/\varepsilon = 10$ and $N = 4$.

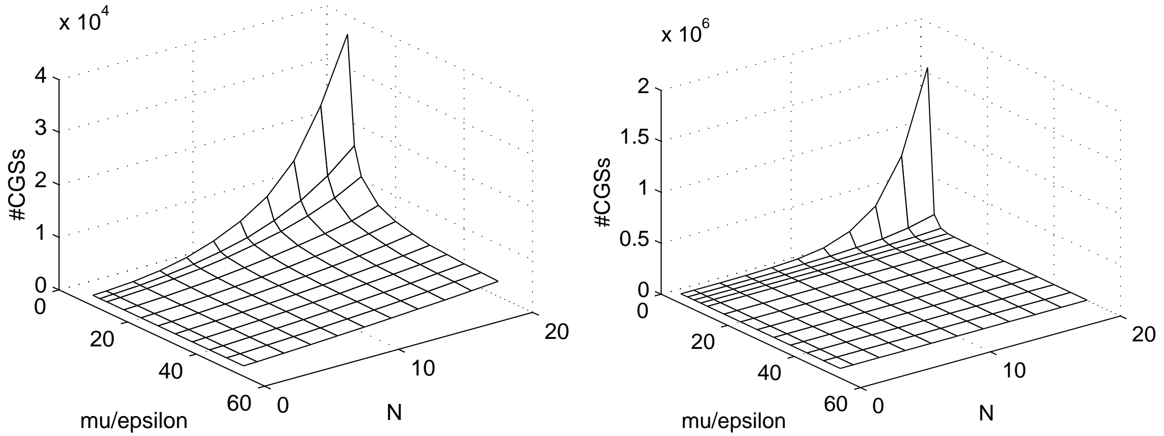

Figure 6: Left: Number of CGSs *vs.* μ/ε and N, for $E = 100$, with μ/ε ranging from 6 to 50. Right: Number of CGSs *vs.* μ/ε and N, for $E = 100$, with μ/ε from 2 to 50. Note that the vertical scales in these two graphs are very different.

5.3 General Algorithm for $\mathbf{Def}^{\stackrel{db}{\rightarrow}}$

The detection algorithm for $\mathbf{Def}^{\stackrel{hb}{\rightarrow}}\,\Phi$ in [6, 23] can be adapted to detect $\mathbf{Def}^{\stackrel{db}{\rightarrow}}\,\Phi$ by (roughly) replacing each condition of the form $e_1 \stackrel{hb}{\rightarrow} e_2$ with $e_1 \stackrel{db}{\rightarrow} e_2$. That algorithm divides the lattice

into *levels*. The level of a local state in a local computation is the number of local states preceding it in that computation. The level of a global state g is the sum of the levels of the constituent local states. Level ℓ of the lattice of CGSs contains the CGSs with level ℓ. Following [6, 23], we give an algorithm in which the monitor constructs one level of the lattice of CGSs at a time. Constructing one level of the lattice at a time is unnecessary and sometimes delays detection of a property; this construction is used only to simplify the presentation.

The algorithm used by the monitor to detect $\mathbf{Def}^{\overset{db}{\to}}\,\Phi$ is given in Figure 7. The lowest level of the lattice contains only the initial CGS. The **while** loop maintains the following invariant: *last* contains CGSs that are reachable from the initial CGS without passing through a CGS satisfying Φ. In line (†) of the algorithm, the monitor considers each global state g in *last* and each process i, and checks whether the local state $\mathrm{succ}(g(i))$ is concurrent with the local states in g of all the other processes. (The monitor waits for the local states $\mathrm{succ}(g(1)), \ldots, \mathrm{succ}(g(N))$, if they have not already arrived.) If so, the monitor adds $g[i \mapsto \mathrm{succ}(g(i))]$ to *current* if it is not already in *current*.

> $g :=$ the CGS returned by PossConjAlg(*true*); (∗ find the initial CGS ∗)
> *last* $:= \{g\}$;
> remove all CGSs in *last* that satisfy Φ;
> **while** *last* $\neq \emptyset$
>   *current* := CGSs that are immediate successors of CGSs in *last*; (†)
>   remove all CGSs in *current* that satisfy Φ;
>   *last* := *current*;
> **endwhile**;
> report $\mathbf{Def}^{\overset{db}{\to}}\,\Phi$

Figure 7: Algorithm for detecting $\mathbf{Def}^{\overset{db}{\to}}\,\Phi$.

Since Φ could be false in all CGSs, and because we assume that the cost of evaluating Φ on each global state is a constant, the worst-case asymptotic time complexity of this algorithm equals the worst-case asymptotic time complexity of constructing $CGS^{\overset{db}{\to}}$. For each CGS, the algorithm advances each of the N processes and, if the resulting global state g is consistent, the algorithm checks whether g is already in *current*. Let T_m denote the total cost of these membership checks; then constructing $CGS^{\overset{db}{\to}}(c)$ takes $\Theta(|CGS^{\overset{db}{\to}}(c)|N^2 + T_m)$ time. T_m depends on the data structure used to implement *current*. With a naive array-based implementation, each check has constant cost, so T_m is $\Theta(E^N)$, due to the cost of

initializing the arrays, so the worst-case time complexity of the algorithm is $\Theta(E^N N^2)$.[9]

However, this implementation has a serious disadvantage: the time complexity remains $\Omega(E^N)$ even if the *actual* number of CGSs is much smaller than E^N, which is typically the case. Thus, generally preferable alternatives are to implement *current* as a dictionary, using a hash table or balanced trees. Let $W(c)$ be the width of the lattice $CGS^{\stackrel{db}{\to}}(c)$, *i.e.*, the maximum size of a level of that lattice. If balanced trees are used, each membership check has cost $O(\log W(c))$, so the worst-case time complexity is $O(|CGS^{\stackrel{db}{\to}}(c)|(N^2 + N \log W(c)))$.

Both $|CGS^{\stackrel{db}{\to}}(c)|$ and $W(c)$ depend on the rate at which events occur relative to the error between clocks. To simplify the complexity analysis, we introduce τ and ε, with the same meanings as in Section 5.2.1. If $\tau > 2\varepsilon$, then there are $O(3^N E)$ CGSs (as above), and $W(c)$ is $O(3^N)$, so the worst-case time complexity of the algorithm is $O(3^N E N^2)$. If $\tau \leq 2\varepsilon$, then there are $O((\lceil 4\varepsilon/\tau \rceil + 2)^{N-1} E)$ CGSs (as above), and $W(c)$ is $O((\lceil 4\varepsilon/\tau \rceil + 2)^{N-1})$, so the worst-case time complexity is $O((\lceil 4\varepsilon/\tau \rceil + 2)^{N-1} E N^2 \log(\lceil 4\varepsilon/\tau \rceil + 2))$. In both cases, the worst-case time complexity of detecting $\mathbf{Def}^{\stackrel{db}{\to}}$ is linear in E; in contrast, the worst-case time complexity of general algorithms for detecting $\mathbf{Def}^{\stackrel{hb}{\to}}$ is $\Omega(E^N)$.

For a more realistic complexity analysis, we consider the same distribution of inter-event times and the same bound on error between clocks as in the last paragraph of Section 5.2. Of course, the number of CGSs is still characterized by Figures 5 and 6. It is easy to argue that under these assumptions, the expected size of each level is independent of E and depends in the same fashion as the number of CGSs on μ/ε and N. Thus, graphs showing the dependence of $W(c)$ on μ/ε and N would have the same shape as the graphs in Figure 6.

6 Detection Based on a Weak Event Ordering: Inst

The "possibly occurred before" ordering on events is defined by: $e_1 \stackrel{pb}{\to} e_2$ iff $\neg(e_2 \stackrel{db}{\to} e_1)$. Using (3), this induces a relation $\stackrel{pb}{\leadsto}$ on local states, with the interpretation: $s \stackrel{pb}{\leadsto} s'$ if s possibly ended before s' started. Two local states are *strongly concurrent* if they are not related by $\stackrel{pb}{\leadsto}$; such local states must overlap in time. We call elements of $CGS^{\stackrel{pb}{\to}}$ *strongly consistent* global states (SCGSs).[10] For example, Figure 8 shows $\langle CGS^{\stackrel{pb}{\to}}(c_1), \preceq\rangle$; recall that computation c_1 is shown in Figure 1. Note that $\langle CGS^{\stackrel{pb}{\to}}(c_1), \preceq\rangle$ is a total order. More generally, we can show:

Theorem 5. For all computations c, $\langle CGS^{\stackrel{pb}{\to}}(c), \preceq\rangle$ is a total order and therefore a lattice.

[9] In on-line detection, E is not known in advance, so the arrays may need to be resized (*e.g.*, doubled) occasionally. This does not change the asymptotic time complexity.

[10] Fromentin and Raynal call elements of $CGS^{\stackrel{hb}{\to}}$ *inevitable* global states [9].

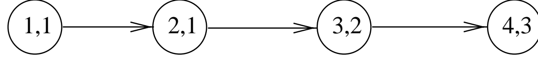

Figure 8: The lattice $\langle CGS^{\stackrel{pb}{\to}}(c_1), \preceq \rangle$.

Proof. Suppose not, *i.e.*, suppose there exist a computation c, two global states $g, g' \in CGS^{\stackrel{pb}{\to}}(c)$, and two processes i and j such that $g(i) \stackrel{pb}{\rightsquigarrow} g'(i)$ and $g'(j) \stackrel{pb}{\rightsquigarrow} g(j)$. By definition of $CGS^{\stackrel{pb}{\to}}(c)$, $\neg(g(i) \stackrel{pb}{\rightsquigarrow} g(j))$, so $C_2(\mathcal{S}(g(j))) < C_1(\mathcal{T}(g(i)))$. By hypothesis, $g(i) \stackrel{pb}{\rightsquigarrow} g'(i)$, so $C_1(\mathcal{T}(g(i))) \leq C_2(\mathcal{S}(g'(i)))$, so by transitivity, $C_2(\mathcal{S}(g(j))) < C_2(\mathcal{S}(g'(i)))$. By definition of $CGS^{\stackrel{pb}{\to}}(c)$, $\neg(g'(j) \stackrel{pb}{\rightsquigarrow} g'(i))$, so $C_2(\mathcal{S}(g'(i))) < C_1(\mathcal{T}(g'(j)))$, so by transitivity, $C_2(\mathcal{S}(g(j))) < C_1(\mathcal{T}(g'(j)))$. By hypothesis, $g'(j) \stackrel{pb}{\rightsquigarrow} g(j)$, so $C_1(\mathcal{T}(g'(j))) \leq C_2(\mathcal{S}(g(j)))$, so by transitivity, $C_2(\mathcal{S}(g(j))) < C_2(\mathcal{S}(g(j)))$, which is a contradiction. ∎

It follows that $\mathbf{Poss}^{\stackrel{pb}{\to}}$ and $\mathbf{Def}^{\stackrel{pb}{\to}}$ are equivalent, *i.e.*, for all computations c and predicates Φ, c satisfies $\mathbf{Poss}^{\stackrel{pb}{\to}}\,\Phi$ iff c satisfies $\mathbf{Def}^{\stackrel{pb}{\to}}\,\Phi$. We define **Inst** ("instantaneously") to denote this modality (*i.e.*, $\mathbf{Poss}^{\stackrel{pb}{\to}}$ and $\mathbf{Def}^{\stackrel{pb}{\to}}$). Informally, a computation satisfies $\mathbf{Inst}\,\Phi$ if there is a global state g satisfying Φ and such that the system definitely passes through g during the computation.

Theorem 1 does not apply to $\stackrel{pb}{\to}$, because:

Lemma 6. $\stackrel{pb}{\to}$ is not a partial ordering.

Proof. Consider the computation in Figure 9. The actual orderings between e_2^2 and e_2^1 and between e_2^2 and e_3^1 cannot be determined from the interval timestamps, so $e_3^1 \stackrel{pb}{\to} e_2^2$ and $e_2^2 \stackrel{pb}{\to} e_2^1$. Since also $e_2^1 \stackrel{pb}{\to} e_3^1$, $\stackrel{pb}{\to}$ contains a cycle. ∎

In light of this, it is not surprising that a minimal increase in $\langle CGS^{\stackrel{pb}{\to}}(c), \preceq \rangle$ does not necessarily correspond to advancing one process by one event. For example, consider the computation c_1 in Figure 1. As shown in Figure 8, two processes advance between the the second and third SCGSs of c_1. In some computations, a minimal increase $\langle CGS^{\stackrel{pb}{\to}}(c), \preceq \rangle$ corresponds to an advance of multiple events per process. Such a computation c_2 is shown in Figure 9. There is no local state of process 2 with which s_2^1 definitely overlaps, so s_2^1 is not part of any SCGS, and process 1 advances by two events between consecutive SCGSs of c_2. Computation c_2 has only two SCGSs: $\langle s_1^1, s_1^2 \rangle$ and $\langle s_3^1, s_2^2 \rangle$.

Since a minimal increase in $\langle CGS^{\stackrel{pb}{\to}}(c), \preceq \rangle$ does not necessarily correspond to advancing one process by one event, the algorithms in Section 5 cannot be adapted easily to detect **Inst**. Our algorithm for detecting **Inst** is based on Fromentin and Raynal's algorithm for detecting **Properly** in asynchronous systems [9, 10]. The definition of **Properly**, generalized to an arbitrary ordering on events, is:

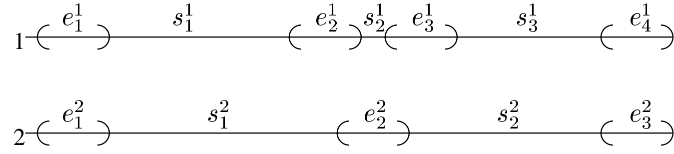

Figure 9: A computation c_2.

Properly: A computation c satisfies $\textbf{Properly}^{\hookrightarrow}\,\Phi$ iff there is a global state satisfying Φ and contained in every path of $\langle CGS^{\hookrightarrow}(c), \preceq\rangle$.

Theorem 7. $\textbf{Properly}^{\stackrel{db}{\rightarrow}}$ is equivalent to **Inst**.

Proof. It suffices to show that a global state g is in $CGS^{\stackrel{pb}{\rightarrow}}(c)$ iff it is contained in every maximal path of $CGS^{\stackrel{db}{\rightarrow}}(c)$. The proof is based on Theorem IGS of [9], which states that a global state g is contained in every maximal path of $\langle CGS^{\stackrel{hb}{\rightarrow}}(c), \preceq\rangle$ iff $(\forall i, j : \mathcal{S}(g(i)) \stackrel{hb}{\rightarrow} \mathcal{T}(g(j)) \vee g(i) = \mathit{last}(c(i)))$, where *last* returns the last element of a sequence. A closely analogous proof shows that a global state g is contained in every maximal path of $\langle CGS^{\stackrel{db}{\rightarrow}}(c), \preceq\rangle$ iff $(\forall i, j : \mathcal{S}(g(i)) \stackrel{db}{\rightarrow} \mathcal{T}(g(j)))$, which by definition of $\stackrel{db}{\rightarrow}$ is equivalent to

$$(\forall i, j : i \neq j \Rightarrow C_2(\mathcal{S}(g(i))) < C_1(\mathcal{T}(g(j)))). \tag{11}$$

The only significant difference involves the last local state of each process. Informally, the disjunct $g(i) = \mathit{last}(c(i))$ is needed in Fromentin and Raynal's analysis because the global state g_f containing the last local state of each process appears in every maximal path of $\langle CGS^{\stackrel{hb}{\rightarrow}}(c), \preceq\rangle$, even though the system might not pass through g_f in real-time, since the processes might terminate at different times. This peculiarity does not arise when real-time timestamps are used, so (11) does not need a disjunct dealing specially with the last local state of each process. Expanding the definition of $CGS^{\stackrel{pb}{\rightarrow}}(c)$ and simplifying yields (11). ∎

As an example of this equivalence, note that $\langle CGS^{\stackrel{pb}{\rightarrow}}(c_1), \preceq\rangle$ in Figure 8 contains exactly the CGSs that are contained in every path of $\langle CGS^{\stackrel{db}{\rightarrow}}(c_1), \preceq\rangle$ in Figure 1.

A straightforward adaptation of Fromentin and Raynal's algorithm for detecting $\textbf{Properly}^{\stackrel{hb}{\rightarrow}}$ yields an algorithm for detecting **Inst** with worst-case time complexity $O(N^3 E)$ (the same as Fromentin and Raynal's algorithm). Optimizations similar to those presented in Section 5.1 reduce this to $O((N \log N)E)$. Expanding the definition of $CGS^{\stackrel{pb}{\rightarrow}}(c)$, a global state g is strongly consistent iff $(\forall i, j : i \neq j \;\Rightarrow\; C_1(\mathcal{T}(g(i))) > C_2(\mathcal{S}(g(j))))$. To check this condition efficiently, we introduce priority queues p_1 and p_2, whose contents are determined by the following invariants:

J1: For each process i such that q_i is non-empty, p_1 contains a record with key $C_1(\mathcal{T}(\text{head}(q_i)))$ and satellite data $\langle i, ptr\rangle$, where ptr is a pointer to the record with satellite data i in p_2. p_1 contains no other records.

J2: For each process i such that q_i is non-empty, p_2 contains a record with key $C_2(\mathcal{S}(\text{head}(q_i)))$ and satellite data i. p_2 contains no other records.

We use p_1 and p_2 to define a function $\text{SC}(p_1, p_2)$ that efficiently tests whether the heads of the non-empty queues are pairwise strongly concurrent. Taking into account the possibility that $C_1(\mathcal{T}(g(i))) < C_2(\mathcal{S}(g(i)))$ for some i, we obtain

$$\begin{aligned} \text{SC}(p_1, p_2) \;\triangleq\;& \text{empty}(p_1) \\ & \lor \text{key}(\text{getMin}(p_1)) > \text{key}(\text{getMax}(p_2)) \\ & \lor (\pi_1(\text{data}(\text{getMin}(p_1))) = \text{data}(\text{getMax}(p_2)) \land \text{countMax}(p_2) = 1), \end{aligned} \tag{12}$$

where $\text{countMax}(p)$ is the number of records containing the maximal value of the key in priority queue p, and $\text{data}(\langle k, d\rangle) = d$ and $\pi_1(\langle i, ptr\rangle) = i$. Thus, the following procedure *makeSC* ("make Strongly Concurrent") loops until the heads of the non-empty queues are strongly concurrent:

```
procedure makeSC()
  while ¬SC(p1, p2)
    ⟨k, ⟨i, ptr⟩⟩ := extractMin(p1);
    remove record for i (i.e., record *ptr) from p2;
    removeHead(qi);
    if ¬empty(qi) then
      add records for i to p1 and p2 to maintain invariants J1 and J2;
    endif
  endwhile
```

where $\text{extractMin}(p)$ is like getMin except that it removes the record it returns.

The optimized algorithm for detecting **Inst** appears in Figure 10, where $\text{head2}(q)$ returns the second element of a queue q. When a SCGS g is found, if g does not satisfy Φ, then the monitor starts searching for the next SCGS by advancing some process j such that this advance yields a CGS (*i.e.*, an element of $CGS^{\overset{db}{\to}}$). If at first no process can be so advanced (*e.g.*, if each queue q_i contains only one element), then the monitor waits for more local states to be reported. It follows from the definitions of $CGS^{\overset{db}{\to}}$ and $CGS^{\overset{pb}{\to}}$ that, if advancing some process yields a CGS, then advancing some process j such that $C_1(\mathcal{S}(\text{head2}(q_j)))$ is minimal yields a CGS. Thus, we reduce the time needed to find such a process j by maintaining a priority queue p_3 satisfying the invariant:

J3: For each process i such that q_i is non-empty, p_3 contains a record with key $C_1(\mathcal{S}(\text{head2}(q_j)))$ and satellite data i. p_3 contains no other records.

Thus, in line (†), it suffices to take $j = \text{head}(p_3)$. Testing whether $g[j \mapsto \text{head2}(q_j)]$ is consistent can be done in $O(\log N)$ time by temporarily updating p_1 and p_2 as if process j had been advanced and then using (9).

We analyze the worst-case time complexity by summing the times spent inside and outside of *makeSC*. Each iteration of the **while** loop in *makeSC* takes $O(\log N)$ time (because each operation on priority queues takes $O(\log N)$ time) and removes one local state. The computation contains $O(NE)$ local states, so the total time spent inside *makeSC* is $O((N \log N)E)$. The total time spent in the code outside *makeSC* is also $O((N \log N)E)$, since there are $O(NE)$ SCGSs (this is a corollary of Theorem 7), and each local state is considered at most once and at $O(\log N)$ cost in the **wait** statement. Thus, the worst-case time complexity of the algorithm is $O((N \log N)E)$.

```
On receiving x from process i:
append(q_i, x);
if head(q_i) = x then
   add records for i to p_1 and p_2 to maintain invariants J1 and J2;
   found := true;
   while found
      makeSC();
      if (∃ i : empty(q_i)) then
         found := false
      else (* found a SCGS *)
         g := the global state (λi. head(q_i));        (* g is a SCGS *)
         if g satisfies Φ then
            return g
         else
            wait until there exists j such that g[j ↦ head2(q_j)] is in CGS^{db→}(c);        (†)
            remove records for j from p_1 and p_2;
            removeHead(q_j);
            add records for j to p_1 and p_2 to maintain invariants J1 and J2
         endif
      endif
   endwhile
endif
```

Figure 10: Algorithm for detecting **Inst** Φ.

7 Sample Applications

7.1 Coherence Protocols

Coherence of shared data is a central issue in many distributed systems, including distributed file systems, distributed shared memory, and distributed databases. A typical invariant maintained by a coherence protocol is:

> *cohrnt*: if one machine has a copy of a data item in write mode, then no other machine has a valid copy of that data item.

As part of testing and debugging a coherence protocol, a monitor might be used to issue a warning if $\mathbf{Poss}^{\overset{db}{\rightarrow}} \neg cohrnt$ is detected and report an error if $\mathbf{Def}^{\overset{db}{\rightarrow}} \neg cohrnt$ is detected. A computationally cheaper but sometimes less informative alternative is to monitor only $\mathbf{Inst}\, \neg cohrnt$ and report an error if it is detected.

A detection algorithm based on happened-before could be used instead, if the system can be modified to maintain vector clocks (or for some reason maintains them already). However, if the coherence protocol uses timers, then time acts as a hidden channel [2] (*i.e.*, a means of communication other than messages), so detection based on happened-before might yield less accurate results. Timers can be used in coherence protocols to:

- *obtain* a lock, by broadcasting a request for a lock and, if no conflicting announcement is received within an appropriate time interval, granting oneself the lock.
- *release* a lock, by associating a finite lifetime with the lock; such a lock is called a *lease* [15]. When the lifetime expires, all processes know (without further communication) that the lock has been released.

For example, the resource allocation algorithm in [19, Section 5.1] uses timers in both of these ways. These techniques use timers (instead of messages) for synchronization, so detection based on happened-before is less appropriate. For example, release of a lock by one process and acquisition of that lock by another process need not be related by happened-before, so $\mathbf{Poss}^{\overset{hb}{\rightarrow}} \neg cohrnt$ may be detected even when coherence was maintained and $\mathbf{Poss}^{\overset{db}{\rightarrow}} \neg cohrnt$ would not be detected.

Clock-based monitoring is useful even for coherence protocols that provide weaker guarantees than *cohrnt*. For example, in the Sun Network File System (NFS) [29, Section 17.6.2], file information is cached. Timers are used to limit staleness: if the cached information is needed again after the timer expires, the client asks the server whether the cached information is still valid. Since this lock-free approach does not enforce the one-copy file-sharing semantics that is traditional in UNIX, it is useful to monitor the system to detect how often

violations of one-copy semantics are seen by applications. For example, the detection algorithm for **Inst** can easily be adapted to count (instead of just detect) SCGSs satisfying the predicate: some process is reading cached information and some other cache contains a more recent version of that information. In NFS, the lifetime of cached data is typically tens of seconds, which is three orders of magnitude larger than typical clock synchronization error in a LAN, so this approach should detect most violations of one-copy semantics.

7.2 Concurrency Control for Distributed Transactions

Leases can also be used for concurrency control in distributed database systems, to reduce the number of messages needed to commit read-only transactions, as described in [22, Section 7]. The idea is that a read-only transaction acquires leases as it uses data objects. If the transaction completes before any of those leases expires, then the coordinator commits the transaction, without further communication. As part of testing and debugging such a system, one might use a monitor to detect violations of the invariant: when a transaction commits, it holds locks on all of the objects it uses. Since commit events and expirations of leases may be unrelated by happened-before, detection based on $\mathbf{Poss}^{\stackrel{hb}{\rightarrow}}$ or $\mathbf{Def}^{\stackrel{hb}{\rightarrow}}$ may report violations even when no violation occurred and detection based on real-time clocks would not report a violation.

References

[1] Sridhar Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. In *Proc. International Conference on Parallel and Distributed Systems*, pages 412–417, December 1994. An expanded version is available from the authors.

[2] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison Wesley, 2nd edition, 1993.

[3] Özalp Babaoğlu and Michel Raynal. Specification and verification of behavioral patterns in distributed computations. In *Fourth International Working Conference on Dependable Computing for Critical Applications*, 1994.

[4] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, January 1995.

[5] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):169–189, 1998.

[6] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991. Appeared as ACM SIGPLAN Notices 26(12):167-174, December 1991.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[8] Claire Diehl, Claude Jard, and Jean-Xavier Rampon. Reachability analysis on distributed executions. In J.-P. Jouannaud and M.-C. Gaudel, editors, *TAPSOFT '93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 629–643. Springer-Verlag, 1993.

[9] Eddy Fromentin and Michel Raynal. Inevitable global states: a concept to detect unstable properties of distributed computations in an observer independent way. In *Proc. Sixth IEEE Symposium on Parallel and Distributed Processing*, 1994.

[10] Eddy Fromentin and Michel Raynal. Characterizing and detecting the set of global states seen by all observers of a distributed computation. In *Proc. IEEE 15th International Conference on Distributed Computing Systems (ICDCS)*, 1995.

[11] Eddy Fromentin, Michel Raynal, Vijay K. Garg, and Alex Tomlinson. On-the-fly testing of regular patterns in distributed computations. In *Proc. 23rd International Conference on Parallel Processing*, volume 2, pages 73–76, August 1994.

[12] Vijay Garg, Craig Chase, Richard Kilgore, and J. Roger Mitchell. Efficient detection of channel predicates in distributed systems. *Journal of Parallel and Distributed Computing*, 45(2):134–147, September 1997.

[13] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[14] Vijay K. Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, 1996.

[15] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file system consistency. In *Proc. 11th ACM Symposium on Operating System Principles*, pages 202–210, 1989.

[16] Michel Hurfin, Noël Plouzeau, and Michel Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[17] R. Jegou, R. Medina, and L. Nourine. Linear space algorithm for on-line detection of global predicates. In J. Desel, editor, *Proc. International Workshop on Structures in Concurrency Theory (STRICT '95)*. Springer-Verlag, 1995.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[19] Leslie Lamport. Using time instead of timeout in fault-tolerant systems. *ACM Transactions on Programming Languages and Systems*, 6(2):256–280, April 1984.

[20] Leslie Lamport. The mutual exclusion problem: Part i—a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[21] Leslie Lamport. On interprocess communication: Part 1. *Distributed Computing*, 1:76–101, 1986.

[22] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proc. Tenth ACM Symposium on Principles of Distributed Computing*, pages 1–9. ACM Press, August 1991.

[23] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proc. 5th Int'l. Workshop on Distributed Algorithms (WDAG '91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.

[24] David L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Communications*, 39(10):1482–1493, October 1991.

[25] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. DARPA Network Working Group Report RFC-1305, University of Delaware, March 1992.

[26] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1995.

[27] OSF. *Introduction to OSF DCE*. Prentice-Hall, 1992.

[28] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[29] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison Wesley, 4th edition, 1994.

[30] Scott D. Stoller. Detecting global predicates in distributed systems with clocks. In Marios Mavronikolas, editor, *Proc. 11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 185–199. Springer-Verlag, 1997.

[31] Scott D. Stoller and Yanhong A. Liu. Efficient symbolic detection of global properties in distributed systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1998.

[32] Scott D. Stoller and Fred B. Schneider. Faster possibility detection by combining two approaches. In Jean-Michel Hélary and Michel Raynal, editors, *Proc. 9th International Workshop on Distributed Algorithms (WDAG '95)*, volume 972 of *Lecture Notes in Computer Science*, pages 318–332. Springer-Verlag, September 1995.

[33] Paulo Veríssimo. Real-time communication. In Sape Mullender, editor, *Distributed Systems*, chapter 17, pages 447–490. Addison Wesley, 2nd edition, 1993.

Appendix

Proof of Theorem 3. It follows immediately from the definitions that $\stackrel{db}{\to}$ is process-wise-total. We need to show that $\stackrel{db}{\to}$ is irreflexive, acyclic, and transitive. Irreflexivity is obvious. For transitivity, we suppose $e_1 \stackrel{db}{\to} e_2$ and $e_2 \stackrel{db}{\to} e_3$, and show $e_1 \stackrel{db}{\to} e_3$. First consider the case $pr(e_1) = pr(e_3)$. In this case, it suffices to show that e_1 occurred before e_3. If $pr(e_2) = pr(e_1)$, then the desired result follows from transitivity of "occurred before". If $pr(e_2) \neq pr(e_1)$, then using TS1, the hypothesis $e_1 \stackrel{db}{\to} e_2$, TS1 again, and finally the hypothesis $e_2 \stackrel{db}{\to} e_3$, we have the chain of inequalities $C_1(e_1) \leq C_2(e_1) < C_1(e_2) \leq C_2(e_2) < C_1(e_3)$, so $C_1(e_1) < C_1(e_3)$, so by TS2, e_1 occurred before e_3. Next consider the case $pr(e_1) \neq pr(e_3)$. Note that $\neg(pr(e_1) = pr(e_2) \wedge pr(e_2) = pr(e_3))$. If $pr(e_2) \neq pr(e_1)$, then it is easy to show (by case analysis on whether $pr(e_2) = pr(e_3)$) that $C_2(e_1) < C_1(e_2) \leq C_1(e_3)$, so $C_2(e_1) < C_1(e_3)$, as desired. If $pr(e_2) \neq pr(e_3)$, then it is easy to show (by case analysis on whether $pr(e_2) = pr(e_1)$) that $C_2(e_1) \leq C_2(e_2) < C_1(e_3)$, so $C_2(e_1) < C_1(e_3)$, as desired.

Given transitivity, to conclude acyclicity, it suffices to show that there are no cycles of size 2. We suppose $e_1 \stackrel{db}{\to} e_2$ and $e_2 \stackrel{db}{\to} e_1$, and derive a contradiction. If $pr(e_1) = pr(e_2)$, then the fact that "occurred before" is a total order on the events of each process yields the desired contradiction. If $pr(e_1) \neq pr(e_2)$, then using TS1, the hypothesis $e_1 \stackrel{db}{\to} e_2$, TS1 again, and finally the hypothesis $e_2 \stackrel{db}{\to} e_1$, we obtain the chain of inequalities $C_1(e_1) \leq C_2(e_1) < C_1(e_2) \leq C_2(e_2) < C_1(e_1)$, which implies $C_1(e_1) < C_1(e_1)$, a contradiction. ∎