Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail[1]

Reinhard Schwarz
Department of Computer Science, University of Kaiserslautern,
P.O. Box 3049, D - 6750 Kaiserslautern, Germany
schwarz@informatik.uni-kl.de

Friedemann Mattern
Department of Computer Science, University of Saarland
Im Stadtwald 36, D - 6600 Saarbrücken, Germany
mattern@cs.uni-sb.de

Report SFB 124 - 15 / 92, December 1992

Abstract*: The paper shows that characterizing the causal relationship between significant events is an important but non-trivial aspect for understanding the behavior of distributed programs. An introduction to the notion of causality and its relation to logical time is given; some fundamental results concerning the characterization of causality are presented. Recent work on the detection of causal relationships in distributed computations is surveyed. The relative merits and limitations of the different approaches are discussed, and their general feasibility is analyzed.*

Keywords: *Distributed Computation, Causality, Distributed System, Causal Ordering, Logical Time, Vector Time, Global Predicate Detection, Distributed Debugging*

1 Introduction

Today, distributed and parallel systems are generally available, and their technology has reached a certain degree of maturity. Unfortunately, we still lack complete understanding of how to design, realize, and test the software for such systems, although substantial research effort has been spent on this topic. It seems that implementing distributed programs is still an art rather than an engineering issue; understanding the behavior of a distributed program remains to be a challenge. One of the main reasons for this is the nondeterminism that is inherent to such programs; in particular, it is notoriously difficult to keep track of the various local activities that happen concurrently and may (or may not) interact in a way which is difficult to predict — leading to, for instance, potential synchronization errors or deadlocks.

For a proper understanding of a distributed program and its execution, it is important to determine the causal and temporal relationship between the events that occur in its computation. For example, it is often the case that two concurrent events may occur in any order, possibly yielding different results in each case. This shows that nondeterminism is closely related to concurrency. In fact, the effects of concurrency and nondeterminism play an important role in the process of analyzing, monitoring, debugging, and visualizing the behavior of a distributed system.

Distributed systems are loosely coupled in the sense that the relative speed of their local activities is usually not known in advance; execution times and message delays may vary substantially for several repetitions of the same algorithm. Furthermore, a global system clock or perfectly synchronized local

1. This is a revised version of report 215 / 91, Dep. of Computer Science, University of Kaiserslautern

clocks are generally not available. Thus, it is difficult to identify concurrent activities in distributed computations. In this paper, we show how the notion of concurrency can be based on the *causality relation* between events. The characterization and efficient representation of this relation is a non-trivial problem. In the sequel, we survey several approaches for the analysis of the causality relation and related concepts such as *logical time* or *global predicates* which are crucial for the understanding of distributed computations.

1.1 System Model: Events, Time Diagrams, and Causality

We use a widely accepted model where a *distributed system* consists of N sequential (i.e., single-threaded) processes $P_1, \ldots, P_N$ communicating solely by messages[2]. The local states of all processes are assumed to be disjoint, i.e., processes do not share common memory. The behavior of each process consists of local state changes, and of the sending of messages to other processes; these actions are completely determined by a local algorithm which also determines the reaction to incoming messages. The concurrent and coordinated execution of all local algorithms forms a *distributed computation*. For the rest of this paper, we assume that communication between processes is point-to-point, and that message transfer may suffer from arbitrary non-zero delays. We do not assume FIFO order of message delivery unless explicitly stated. Furthermore, we do not assume the availability of a global clock or perfectly synchronized local clocks.

The occurrence of actions performed by the local algorithms are called *events*. From an abstract point of view, a distributed computation can be described by the types and relative order of events occurring in each process. Let E_i denote the set of events occurring in process P_i, and let $E = E_1 \cup \ldots \cup E_N$ denote the set of all events of the distributed computation. These event sets are evolving dynamically during the computation; they can be obtained by collecting traces issued by the running processes. As we assume that each P_i is strictly sequential, the events in E_i are totally ordered by the sequence of their occurrence. Thus, it is convenient to index the events of a process P_i in the order in which they occur: $E_i = \{e_{i1}, e_{i2}, e_{i3} \ldots\}$. We will refer to this occurrence order as the *standard enumeration* of E_i.

For our purposes, it suffices to distinguish between three kinds of events: *send* events, *receive* events, and *internal* events. A send event causes a message to be sent; a receive event causes a message to be received and the local state to be updated according to the contents of that message. A send event and a receive event are said to *correspond* if the same message that was sent in the send event is received in the receive event. We assume that a send event and its corresponding receive event occur in different processes. *Internal* events affect only the local process state. Events are assumed to be atomic. Thus, we do not have to bother with events that are simultaneous in real time, and an event can safely be modelled as having a zero duration.

It should be noted that our model does not explicitly deal with *conflicts*, as is common practice in Petri net theory or related concurrency theories [Nie81, Win88, Pra91]. This does, however, not imply that the local algorithms are required to work deterministically, i.e., that the possibility of conflicts is excluded. Our discussion merely refers to computations which have actually occurred (i.e., so-called single *runs* or *executions*); we do not discuss methods for the specification of *possible* runs. Thus, our model of computation does not preclude constructs such as CSP-like *guarded commands*, or nondeterministic message *select* statements in the underlying programming or specification language.

A convenient way to visualize distributed computations are *time diagrams*. Figure 1 shows an example for a computation comprising three processes. The passage of time is depicted as a directed *process line* for each process. Events are symbolized by dots on the process lines, according to their relative order

2. A fixed number of processes is assumed mainly for notational convenience; a generalization of our model to a dynamically changing set of processes is straightforward. One could, for example, model dynamically created (or destroyed) processes as being silently present throughout the computation, producing events only during their actual lifetime. Creating a new process would then correspond to sending an activation message to a process already kept in reserve.

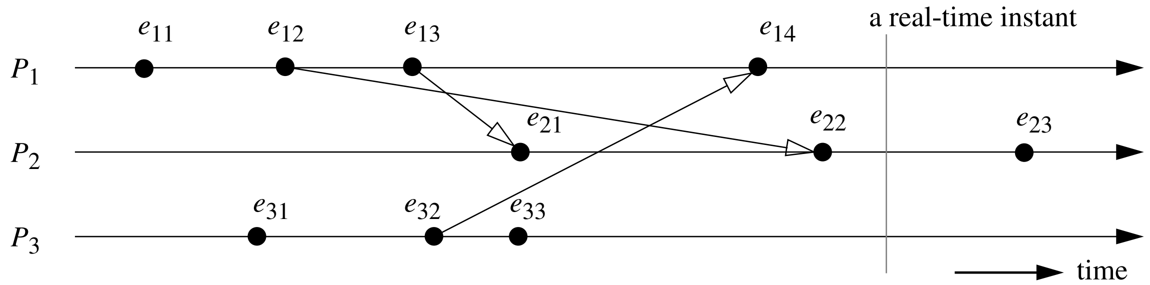

Figure 1: A time diagram of a distributed computation

of occurrence. Messages are depicted as arrows connecting send events with their corresponding receive events.

By examining time diagrams like Figure 1, it becomes obvious that an event e may *causally affect* another event e' if and only if there is a directed left-to-right path in the time diagram starting at e and ending at e'. Thus, event e_{11} may affect events e_{12}, e_{13}, and e_{14} which are local relative to e_{11}, and also non-local events like e_{21} or e_{23}. On the other hand, event e_{12} can neither influence event e_{11} occurring earlier on the same process line, nor can it affect non-local events like e_{31} or e_{33}. We can formalize this observation by defining the *causality relation* as follows:

Definition 1.1 *Given the standard enumeration of E_i, the* causality relation $\rightarrow \subseteq E \times E$ *is the smallest transitive relation satisfying:*

(1) If e_{ij}, $e_{ik} \in E_i$ occur in the same process P_i, and $j < k$, then $e_{ij} \rightarrow e_{ik}$.

(2) If $s \in E_i$ is a send event and $r \in E_j$ is the corresponding receive event, then $s \rightarrow r$.

Note that $\rightarrow$ is irreflexive, asymmetric, and transitive; i.e., it is a *strict partial order.* By definition, the causality relation extends the partial order defined by the standard enumeration of E_1, E_2,…, and E_N. Informally, our reasoning about the causal relationship between events can be stated in terms of the causality relation as follows: An event e may causally affect another event e' if and only if $e \rightarrow e'$.

The causality relation $\rightarrow$ of Definition 1.1 is actually identical to the “happened before” relation defined by Lamport in [Lam78]. We prefer to use the term “causality” rather than “happened before” because the relation defined in Definition 1.1 is causal rather than temporal. For example, event e_{33} in Figure 1 occurs at a later real-time instant than event e_{11}, although they are not causally related.

If, for two events e and e', neither $e \rightarrow e'$, nor $e' \rightarrow e$ holds, then neither of them causally affects the other. As we assume that there is no global real-time clock available, there is no way to decide which of the events e and e' took place first “in reality” — we do not know their absolute order. This motivates the following definition of *concurrency*:

Definition 1.2 *The* concurrency relation $\| \subseteq E \times E$ *is defined as*

$$e \parallel e' \quad \textit{iff} \quad \neg\,(e \rightarrow e') \text{ and } \neg\,(e' \rightarrow e).$$

If $e \parallel e'$ holds, e and e' are said to be concurrent.

Observation 1.3 *The concurrency relation is not transitive.*

For example, in Figure 1 $e_{12} \parallel e_{31}$ and $e_{31} \parallel e_{22}$ hold, but obviously e_{12} and e_{22} are not concurrent.

1.2 The Significance of the Causality Relation

Causality is fundamental to many problems occurring in distributed computing. For example, determining a *consistent global snapshot* of a distributed computation [ChL85, FoZ90, Mat93] essentially requires to find a set of local snapshots such that the causal relation between all events that are included in the snapshots is respected in the following sense: if e' is contained in the global snapshot formed by the union of the local snapshots, and $e \rightarrow e'$ holds, then e has to be included in the global snapshot, too. That is, consistent snapshots are subsets of E that are left-closed with respect to the causality relation $\rightarrow$.

Thus, the notion of consistency in distributed systems is basically an issue of correctly reflecting causality.

Causal consistency has many important applications. For example, determining consistent recovery points is a well-known problem in the field of distributed database management [StY85]. For determining deadlocks or detecting the termination of a distributed computation [Mat87], the global view of the computation state must also be causally consistent in order to prevent so-called phantom deadlocks and false termination states. In distributed debugging, detecting global predicates is a key issue, and the causality relation is of utmost importance [CoM91, HaW88, HsK90, MaN91]. Again, the problem is to obtain a consistent view in order to correctly evaluate the global predicate. Analyzing the causal relationship between events is also helpful for the detection of race conditions and other synchronization errors — one of the most difficult problems in distributed programming. Another issue is the proper visualization of concurrent activities in distributed system monitoring [Sto87]. Here, the causal relation determines the sequence in which events must be visualized so that cause and effect appear in the correct order.

Causality plays also an important role in the exploitation of maximum parallelism, i.e., for distributed applications which are required to run "as asynchronous as possible". An analysis of the causality relation can therefore serve as an abstract concurrency measure of an algorithm [Cha89, Fid91]. Note that all events which are not causally related can be executed in parallel — at least in principle. Hence, a careful study of causality could yield the "optimal parallelization" of a given set of events, and comparing this with a sequential ordering may lead to a formal definition of the "inherent degree of parallelism" of the underlying computation.

In distributed protocols, the relaxation of unnecessary synchronization constraints may permit a higher degree of concurrency; a minimum requirement for synchronization is that the causal order of events is respected. Communication protocols for point-to-point or multicast communications which enforce only a causal delivery order (instead of insisting on synchronous delivery) are based on this idea [BiJ87, Sch89]. Here, different communication activities can proceed in parallel, only the delivery of messages has to be delayed according to causality constraints. For multicast operations, this technique was successfully employed in the ISIS system [BiJ87, Bir91]. Causally ordered broadcast protocols are useful, for example, for the realization of fault tolerant systems [Bir91b]. A similar idea is used in the implementation of "causal shared memory" [Aha90, HuA90], a weak form of shared virtual memory.

In the theory of distributed computing, causality has also been used for reasoning about the properties of asynchronous systems. In [PaT92], for example, it is argued that in many cases causality can serve as a more appropriate substitute for the traditional notion of real-time, and that reasoning based on the causal rather than on the temporal structure of a system is the correct level of abstraction in a distributed setting. This view is now also shared by most researchers working on distributed operating systems as a recent debate among experts shows [Ren92].

2 Causal History and Vector Time

In this section, we aim at a practical method to determine the causal relationship between events. We start with an easy-to-understand, but rather impracticable approach by assigning a complete *causal history* to each event, and we show that these histories accurately characterize causality. Some refinements of the basic scheme will finally lead to a more practical concept generally known as *vector time*.

2.1 Causal Histories

In principle, we can determine causal relationships by assigning to each event e its *causal history* $C(e)$, where $C(e)$ is a set of events defined as follows:

Definition 2.1 *Let $E = E_1 \cup \ldots \cup E_N$ denote the set of events of a distributed computation, and let $e \in E$ denote an event occurring in the course of that computation. The* causal history *of e, denoted C(e),* *is defined as* $C(e) = \{e' \in E \mid (e' \rightarrow e)\} \cup \{e\}$.

The projection *of* $C(e)$ *on* E_i, *denoted* $C(e)[i]$, *is defined by* $C(e)[i] = \{e' \in C(e) \mid e' \in E_i\}$.

A causal history is a prefix-closed set of events under the causal ordering. $C(e)$ contains all events which causally precede e, i.e., which might have affected e. Note that e' causally precedes e if and only if there is a directed path in the time diagram from e' to e. Thus, $C(e)$ essentially contains those events that can reach e along a directed path. For example, event e_{23} in Figure 1 is reachable by e_{11}, e_{12}, e_{13}, e_{21}, and e_{22}; hence, $C(e_{23}) = \{e_{11}, e_{12}, e_{13}, e_{21}, e_{22}, e_{23}\}$. A discussion of further interesting properties of causal histories may be found in [Nie81, Win88, Pra92].

Lemma 2.2 *Let* $e, e' \in E$, $e \neq e'$. *Causality and causal history are related as follows:*

(1) $e \rightarrow e'$ *iff* $e \in C(e')$.

(2) $e \parallel e'$ *iff* $e \notin C(e') \wedge e' \notin C(e)$.

Proof. This follows directly from the definition of $C(e)$. ❑

Lemma 2.2 states that the causal histories $C(e)$ and $C(e')$ suffice to determine causality or concurrency of two events e and e'. Furthermore, there is a straightforward algorithm that assigns $C(e)$ to every event e of a distributed computation:

(1) Let $E_i = \{e_{i1}, e_{i2}, \ldots, e_{ik}\}$ denote the local events of P_i in standard enumeration, and define dummy events e_{i0} for $i = 1, \ldots, N$ such that $C(e_{i0}) = \varnothing$.

(2) If $e_{ij} \in E_i$ is an internal event or a send event, and $e_{i,j-1} \in E_i$ is its local predecessor, then compute $C(e_{ij})$ as follows: $C(e_{ij}) := C(e_{i,j-1}) \cup \{e_{ij}\}$.
Informally, e_{ij} simply inherits the causal history of its immediate predecessor.

(3) If $e_{ij} \in E_i$ is a receive event, s its corresponding send event, and $e_{i,j-1} \in E_i$ is the local predecessor of e_{ij}, then compute $C(e_{ij})$ as follows: $C(e_{ij}) := C(e_{i,j-1}) \cup C(s) \cup \{e_{ij}\}$.
Informally, e_{ij} inherits the causal history of both of its immediate predecessors.

2.2 Vector Time

The scheme described above allows to determine causal histories on-the-fly during a distributed computation by maintaining sets of events at the processes and by piggybacking $C(s)$ on the outgoing message for each send event s. However, the algorithm is only of theoretical interest, because the size of the causal history sets is of the order of the total number of events that occur during the computation. Fortunately, the basic scheme can be improved substantially based on the following observation:

Observation 2.3 *Recall that* $C(e) = C(e)[1] \cup \ldots \cup C(e)[N]$. *If* $E_k = \{e_{k1}, \ldots, e_{km}\}$ *is given in standard enumeration, then* $e_{kj} \in C(e)[k]$ *implies that* $e_{k1}, \ldots, e_{k,j-1} \in C(e)[k]$. *Therefore, for each k the set* $C(e)[k]$ *is sufficiently characterized by the largest index among its members, i.e., its cardinality. Thus,* $C(e)$ *can be uniquely represented by an N-dimensional vector* $V(e)$ *of cardinal numbers, where* $V(e)[k] = |\, C(e)[k] \,|$ *holds for the k-th component* ($k = 1, \ldots, N$) *of vector* $V(e)$[3].

As an example, the causal history of event e_{23} in Figure 1 can be represented by $V(e_{23}) = [3, 3, 0]$ because the cardinality of $C(e_{23})[1]$, $C(e_{23})[2]$, and $C(e_{23})[3]$ is 3, 3, and 0, respectively. Figure 2 depicts a distributed computation, with the associated vectors assigned to each event.

The use of vectors can be generalized in a straightforward way to represent an arbitrary prefix-closed event set $X \subseteq E$, again by taking the locally largest event index: $V(X)[k] = |\, X \cap E_k \,|$. For notational convenience, let the *supremum* $\sup\{v_1, \ldots, v_m\}$ of a set $\{v_1, \ldots, v_m\}$ of n-dimensional vectors denote the vector v defined as $v[i] = \max\{v_1[i], \ldots, v_m[i]\}$ for $i = 1, \ldots, n$. The following lemma is the key to an efficient implementation of the above algorithm:

Lemma 2.4 *Let* $e, e' \in E$ *denote events, let* $C(e)$, $C(e')$ *denote their causal histories, and let* $V(e)$, $V(e')$ *denote the corresponding vector representations, respectively. The vector representation of the*

---

3. If the number of processes N is not statically fixed, then $V(e)$ can be represented by the set of all those pairs $(k, |C(e)[k]|)$ for which the second component is different from 0.

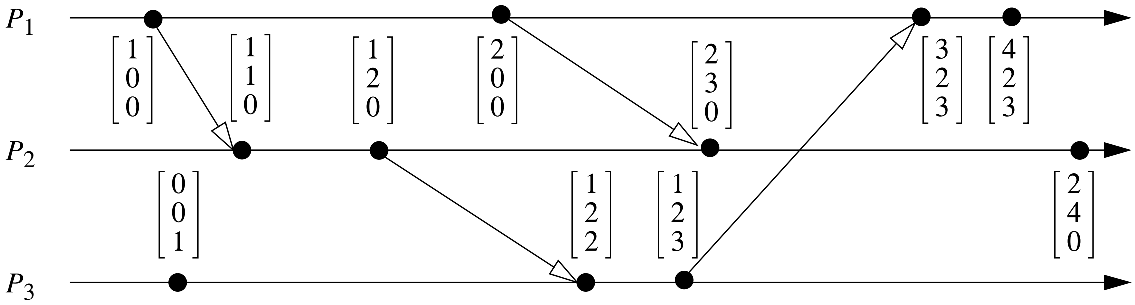

Figure 2: Events of a distributed computation with their associated vector timestamps

union $C(e) \cup C(e')$ is $V(C(e) \cup C(e')) = sup\{V(e), V(e')\}$.

Proof. This follows immediately from the definition of $C(e)$, $V(e)$, and Observation 2.3. ❑

Applying Lemma 2.4, and translating the set operations on causal histories to the corresponding operations on vectors yields an improved version of our above-mentioned algorithm which maintains vectors instead of sets. In fact, the resulting algorithm is essentially the same as the one given in [Fid88] or in [Mat89]. There, the vectors defined as in Observation 2.3 are called *time vectors*, and the general concept is called *vector time*[4]. We state the operational definition from [Mat89] here:

Definition 2.5 *Let $P_1,\ldots, P_N$ denote the processes of a distributed computation. The* vector time *V_i of process P_i is maintained according to the following rules:*

(1) Initially, $V_i[k] := 0$ for $k = 1,\ldots, N$.

(2) On each internal event e, process P_i increments V_i as follows: $V_i[i] := V_i[i] + 1$.

(3) On sending message m, P_i updates V_i as in (2), and attaches the new vector to m.

(4) On receiving a message m with attached vector time $V(m)$, P_i increments V_i as in (2). Next, P_i updates its current V_i as follows: $V_i := sup\{V_i, V(m)\}$.

Let $V(e)$ denote the vector time V_i which results from the occurrence of event e in process P_i. $V(e)$ is said to be the vector timestamp *of event e. Accordingly, $V(m)$ denotes the vector timestamp attached to message m.*

It should be clear that the rules of Definition 2.5 specify a simplified version of the above-mentioned algorithm for the computation of the causal history $C(e)$ of an event e. Instead of $C(e)$, the associated vector $V(e)$ is determined according to Observation 2.3. Obviously, this version of the algorithm can be realized more efficiently than the original scheme that manipulates sets of events. The application of Definition 2.5 is demonstrated in Figure 2 which illustrates that timestamps of messages propagate the knowledge about vector time (and thus about causally preceding events) along the directed paths of a time diagram. Since a simple one-to-one correspondence between vector time $V(e)$ and causal history $C(e)$ exists for all $e \in E$, we can determine causal relationships solely by analyzing the vector timestamps of the events in question.

We conclude our discussion of vector time with a "knowledge-based" interpretation. Informally, the component $V_i[i]$ of P_i's current vector time reflects the accurate logical time at P_i (measured in "number of past events" at P_i), while $V_i[k]$ is the best estimate P_i was able to derive about P_k's current logical clock value $V_k[k]$. Thus, if $V(e)$ is the vector timestamp of an event occurring in P_i, then $V(e)[k]$ is the number of events in P_k which e "knows about", where "x knows about y" is synonymous to "y is in the

4. Actually, the concept of vector time cannot be attributed to a single person. Several authors "re-invented" time vectors for their purposes, with different motivation, and often without knowing of each other. To the best of our knowledge, the first applications of "dependency tracking" vectors appeared in the early 80's in the field of distributed database management [FiM82, WuB84, StY85]. In [Fid88] and [Mat89], however, vector time is introduced as a generalization of Lamport's logical time, and its mathematical structure and its general properties are analyzed.

causal history of x".

Postulating an idealized global observer who is instantaneously informed about the occurrence of all events yields another interesting interpretation of vector time. This "omniscient" observer could maintain a vector clock Ω defined as $\Omega[i] = V_i[i]$ (for $i = 1, \ldots, N$), thus having perfect knowledge of all past events at any time. Clearly, P_i's "current knowledge" about each process (represented by V_i) is only a subset of the omniscient observer's knowledge since $\Omega = \sup\{V_1, \ldots, V_N\}$. However, it should also be evident that at any time V_i represents the best possible "approximation" of global knowledge that P_i is able to obtain *within* the system.

3 Causality and Time

In the previous section, we introduced the causality relation $\rightarrow$ and tried to characterize this relation by assigning a causal history to each event; we finally obtained an operational definition of vector time, and we gave an informal interpretation of our result. In this section, we further investigate the properties of vector time, and analyze its relation to real time and causality.

3.1 Characterizing Causality with Vector Time

Vector time has several interesting properties, for example, its mathematical structure is similar to Minkowski's relativistic space-time in the sense that causal histories correspond to light cones [Mat89]. Most interestingly, however, the structure of vector time is isomorphic to the causality structure of the underlying distributed computation. In this section, we proof this fact by rephrasing Lemma 2.2 in terms of time vectors.

Definition 3.1 *Let E denote the set of events of a distributed computation, and let (S, <) denote an arbitrary partially ordered set. Let ϕ: E → S denote a mapping.*

(1) (ϕ, <) is said to be consistent with causality, *if for all e, e' ∈ E*

$$\phi(e) < \phi(e') \quad \text{if} \quad e \rightarrow e'.$$

(2) (ϕ, <) is said to characterize causality, *if for all e, e' ∈ E*

$$\phi(e) < \phi(e') \quad \text{iff} \quad e \rightarrow e'.$$

For a given ϕ which satisfies (1) or (2) we say for short that (S, <) is consistent with or characterizes causality.

Note that a partial order (E, <) on the set E of events which is consistent with causality represents an *extension* of (E, $\rightarrow$) (in particular, a *linear extension* if < is a total order), and that any partial order (S, <) which characterizes causality represents an isomorphical *embedding* of (E, $\rightarrow$).

Let $V = \{V(e) \mid e \in E\}$ denote the set of vector time values assigned to the events of a distributed computation according to Definition 2.5. We aim at a computationally simple relation < defined on time vectors, such that (V, <) characterizes causality.

Definition 3.2 *Let u, v denote time vectors of dimension m.*

(1) $u \leq v$ iff $u[k] \leq v[k]$ for $k = 1, \ldots, m$.

(2) $u < v$ iff $u \leq v$ and $u \neq v$.

(3) $u \parallel v$ iff $\neg (u < v)$ and $\neg (v < u)$.

We will now show that (V, <) in fact characterizes causality:

Theorem 3.3 *For two events e and e' of a distributed computation, we have*

(1) $e \rightarrow e'$ iff $V(e) < V(e')$.

(2) $e \parallel e'$ iff $V(e) \parallel V(e')$.

Proof. (1) Suppose that $e \rightarrow e'$ holds. According to Lemma 2.2, $e \in C(e')$; from Definition 2.1 and the fact that $\rightarrow$ is transitive, it follows that causal histories are left-closed with respect to $\rightarrow$, hence we conclude that $C(e) \subseteq C(e')$. Thus, $C(e)[k] \subseteq C(e')[k]$, and therefore $V(e)[k] = \mid C(e)[k] \mid \leq \mid C(e')[k] \mid =$

$V(e')[k]$ for $k = 1,\ldots, N$. That is, $V(e) \leq V(e')$. Because $\rightarrow$ is a strict partial order, $e' \notin C(e)$. Thus, $C(e) \subset C(e')$, and it follows that $V(e) \neq V(e')$.

Conversely, suppose $V(e) < V(e')$. From Observation 2.3, we learn that $C(e)[k] \subseteq C(e')[k]$ for $k = 1,\ldots, N$, i.e., $C(e) \subseteq C(e')$. From $V(e) \neq V(e')$ it follows that $e \neq e'$. But $e \in C(e) \subseteq C(e')$, and therefore $e \rightarrow e'$ must hold according to the definition of $C(e')$.

Property (2) follows immediately from (1) and the definition of concurrency. ❑

Theorem 3.3 offers a convenient method to determine the causal relationship between events based on their vector times. In fact, instead of comparing whole time vectors, the necessary computations can often be reduced even further, according to the following lemma. A proof is straightforward and may be found, for example, in [HaW88].

Lemma 3.4 *For two events $e \in E_i$ and $e' \in E_j$, $e \neq e'$, we have*

(1) $e \rightarrow e'$ *iff* $V(e)[i] \leq V(e')[i]$.

(2) $e \parallel e'$ *iff* $V(e)[i] > V(e')[i]$ *and* $V(e')[j] > V(e)[j]$.

Lemma 3.4 states that we can restrict the comparison to just two vector components in order to determine the precise causal relationship between two events if their origins P_i and P_j are known. The intuitive meaning of the lemma is easy to understand. If the "knowledge" of event e' in P_j about the number of local events in P_i (i.e., $V(e')[i]$) is at least as accurate as the corresponding "knowledge" $V(e)[i]$ of e in P_i, then there must exist a chain of events which propagated this knowledge from e at P_i to e' at P_j, hence $e \rightarrow e'$ must hold. If, on the other hand, event e' is not aware of as many events in P_i as is event e, and e is not aware of as many events in P_j as is e', then both events have no knowledge about each other, and thus they are concurrent. Clearly, the converse arguments are equally valid for both cases.

3.2 Real Time and Lamport Time

The analysis of causality is closely related to temporal reasoning. As everyday experience tells us, every cause must precede its effect. Names like "happened before" [Lam78] and "is concurrent with" for relations which are causal rather than temporal reflect this fact. However, such a terminology — although quite suggestive — is somewhat misleading. In this section, we briefly discuss the relationship between time and causality.

Let $t(e)$ denote the real-time instant at which event e of a given computation takes place. Obviously, idealized real time $(t, <)$ is *consistent with causality*; it does not, however, *characterize causality*, because $t(e) < t(e')$ does not necessarily imply $e \rightarrow e'$. An additional problem is that a set of synchronized local real time clocks, i.e. a proper realization of an idealized "wall clock", is generally not available in distributed systems. Fortunately, it is possible to realize a system of *logical clocks* which guarantees that the timestamps derived are still consistent with causality. This was shown by Lamport in [Lam78].

Definition 3.5 *The* Lamport time *is a mapping $L\colon E \rightarrow \mathbf{N}$ which maps events to integers, defined recursively as follows:*

(1) *If e is an internal event or a send event, and e has no local predecessor, then $L(e) = 1$; if e has a (unique) local predecessor e', then $L(e) = L(e') + 1$.*

(2) *If r is a receive event and s is the corresponding send event, and r has no local predecessor, then $L(r) = L(s) + 1$;*
if r has a (unique) local predecessor e', then $L(r) = \max\{L(s), L(e')\} + 1$.

Figure 3 shows a distributed computation with Lamport timestamps assigned to the events. Lamport time can be implemented easily with a scheme similar to the one of Definition 2.5, but with simple integers instead of vectors [Lam78]. One can easily see that by construction, Lamport time $(L, <)$ is consistent with causality. However, as Figure 3 shows, it does not characterize causality: $L(e_{11}) < L(e_{22})$ although e_{11} and e_{22} are causally independent. Hence, although Lamport time implies a natural partial order on the set of events (by defining that an event e precedes an event e' iff $L(e) < L(e')$), this order is different

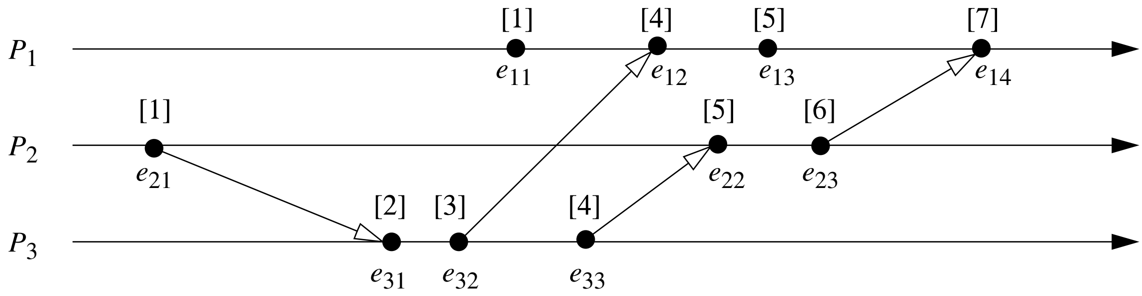

Figure 3: Events of a distributed computation and their associated Lamport timestamps

from $\rightarrow$. We can, however, easily define a linear extension of this implied order, for instance by the following definition.

Definition 3.6 *Let $e \in E_i$, $e' \in E_j$, and let $L(e)$, $L(e')$ denote their Lamport timestamps. The* total event order $\Rightarrow \subseteq E \times E$ *is defined by*

(1) If $L(e) < L(e')$, then $e \Rightarrow e'$.

(2) If $L(e) = L(e')$ and $i < j$ holds, then $e \Rightarrow e'$.

Clearly, $(L, \Rightarrow)$ is consistent with causality (i.e., $\rightarrow \subseteq \Rightarrow$). Hence, if we order all events by $\Rightarrow$, then an event will not occur prior to any other event that might have caused it. Therefore, this ordering can be regarded as a "valid" reconstruction of the linear sequence of atomic actions that actually took place during the distributed computation. However, if two events e and e' are concurrent, then $\Rightarrow$ determines just *one* of several possible, causally consistent interleavings of the local event sequences. Note that even if $t(e) < t(e')$ actually holds for two concurrent events e and e' — which is, of course, not known within the system — $L(e) > L(e')$ is still possible, as the events e_{32} and e_{11} in Figure 3 demonstrate. Interestingly, this is not possible for vector time. If $V(e) < V(e')$ holds, then we necessarily have $t(e) < t(e')$, whereas nothing about the real-time order can be derived from $L(e) < L(e')$.

To summarize our discussion, we remark that Lamport time induces an interleaving of the local event streams which is *consistent* with causality. Thus, although not consistent with real time, Lamport time may serve as an adequate substitute for real time with respect to causality. However, both real time and Lamport time are insufficient to *characterize* causality and can therefore not be used in general to prove that events are *not* causally related. This, however, is quite important for the analysis of distributed computations. Stronger concepts like vector time are required for that purpose.

4 Efficient Realizations of Vector Time

In the previous section we saw that vector time characterizes causality. Furthermore, provided that the vector timestamps of all events are available, Lemma 3.4 offers a convenient method to compute the relations $\rightarrow$ and $\|$. The major drawback of vector time is the size of the time vectors. This might pose problems for massively parallel computations. In this section, we present some techniques for an efficient realization of vector time in distributed computations.

4.1 Compressing Message Timestamps

According to Definition 2.5, all messages of a distributed computation have to be tagged with a timestamp of size N to maintain vector time. If N, the number of processes, is large, the amount of timestamp data that has to be attached to each message seems unacceptable. Two observations may lead to a substantial improvement of the basic technique to maintain vector time:

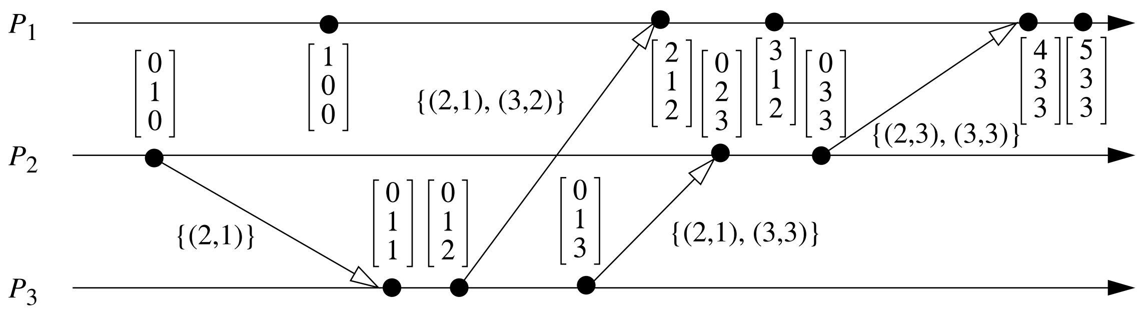

Figure 4: Singhal's and Kshemkalyani's method to maintain vector time

Observation 4.1 *In a distributed computation, we typically observe the following:*

(1) Even if the number N of processes is large, only few of them are likely to interact frequently by direct message exchange.

(2) If we compare an event e with its local successor e', only few entries of the time vector V(e') are likely to differ from those of V(e).

The first observation motivates the second, since, if two processes never directly or indirectly interact, they will never receive new knowledge about each other's causal histories, and hence the corresponding vector entries remain unchanged.

Based on Observation 4.1, Singhal and Kshemkalyani [SiK92] propose an improved implementation technique for vector clocks which typically saves communication bandwidth at the cost of slightly increased storage requirements. Their idea is to append only those entries of the local time vector V_i to a message sent to P_j which have changed since the last transmission to P_j. For this purpose, each process P_i maintains two additional vectors LS_i ("last sent") and LU_i ("last update"). $LS_i[j]$ is set to the "local time" $V_i[i]$ when P_i sends a message to P_j. $LU_i[j]$ is set to $V_i[i]$ when P_i updates entry $V_i[j]$, which (for $i \neq j$) can only appear on receipt of a message. Instead of timestamping each message with $V(m) = V_i$ when sending a message to P_j (see Definition 2.5), process P_i behaves as follows after incrementing $V_i[i]$ and setting $LU_i[i]$ to $V_i[i]$:

(1) For $k = 1,\ldots, N$, if $LU_i[k] > LS_i[j]$ then a pair $(k, V_i[k])$ is added to an (initially empty) set $S(m)$.

(2) The message m is sent together with $S(m)$ to its destination P_j, and $LS_i[j] := V_i[i]$.

According to rule (1) above, $S(m)$ contains exactly those entries of V_i which have been updated since their last transmission to P_j — the only entries which may cause a change of P_j's local vector V_j. Thus, it is obviously sufficient to just send $S(m)$ instead of $V(m)$ in order to maintain the local time vector. Note, however, that FIFO channels are required; otherwise, the information in $S(m)$ might be insufficient for a proper update of the receiver's time vector. Figure 4 shows an example of how the technique works.

For large systems, the proposed method can result in substantial savings in communication bandwidth. However, it suffers from a slight deficiency, as mentioned by Meldal et al. in [Mel91]. By compressing the message timestamps, we lose immediate access to some information about the causal relationship between different messages sent to the same receiver. In particular, it is no longer possible to decide whether two such messages (or, more precisely, their corresponding send events) are causally dependent solely by comparing their (compressed) timestamps. This is illustrated in Figure 5. In both scenarios shown, P_3 receives messages m_1 and m_2 at times $V(r_1) = [1, 0, 1]$ and $V(r_2) = [2, 3, 4]$, respectively; the compressed message timestamps are $S(m_1) = \{(1,1)\}$ and $S(m_2) = \{(2, 3)\}$. However, in the first scenario m_1 and m_2 are causally unrelated, while in the second m_2 causally depends on m_1 because the send event of m_1 causally precedes the send event of m_2. From P_3's point of view, the two different scenarios are indistinguishable. Note that if messages were equipped with the full vector timestamps, then the receiver P_3 would know whether m_1 and m_2 are causally unrelated ([1, 0, 0] || [0, 3, 0] in the first

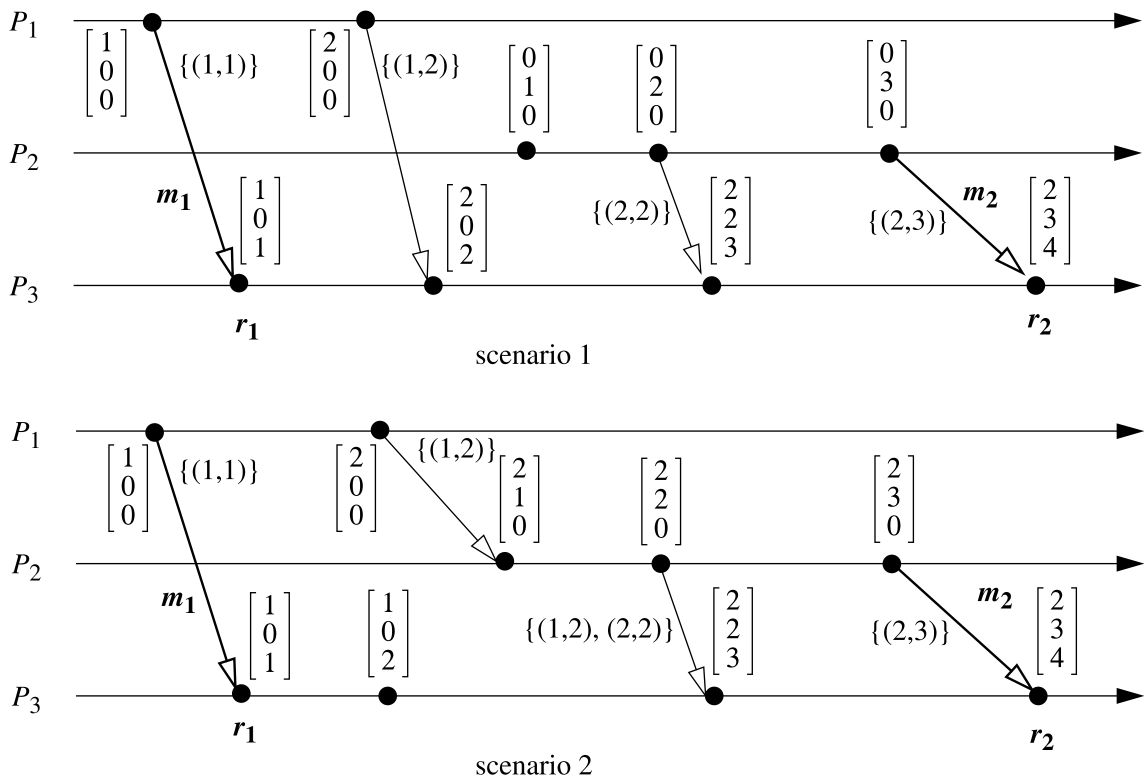

Figure 5: Loss of information about the causal relationship between two messages

scenario) or not ([1, 0, 0] < [2, 3, 0] in the second scenario). In particular, P_3 would then be able to determine that it received m_1 “out of causal order” if in the second scenario m_1 is delayed such that it arrives after m_2.

With compressed timestamps, this is impossible if only compressed message timestamps are taken into account. In principle, however, no information is actually lost because the compression scheme only suppresses those parts of a message’s timestamp which are already known to the receiver. That is, each process may recover the original, uncompressed timestamps, but this would require the collection of some additional information about the local vector time at which the compressed timestamps were received, and about the components of the local time vector which were last updated. Thus, in applications like, e.g., causally ordered message delivery protocols [Bir91, CMT92, Sch89] where such detailed knowledge is required, some additional book-keeping and computational effort is needed to locally restore the suppressed information. An approach to recover the full timestamp of each message requiring $O(N^2)$ space at each process may be found in [SiK92].

4.2 Reconstructing Time Vectors

In the previous section, it was shown how message timestamps can be efficiently coded so as to save communication bandwidth in typical cases. The technique is especially valuable if the number N of processes is large. The main disadvantage, however, is that the size of the message timestamps is still linear in N in the worst case; also, three vectors per process are needed instead of just one as in the basic approach. In this section, we try to further reduce the amount of data that has to be attached to each message in order to maintain vector time. However, this comes at the cost of an increased computational overhead for the calculation of the time vectors assigned to events. In most cases, this overhead is probably too large for an on-line computation because this would slow down the distributed computation in an

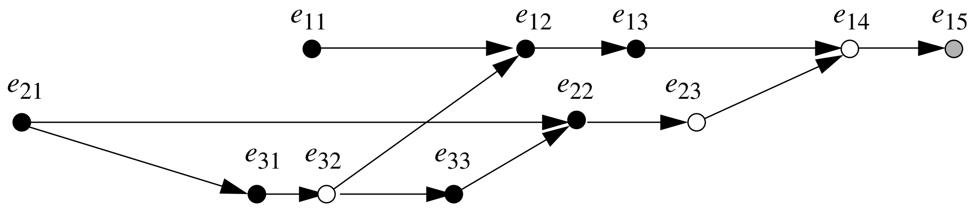

Figure 6: Directed graph corresponding to the time diagram of Figure 4

unacceptable way. The methods described here might be used, however, for a trace-based off-line analysis of the causality relation.

Recall that the vector timestamp is just a compact notation for the causal history of an event. That is, in principle we can determine the vector timestamp of an event e by simply computing the set of all events in the time diagram that can reach e via a directed path. Note that a time diagram is basically a directed graph; for example, Figure 6 shows the graph resulting from the time diagram depicted in Figure 4. There are several well-known algorithms which compute reachable vertices in directed graphs. However, these algorithms do not efficiently exploit the peculiar structure of time diagrams, in particular:

- In a directed graph derived from a time diagram, each vertex (i.e., each event) has at most two direct predecessors.
- Vertices denoting events occurring in the same process are totally ordered and thus form a directed path in the graph. That is, a graph that represents a distributed computation comprising N processes contains N local "chains".

As a result, the general algorithms are too inefficient; the Floyd-Warshall algorithm, for example, requires $O(K^3)$ steps to determine the reachability matrix for a directed, acyclic graph containing K vertices [Flo62, War62]. For the special case of time diagrams, more efficient solutions are feasible.

Figure 7 shows a simple recursive graph searching algorithm which determines the vector time $V(e)$ of an event $e \in E$. Basically, the algorithm determines $V(e)$ by applying a recursive "backward search" on the directed graph, and by counting the events belonging to $C(e)$. Since GraphSearch is called exactly once for each event in $C(e)$, the complexity for determining the vector timestamp of a single event is linear in the number of events.

```
TimeVector (e: Event)
    /* Computes the vector time V(e) for event e */
    Assign 0 to all components of V(e);
    GraphSearch(e);
    return V(e);
end TimeVector;

GraphSearch (z: Event)
    Mark z as visited;
    Determine i such that z ∈ E_i;
    V(e)[i]:= V(e)[i] + 1;
    if z has an unmarked direct local predecessor x ∈ E_i,
        then GraphSearch(x) endif;
    if z has an unmarked direct non-local predecessor y ∈ E_j, i ≠ j,
        then GraphSearch(y) endif;
end GraphSearch;
```

Figure 7: Simple algorithm for the reconstruction of $V(e)$

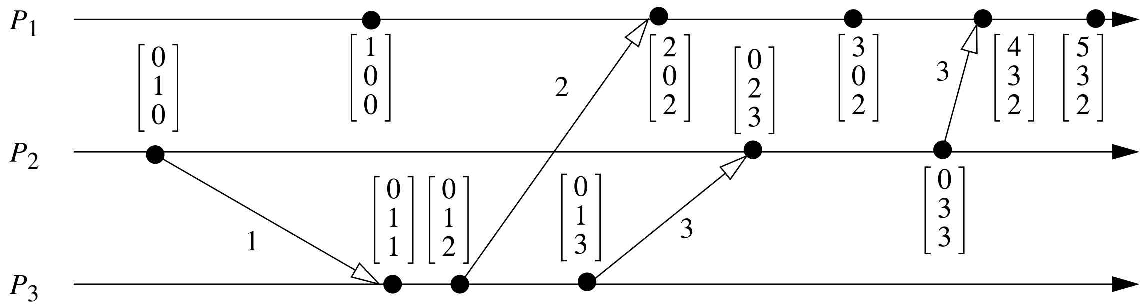

Figure 8: Maintaining direct dependency vectors

It should be noted that the algorithm is suited for an off-line computation of time vectors and causal histories where the graph corresponding to the time diagram is available in the sense that each event can provide a pointer to its non-local predecessor. Reconstructing time vectors and causal histories *after* the distributed computation might be computationally expensive but it avoids the sending of vector timestamps in messages and the keeping of local time vectors during the execution of the application. It clearly depends on the application whether such an approach is suitable or not.

The algorithm depicted in Figure 7 is linear in the “duration” of the distributed computation, because $|E|$ increases as the computation proceeds. Hence, it may take rather long to reconstruct the time vectors of “late” events. Fortunately, the graph of a distributed computation consists of N totally ordered chains of events. Therefore, it suffices to know the *most recent* predecessor of event e with respect to each chain in order to determine $C(e)$; indexing all events in standard enumeration will yield the prefix of each chain. To exploit this fact, it is required that a process keeps track of the most recent event in each process which directly influenced it. As an example, in Figure 6 the events of each process which *directly* affected P_1 *most recently* with respect to event e_{15} (namely, e_{14}, e_{23} and e_{32}) are depicted as white dots. By maintaining at runtime a vector D, such that $D(e)[k]$ denotes the index of the event in P_k which most recently (with respect to e) sent a message to e’s process, we can derive $V(e)$ with even less effort than is required by the algorithm depicted in Figure 7.

The approach described here is due to Fowler and Zwaenepoel [FoZ90]. Basically, their “time vectors” only reflect *direct* dependencies, while vector time takes into account also *transitive* dependencies. By ignoring indirect causal relationships, it suffices to attach only a single event index (i.e., a scalar instead of a vector) to each message that is transmitted. As a disadvantage, transitive causal dependencies must be re-computed for each event. More specifically, each process P_i maintains a *dependency vector* D_i as follows:

(1) Initially, $D_i[j]:= 0$ for all $j = 1,\ldots, N$.
(2) On each event occurrence in P_i, D_i is incremented: $D_i[i]:= D_i[i] + 1$.
(3) On sending a message m, P_i attaches (the incremented) value $D(m) = D_i[i]$ to m.
(4) On receiving message m sent by P_j with attached value $D(m)$, P_i updates D_i:
$D_i[j]:= \max\{D_i[j], D(m)\}$.

Let $D(e)$ denote the dependency vector associated to an event e, and more particularly, let $D_i(k)$ denote the dependency vector D_i which results from the occurrence of the k-th event in process P_i. As with $V_i[i]$, $D_i[i]$ serves as an event counter for the local events in P_i, i.e., $D_i(k)[i] = k$. For $i \neq j$, $D_i(k)[j]$ denotes the sequence number of the most recent event in P_j (actually, a send event) that *directly* influenced the k-th event in P_i. Figure 8 depicts the distributed computation of Figure 4 with the resulting dependency vectors. If we compare Figure 4 with Figure 8, we observe that $D(e) \le V(e)$ for all e, which is obviously true in general.

In order to determine the *transitive* causal dependencies necessary for the full time vectors, $V(e)$ is derived from $D(e)$ by recursively retracing the *direct* dependencies, i.e., by computing the transitive left-closure of the direct dependence relation. In [FoZ90], Fowler and Zwaenepoel present a simple procedure which transforms $D(e)$ into the corresponding $V(e)$. Their method is very similar to the simple graph searching algorithm presented in Figure 7; the main difference is that the events on which an event e directly depends are not retraced, but can be addressed immediately via their index stored in $D(e)$. Figure 9 depicts the graph of Figure 6 with the dependency vectors attached to each event; obviously, the entries of the vectors serve as pointers to the most recent events which potentially had influence on the process. By performing a depth-first walk along each dependence path, using the indices of D_i as pointers to the next relevant predecessor events, Fowler's and Zwaenepoel's method reconstructs the left-closure of the direct dependence relation in at most $O(M)$ steps, where M denotes the total number of messages sent during the computation. The details of their algorithm and a derivation of its complexity bound may be found in [FoZ90].

Recently, Baldy et al. proposed an improved variant [Bal92]. Basically, their idea is to apply breadth-first retracing to all dependence paths in parallel instead of following each single path in a depth-first manner as was proposed by Fowler and Zwaenepoel. Figure 10 shows the resulting algorithm. A formal proof of its correctness may be found in [Bal92]; here, we merely present an informal correctness argument.

Initially, the algorithm starts with $D(e)$ as a first approximation of $V(e)$. Consider the first iteration of the algorithm's outer loop. What the algorithm actually does is to visit all *direct* causal predecessors of event e. These predecessors are directly accessed in the inner loop by interpreting the components of $D(e)$ as pointers, as shown in Figure 9. The components provided by those predecessors' dependency vectors — i.e., pointers to their respective predecessor events — are taken into account by considering $sup\ \{D(x) \mid x \text{ is pointed at by a component of } D(e)\}$ and by updating the approximation $V(e)$ accordingly. As noted earlier, $D_i(k)[i] = k$. Therefore, after one iteration of the outer loop, at least the indices of all *immediate* predecessors have been incorporated into $V(e)$, and $V(e)$ contains pointers to the predecessors of e at indirection level 1. By a simple, inductive argument it is easy to see that after the l-th iteration of the outer loop at least all predecessor events of e at indirection level l-1 have been determined, and that $V(e)$ contains pointers to those predecessors at indirection level l. It remains to be shown that the outer loop terminates after a finite number of iterations.

To see why this is in fact the case, recall that if we follow the dependence path of an event $e \in E_i$ back into the past starting at process P_i, we can stop the retracing as soon as we come back to some event $e' \in E_i$ again — since the dependence paths do not contain cycles, e' is a local predecessor of e and the past of e' is already contained in the past of e. Therefore, as soon as the retracing of a dependence path returns to a process which has already been visited before during the inspection of that path, we can

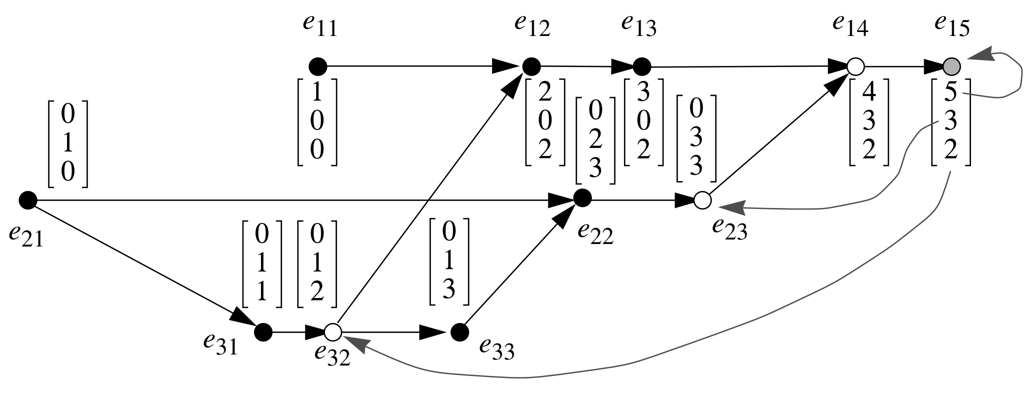

Figure 9: Dependency vectors as pointer arrays

```
VectorTime (e: Event)
    /* Computes V(e) for event e in a distributed system of N processes */
    V(e):= D(e);
    repeat /* outer loop: executed at most N times */
        old_V := V(e);
        for k := 1 to N do /* inner loop: executed N times */
            V(e) := sup( V(e), Dk(old_V[k]) ); /* requires N comparisons */
        endfor
    until old_V = V(e) endrepeat;
    return V(e)
end VectorTime;
```

Figure 10: Algorithm for the conversion of *D*(*e*) to *V(e)* according to Baldy et al.

immediately stop its further retracing. As there are only *N* processes and every retracing step of the path must jump to a process that has not been visited yet, the set of processes not visited so far is exhausted after at most *N* steps. Therefore, the maximum number of steps to complete the inspection of a single dependence path is *N*. The algorithm depicted in Figure 10 inspects all *N* dependence paths originating at *e* in parallel; it therefore requires at most *N* iterations of its outer loop. Consequently, the number of execution steps required for the reconstruction of a single time vector is bounded by $O(N^3)$ which compares favorably to the complexity derived for Fowler's and Zwaenepoel's original scheme. In fact, according to Baldy et al. an even more efficient algorithm which reconstructs *V(e)* in at most $O(N^2)$ steps is feasible, but requires a more involved and less intuitive derivation. The interested reader is referred to [Bal92] for further details.

Like Fowler's and Zwaenepoel's method, the algorithm proposed by Baldy et al. requires random access to all local event streams. If events occur rarely, and a large amount of data has to be recorded for each event anyway, then a reconstruction approach might be advantageous; a typical example is dependency tracking in distributed databases [StY85]. On the other hand, if events occur very frequently, then it might be impossible to record the complete event traces which are required for a reconstruction of all vector timestamps. In such cases, vector time has to be maintained on-the-fly by the classical scheme described earlier. Typically, on-line monitors belong to this type of applications; there, complex reconstruction schemes are prohibitive anyway because they are too time expensive.

Finally, it should be noted that Fowler's and Zwaenepoel's original aim was to compute *causal distributed breakpoints* rather than vector time. Informally, the causal distributed breakpoint corresponding to an event *e* is defined as the earliest consistent global state that contains *e*. That is, in order to guarantee minimality and consistency, the breakpoint reflects the occurrence of an event e' if and only if $e' \in C(e)$. Hence, causal distributed breakpoints and causal histories are equivalent. Since according to Observation 2.3 $C(e)$ is isomorphic to $V(e)$, this explains why Fowler's and Zwaenepoel's algorithm and the algorithm depicted in Figure 10 actually compute vector time.

4.3 About the Size of Vector Clocks

Vector time is a powerful concept for the analysis of the causal structure of distributed computations. However, having to deal with timestamps of size *N* seems unsatisfactory — even if we take into account the improvements suggested in the previous sections. The question remains whether it is really necessary to use time vectors of that size. Is there a way to find a "better" timestamping algorithm based on smaller time vectors which truly characterizes causality?

As it seems, the answer is negative. Charron-Bost showed in [Cha91] that causality can be characterized only by vector timestamps of size *N*. More precisely, she showed that the causal order $(E, \rightarrow)$ of a distributed computation of *N* processes has in general *dimension N*. This induces a lower bound on the size of time vectors because a partial order of dimension N can be *embedded* in the partially ordered set $(\mathbf{R}^k, <)$ of real-valued vectors only if $k \geq N$. We summarize Charron-Bost's results:

Definition 4.2 *A partial order $(X, <')$ is said to be* isomorphically embedded *into a partial order $(Y, <)$ if there exists a mapping $\phi: X \rightarrow Y$ such that for all $x, y \in X$, $\phi(x) < \phi(y)$ iff $x <' y$.*

Note that according to Definition 3.1, $(X, <')$ characterizes causality iff $(E, \rightarrow)$ can be isomorphically embedded into $(X, <')$.

Definition 4.3 *Let $(X, <)$ denote a partial order. A* realizer *of $(X, <)$ is a set of linear extensions of $(X, <)$ such that the intersection of all extensions is equal to $(X, <)$. The cardinality of a smallest realizer of $(X, <)$ is called the* dimension *of $(X, <)$, denoted $dim(X, <)$.*

We cite Ore's characterization of the dimension of a partial order from [Cha91]:

Theorem 4.4 (Ore) *A finite partially ordered set $(X, <')$ can be isomorphically embedded into $(\mathbf{R}^k, <)$ if and only if $k \geq dim(X, <')$.*

The following theorem is the key result of [Cha91]:

Theorem 4.5 *For every N there exist processes $P_1, \ldots, P_N$ forming a distributed computation, and a set E of events produced by that computation, such that $dim(E, \rightarrow) = N$.*

For a proof the reader is referred to [Cha91]. What this theorem actually implies is that, if we represent logical time by integer-valued vectors and if we use the canonical vector order $<$ of Definition 3.2 to compare these vectors, then we need vectors of size N to isomorphically embed the $\rightarrow$ relation, i.e., to characterize causality — no matter what scheme is applied to maintain the time vectors. However, it does *not* imply that vectors of dimension N are mandatory! In fact, we can uniquely map each vector on a (rather large) scalar value and vice versa. Typically, this will result in scalars which are at least as "clumsy" as vectors are. But still, it is not immediately evident that — for a more sophisticated type of vector order than $<$ — a smaller vector could not suffice to characterize causality, although the result of Charron-Bost seems to indicate that this is rather unlikely. At least we have the following fact [Cha91]:

Corollary 4.6 *Let T denote a set of an arbitrary kind of timestamps assigned to the events of arbitrary computations of N processes. Any partial order $(T, <')$ that characterizes causality must have a dimension $dim(T, <') \geq N$.*

Proof. (By contradiction). According to Theorem 4.5, choose a distributed computation of N processes, such that $\dim (E, \rightarrow) = N$. Suppose that $\dim(T, <') = k < N$. Theorem 4.4 states that there exists a mapping $\phi: T \rightarrow \mathbf{R}^k$ which embeds $(T, <')$ into $(\mathbf{R}^k, <)$. If $(T, <')$ characterizes causality, we have for all $e, e' \in E$:

(1) $T(e) <' T(e')$ *iff* $e \rightarrow e'$.

(2) $\phi(T(e)) < \phi(T(e'))$ *iff* $T(e) <' T(e')$.

Putting (1) and (2) together, we obtain $\phi(T(e)) < \phi(T(e'))$ iff $e \rightarrow e'$. Thus, $(\phi \circ T)$ is a mapping that embeds $\rightarrow$ into $(\mathbf{R}^k, <)$, and according to Theorem 4.4 $\dim(E, \rightarrow) \leq k < N$, which contradicts our choice of E. ❑

A definite theorem about the size of vector clocks would require some statement about the minimum amount of information that has to be contained in timestamps in order to define a partial order of dimension N on them. Finding such an information theoretical proof is still an open problem.

5 Characterizing Concurrency with Concurrent Regions

In the previous section, we gave a brief survey of known techniques to characterize causality by timestamping the events of a distributed computation. Two main results were obtained:

- By using vector time, it is possible to faithfully represent causality.
- Although several refinements to the basic vector time approach are feasible, timestamps character-

izing causality seem intrinsically complex.

The latter insight is somewhat disappointing, because it might substantially limit the application of vector time in practice. Therefore, it is not surprising that alternative ways to assess causality were pursued. In this section, we will investigate a popular approach, namely the concept of *concurrent regions*.

5.1 Concurrent Regions and Concurrency Maps

For some applications like, for example, the detection of race conditions in concurrent computations, it is sufficient to know whether two arbitrary events e and e' occurred concurrently or not; if $e \parallel e'$ does not hold, then the exact causal relation (i.e., $e \rightarrow e'$ or $e' \rightarrow e$) is irrelevant. One might suspect that it is cheaper to supply only this restricted form of "concurrency information" instead of the full causality relation.

Consider, for example, event x of the distributed computation depicted in Figure 11. All events occurring in the shaded segments of the time lines of P_1 and P_2 are causally independent from x, and therefore, according to Definition 1.2, concurrent with x. These segments form *concurrent regions* with respect to x. If it were possible to identify such regions with only little effort, then detecting concurrency would be simple!

For a first step towards this aim, it might be useful to visualize concurrent regions with a *concurrency map*. This approach was proposed by Stone, who suggested the use of concurrency maps to support the visual analysis of concurrent processes [Sto88, Sto89]. To this end, the local event streams of a distributed computation are partitioned into so-called *dependence blocks*. The underlying idea is that all events contained in a dependence block can be regarded as a single, atomic "super event", i.e., if one of these events is concurrent with a non-local event e, then all the other events occurring in that dependence block are concurrent with e, too. More formally, let us define an equivalence relation $(E_i, \sim)$ on the set of events local to a process P_i as follows:

Definition 5.1: *For $x, y \in E_i$, the relation $x \sim y$ holds if and only if for all $z \in E \setminus E_i$ the following conditions are satisfied:*

$$\begin{array}{lll} x \parallel z & \text{iff} & y \parallel z \quad \wedge \\ x \rightarrow z & \text{iff} & y \rightarrow z \quad \wedge \\ z \rightarrow x & \text{iff} & z \rightarrow y \end{array}$$

The dependence block DB of an event e_{ij} can now be characterized as an equivalence class with respect to $(E_i, \sim)$, i.e., $DB(e_{ij}) ::= \{x \in E_i \mid x \sim e_{ij}\}$. Note, that this definition implies that the borders between two dependence blocks on a time line necessarily lie after a send event and before a receive event

The causality relation on events induces dependencies between different dependence blocks. More specifically, the first event ("successor event") of some block may depend on the occurrence of the last event ("predecessor event") of another block on a different time line. That is, each send event is a predecessor event, while the corresponding receive event is a successor event. The concurrency map is obtained by partitioning the time diagram by vertical lines in *history segments* such that causally dependent dependence blocks on different process lines occur in different segments, whereas dependence blocks on different process lines appearing in the same segment are concurrent. Figure 12 shows a concurrency map of the computation depicted in Figure 11. For more details about the construction of con-

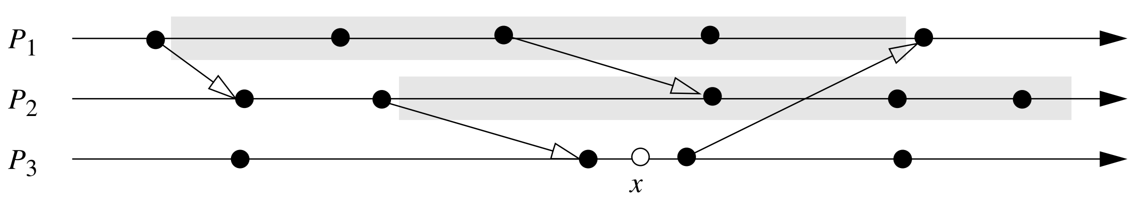

Figure 11: The concurrent regions with respect to event x

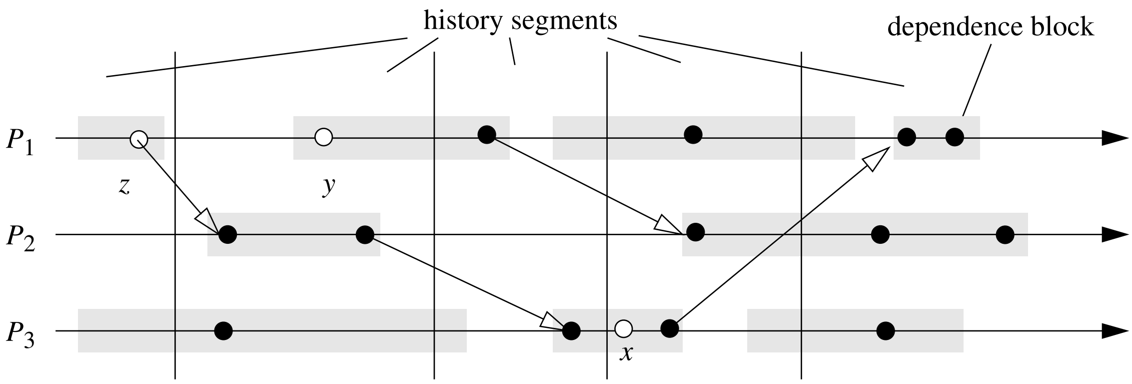

Figure 12: A concurrency map according to Stone

currency maps, the interested reader is referred to [Sto89].

The following *transformations* which preserve the causal dependencies may be applied to concurrency maps:

(1) The horizontal size of a dependence block may be scaled up and down by any factor, as long as blocks on the same time line do not overlap.

(2) The position of events within a dependence block may change, as long as their relative order remains untouched.

(3) Dependence blocks may be moved to the right or to the left; they may even cross the boundary of a history segment, with the following restriction: The dependence block of a predecessor event and the dependence block of the corresponding successor event must always remain separated by a history segment boundary.

A concurrency map (together with its feasible transformations) implicitly represents all possible total event orderings which are consistent with causality. In [Sto87] it is shown that for every distributed computation the construction of a concurrency map is in fact possible, and that for two given events e and e', $e \parallel e'$ holds if and only if there is a transformation of the concurrency map such that e and e' occur in the same history segment. For example, in Figure 12 event x and event y are clearly concurrent, because according to rule (2) we can move x one segment to the left, and y one segment to the right. Also, x and z are clearly *not* concurrent, because according to rule (3) they have to be separated by at least two segment boundaries.

5.2 Identifying Concurrent Regions with Region Numbers

Stone's dependence blocks are good candidates for the construction of *concurrent regions*. All that is needed is an efficient method for the identification of history segments and a tag for each block that tells us which history segments the block may possibly enter. If $DB(x)$ may enter the same history segment as $DB(y)$, then the events x and y are concurrent, otherwise they are causally dependent.

Ideally, we would like to divide the time diagram of a given computation into contiguous regions corresponding to dependency blocks, and we would like to assign some number to each region such that two given events x and y are concurrent if and only if the numbers of their respective regions satisfy a simple criterion. Can we assign appropriate region numbers and define a suitable binary relation that characterizes concurrency in the sense of the following definition?

Definition 5.2 *Let E denote the set of events of a distributed computation, let S denote an arbitrary set, and let $\# \subseteq S \times S$ denote an arbitrary binary relation. Let ϕ: $E \rightarrow S$ denote a mapping. $(\phi, \#)$ is said to characterize concurrency, if for all $e, e' \in E$*

$$e \parallel e' \quad \text{iff} \quad \phi(e) \# \phi(e').$$

For a given ϕ, we say for short that $(S, \#)$ characterizes concurrency.

This definition should be compared to Definition 3.1 where the characterization of causality is defined.

Unfortunately, it can be shown that the problem of characterizing *causality* is essentially reducible to characterizing *concurrency*. Note that once we are able to characterize concurrency, we can determine whether two given events x and y are causally dependent or not; if they turn out to be causally related, we can simply use Lamport time to distinguish between $x \rightarrow y$ and $y \rightarrow x$. This shows that region numbers must have essentially the same complexity as time vectors. Therefore the results of Section 4.3 still apply. That is, we cannot really hope to gain much by substituting region numbers for vector time.

For a formal proof of our informal reasoning, let us assume that we know a mapping $\phi: E \rightarrow \mathbf{R}$, such that $\phi(e)$ denotes the region number of the region to which event e belongs. Let us further assume that a binary relation # exists, such that $(\phi, \#)$ characterizes concurrency. In other words, suppose we are able to identify concurrent regions with simple real-valued region numbers. If both ϕ and # are computable, then we can show that there exists an implementation of vector time which only requires vectors of size 2:

Proposition 5.3 *Let $\phi: E \rightarrow \mathbf{R}$ denote a mapping, and let $\# \subseteq \mathbf{R} \times \mathbf{R}$ denote a binary relation such that $(\phi, \#)$ characterizes concurrency. Then there exists a mapping $\phi': E \rightarrow \mathbf{R} \times \mathbf{N}$, and a partial order $<'$ on $(\mathbf{R} \times \mathbf{N})$, such that $(\phi', <')$ characterizes causality.*

Proof. Define ϕ' as follows:

(1) $\phi'(e) := [\phi(e), L(e)]$, where $L(e)$ denotes the Lamport time of event e.

Next, define $<'$ as follows:

(2) $\phi'(e) <' \phi'(e')$ iff $\neg (\phi'(e)[1] \# \phi'(e')[1]) \wedge (\phi'(e)[2] < \phi'(e')[2])$.

According to Definition 5.2, we can restate (2) in terms of events and their Lamport times:

(2') $\phi'(e) <' \phi'(e')$ iff $\neg (e \parallel e') \wedge (L(e) < L(e'))$.

We note that Lamport time is consistent with causality. Thus, $e \rightarrow e'$ implies $\phi'(e) <' \phi'(e')$. If, on the other hand, $e \rightarrow e'$ does not hold, it follows that $e' \rightarrow e$ holds implying $L(e') < L(e)$, or that $e \parallel e'$ is satisfied. In either case, $\phi'(e) <' \phi'(e')$ does not hold according to (2'). As a consequence, $(\phi', <')$ characterizes causality. ❑

Actually, $(\phi', <')$ is an alternative realization of vector time. Note that $<'$ can be computed with almost as little effort as #. Only one additional vector entry is needed, and maintaining Lamport time does not require any additional messages. Note that it is straightforward to extend Proposition 5.3 to the general case where we take region numbers from the domain $\mathbf{R}^k$ instead of $\mathbf{R}$. Hence, if there is a way to implement region numbers based on vectors of size k which characterize concurrency, we can immediately derive an implementation of vector timestamps of size $(k+1)$ which characterize causality.

As we have seen in Section 4.3, there are good reasons to believe that vector time is inherently complex to compute and requires vectors of dimension N. Thus, Proposition 5.3 seems to imply that we cannot hope for region numbers which identify concurrent regions smaller than of size $(N-1)$. Anyhow, whatever the actual size of vectors may be that is required for a realization of vector time: region numbers require vectors of essentially the same size.

An approach for the detection of concurrency based on comparing the numbers of concurrent regions is described by Spezialetti and Kearns in [SpK89]. In their model, it is assumed that there exists an *event monitor* which observes the local state changes (i.e., events) and determines *global state changes* by combining appropriate concurrent local events to so-called *global events*. Consequently, the problem that has to be solved is to determine whether two given events are concurrent or not. To this end, so-called *regions* are defined local to each process and *region numbers* are attached to each of them, such that two events e and e' are considered as concurrent if and only if their corresponding region numbers are equal. Interestingly, Spezialetti and Kearns base their notion of concurrency on $(\mathbf{N}, =)$, i.e., on integer region numbers and on simple equality. According to Observation 1.3, however, concurrency is not an equivalence relation; it follows that $(\mathbf{N}, =)$ cannot suffice to characterize concurrency. Consequently, without going into the details of the detection algorithm presented in [SpK89], our discussion reveals that Spezialetti's and Kearns' notion of concurrency must be incomplete in some way or the other. In fact, Cooper and Marzullo [CoM91] present a simple scenario where the proposed algorithm fails to detect a global event that actually occurred.

The discussion presented in this section supports our claim that detecting causal relationships in distributed computations is far from being trivial. Furthermore, it shows that it is important to have a clear understanding of the fundamental characteristics of $\|$ and $\rightarrow$ to avoid fundamental misconceptions in the approach that is taken to tackle the problem.

6 Evaluating Global Predicates

It is often required to know whether for a distributed computation a certain property holds or does not hold. Formally, properties are predicates of the global state. An important application domain for global predicates is the field of debugging. Typically, the expected behavior or suspected misbehavior of the system under test is specified as a global predicate, and debugging is done by checking whether this predicate is satisfied at runtime or not. In order to be sensible, the underlying global states on which predicates are evaluated must be causally consistent — if the effect of an event is reflected by the state, then its cause must also be reflected by it. Or, to put it differently, an observer of the computation must never observe an effect before its cause. However, as we shall see, it is usually the case that different observers see different, mutually exclusive consistent global states. It might thus happen that one observer establishes the truth of a given predicate, while another observer does not. This seemingly paradoxical situation gives rise to a more detailed analysis of the underlying notions and concepts. It turns out that in distributed systems, a proper evaluation of global predicates requires a careful consideration of the causal structure that the computation reveals. In this section, the impact of causality on global predicate detection is discussed, and some detection schemes are surveyed.

6.1 Computations, Observations, and Global States

In the previous sections, we used terms like "observer", "observation", or "global state" in a rather informal way. Before continuing our discussion, we need to elaborate these concepts a little further. Our discussion is based on some notions which have their origin in concurrency theory and in temporal logic. In particular, the subsequent presentation shares many concepts with Katz's and Peled's work on interleaving set temporal logic [KaP90, KaP92], with Pratt's geometric model of concurrency [Pra91], and with Reisig's causality based partial order semantics of non-sequential systems [Rei88, Rei91]. It should be noted, however, that most of these theories are based on more abstract models (where, for example, the notion of processes in the sense of linearly ordered disjoint subsets of events does not exist), and that a different terminology is used in most cases.

Informally, a *distributed computation* is an execution of a distributed program which consists of communicating sequential processes. Because of the nondeterminism introduced by varying message delays, a single distributed program usually allows several different computations. If we assume that each event which appears in the course of a computation is timestamped with the real-time instant of its occurrence, then each computation corresponds to a unique time diagram with real-time axes. Of course, real time is generally not available, and any *observation* of the computation suffers from unpredictable notification delays. Hence, observations will not preserve the real-time relation between the events; it is, however, possible to preserve the causality relation, as will be sketched further down.

If we abstract from real time, then a distributed computation, i.e., a single execution of a distributed program, allows different *views* in the sense of different but equivalent time diagrams, as is shown in Figure 13. Two time diagrams are considered equivalent if one can be transformed into the other by stretching or compressing the process axes (or parts of them) without changing the relative order of events. Hence, equivalent time diagrams always represent the same partial order of the causality relation. It seems plausible to assume that each such time diagram represents an equally valid view of the compu-

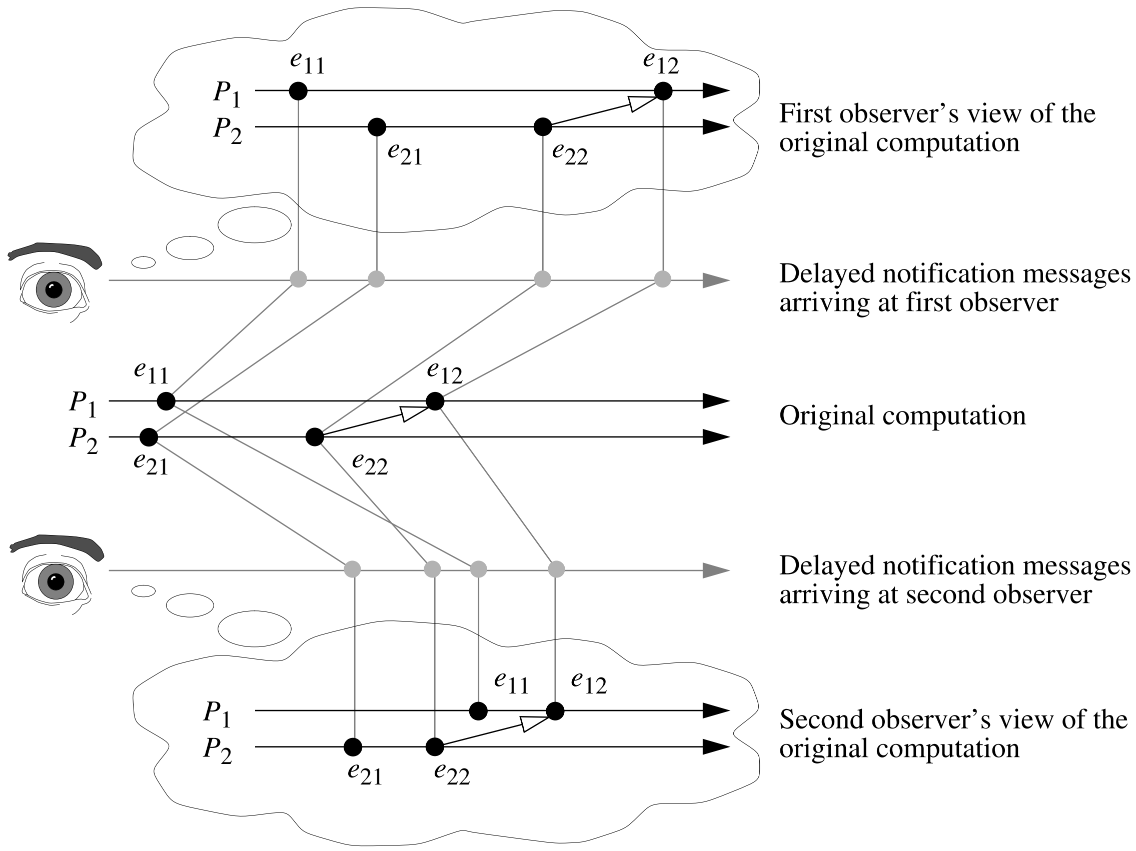

Figure 13: Different but equivalent views of a single, distributed computation

tation, and that any observation which allows such a view is correct. Therefore, we postulate that an *observer* is an entity which observes event occurrences in a strictly sequential manner, one after the other. Typically, measures are taken to guarantee that the observed event sequence is consistent with causality, i.e., that cause and effect always occur in the correct order to avoid confusion. This can be done, for example, by using a causal delivery order protocol as will be described in Section 7.

Since any member of a class of equivalent time diagrams represents an equally valid view of the computation, any vertical projection of the events of such a time diagram onto the hypothetical global time axis represents a valid observation of the computation. Or, conversely, for a given causally consistent observation (where the events are stamped with their "observation time") it is possible to reconstruct a valid view in the form of a time diagram, as shown in Figure 13. This motivates the following definition:

Definition 6.1 *An* observation *of a distributed computation is a linear extension $(E, \ll)$ of the causality relation $(E, \rightarrow)$, such that for all events $e \in E$ the set $\{e' \in E \mid e' \ll e\}$ is finite. An entity that is capable of obtaining a specific observation is called an* observer.

The required finite cardinality of $\{e' \in E \mid e' \ll e\}$ — the so-called *axiom of finite causes* [Win88] — ensures that, even for an infinite set of events, the observation is *fair* in the sense that every event on every process is observed within finite time.

In general, many different observations of a single computation exist; a special case is a computation consisting of only a single process, namely, a sequential program: Here, exactly one observation is possible. Interestingly, it follows from Szpilrajn's theorem [Szp30] that the intersection of all possible observations, i.e., *what all observations have in common*, is precisely the causality relation $(E, \rightarrow)$ which is the essence of the computation. This shows again that the possible observations are all equivalent with respect to causality; none of them is superior in reflecting "reality" if global time is not available.

Usually, a *global state* of a distributed computation is defined as a collection of the local states of all processes *at a certain instant of time* (for simplicity, we assume that *messages in transit* are appropriately reflected by the local states of their senders and receivers). As global time is not available, we need an adequate substitute for the notion of a *real-time instant* — a so-called *consistent cut*:

Definition 6.2 *A finite subset $C \subseteq E$ is called a* consistent cut, *iff $e \in C$ implies $e' \in C$ for all $e' \to e$.*

That is, a consistent cut is a subset of E which is left-closed with respect to causality. It follows immediately that the causal history of an event (Definition 2.1) forms a consistent cut. We can depict a consistent cut in a time diagram by drawing a *cut line* which separates C on the left from $E \setminus C$ on the right, as shown in Figure 14. Note that a message can never cross the cut line of a consistent cut from right to left, for that would imply that the receive event for that message belongs to the cut, while the corresponding send event — which, of course, causally precedes the receipt — does not. Conversely, any line which is consistent in the sense that it cuts the time diagram into a left part and a right part such that no message crosses the line from right to left defines a consistent cut. Thus, consistent cuts and consistent cut lines correspond to each other. Intuitively, a cut line can be interpreted as an instant of (logical) time that consistently partitions a time diagram into *past* and *future*. It should also be clear that for a given time diagram with a cut line of a consistent cut there is always an equivalent time diagram where the cut line forms a straight vertical line. This motivates again that consistent cuts are adequate substitutes for real-time instants.

It is now possible to define consistent global states as "current" relative to consistent cuts, i.e., along the corresponding cut line. Informally the *global state* $S(C)$ of a consistent cut C consists of the local states of all processes taken just after the last event of each process P_i that belongs to C (i.e., the right-most event left to the cut line), or the initial local state if there is no such event. More formally, $S(C)$ is the global state that is reached by successively executing all $e \in C$ in some linear order that is consistent with the causality relation, starting from the initial state. Clearly, causally independent (i.e., concurrent) events can be executed in arbitrary order without affecting the final result $S(C)$.

As the events of an observation occur one after the other, the global state of the computation is evolving over time. Each event occurrence denotes a global state transition. At every point of an observation, the set of events that have been observed so far forms a consistent cut. Hence, every observation induces a totally ordered sequence of consistent global states.

Interestingly, the set of all consistent cuts of a computation together with operations $\cup$ and $\cap$ has the mathematical structure of a lattice [JoZ90, Mat89, Nie81, Win88]. Therefore, a convenient method to graphically represent the consistent cuts of a distributed computation is an N-dimensional *state lattice* [Cha92, CoM91, Mat 89, Pra91] as shown in Figure 14. In our two-dimensional example, each vertical

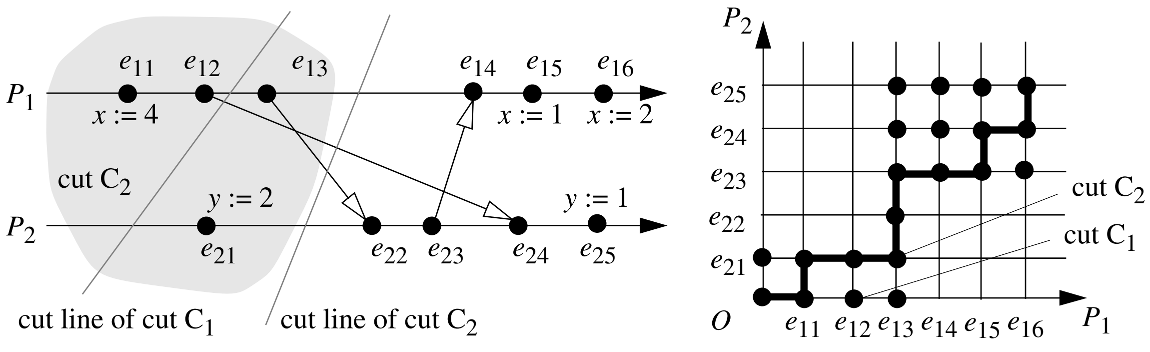

Figure 14: A time diagram, the corresponding state lattice, and a path through that lattice

line of the state lattice corresponds to an event in P_1, and each horizontal line represents an event in P_2. An intersection point $p = [e_{1i}, e_{2j}]$ of two event lines denotes the finite set $\{e_{11},\ldots, e_{1i}, e_{21},\ldots,e_{2j}\} \subseteq E$. Of course, this set is not necessarily a consistent cut. For instance, the point with coordinates $[e_{14}, e_{21}]$ denotes a set which contains the receive event e_{14}, but not the corresponding send event e_{23} preceding it. In the lattice of Figure 14, all intersections denoting consistent cuts — *valid intersection points,* for short — are marked by dots. (Note that a zero coordinate of an intersection point does not correspond to an event; one may, however, postulate a dummy event e_{k0} for that purpose.) In general, a distributed computation comprising *N* processes is represented by an *N*-dimensional state lattice. The intersection points corresponding to an observed event sequence form a *path* [KaP90] in the lattice diagram:

Definition 6.3 *Let L be the state lattice of a distributed computation comprising N processes, and let C(p) denote the consistent cut that corresponds to a valid intersection point p.*

(1) *A sequence $p_0, p_1, p_2,\ldots$ of valid intersection points is called a* path *through L, if $C(p_0) \subset C(p_1) \subset C(p_2)\ldots$, and if $|C(p_i)| = i$ for all p_i contained in the sequence.*

(2) *A path is called* complete*, if for all $e \in E$ some valid intersection point p_i is contained in that path such that $e \in C(p_i)$.*

As an example, Figure 14 shows a complete path which induces the sequence of consistent cuts $\varnothing$, $\{e_{11}\}$, $\{e_{11}, e_{21}\}$, $\{e_{11}, e_{21}, e_{12},\}$,…, $\{e_{11}, e_{21},\ldots, e_{16}, e_{25}\} = E$. Note that every complete path through the state lattice induces a sequence of events $e_1, e_2, e_3,\ldots$, defined by $e_i = C(p_i) \setminus C(p_{i\text{-}1})$; for the path shown in Figure 14 we obtain $e_{11}, e_{21}, e_{12},\ldots, e_{25}$. Obviously, this sequence defines a total order on E which is consistent with causality. Furthermore, it is fair in the sense that every event $e \in E$ has only a finite number of predecessors with respect to that order. Hence, it satisfies Definition 6.1, and it follows that *every complete path corresponds to an observation (and vice versa).*

Recall that every consistent global state corresponds to a consistent cut $C(p)$ for some valid intersection point *p*, and that there always exists some complete path containing *p*. In other words, every consistent global state is observed by at least one observation (as we would expect). However, a single complete path does typically not contain *all* valid intersection points of the state lattice. That is, a single observation will only reveal a *subset* of all possible global states. As a consequence, two observers of the same distributed computation may observe different sets of consistent global states. For example, state S_1 corresponding to cut line C_1 in Figure 14 occurs in observation $O_1 = \mathrm{e}_{11}, \mathrm{e}_{12}, \mathrm{e}_{13},\ldots$, but obviously not in observation $O_2 = \mathrm{e}_{11}, \mathrm{e}_{21}, \mathrm{e}_{12},\ldots$ corresponding to the path marked in the Figure. This has serious consequences.

Suppose, for example, that two observers simultaneously observe the computation shown in Figure 14 to find out whether a given predicate Φ defined on the consistent global states of that computation is satisfied or not. Assume further that Φ holds only for state S_1, but not for any other possible state. Now, if the first observer makes observation O_1, and the second observer makes observation O_2, then the first observer will conclude that Φ holds, while the second will claim that the computation failed to satisfy Φ. Which observation is "correct"?

Obviously, none of them! The decision whether a global state predicate Φ is satisfied in the course of a distributed computation depends on the specific observation that it refers to. Therefore, an accurate description of a predicate's occurrence should be stated as follows: Predicate Φ holds for the set of observations $\{O_1,\ldots, O_k\}$ of the given computation. That is, a specification of Φ should comprise a qualifier denoting the set of observations that it covers. The fact that in distributed systems the truth of a global predicate depends on the observer might be surprising at the first sight — it is, in fact, a phenomenon that is unknown in the sequential world where a computation has only a single valid observation and the validity of a predicate can thus safely be attributed to the computation.

6.2 Possibly and Definitely

In the previous section, we showed that the specification of a global state predicate is generally meaningless as long as it does not refer to a well-defined set of observations. In [CoM91], Cooper and Marzullo address this issue, and they introduce two useful predicate qualifiers, defined as follows:

Definition 6.4 *Let Φ denote a predicate defined on the global states of a distributed computation, let L denote the state lattice of that computation, and let "*Φ holds at *p" mean that Φ holds for the consistent state corresponding to intersection point p of lattice L.*

(1) possibly Φ *holds iff there exists a path P through L and an intersection point p on P such that Φ holds at p.*

(2) definitely Φ *holds iff every complete path through L contains an intersection point p such that Φ holds at p.*

That is, *possibly* Φ holds for a given computation if there exists *at least one* observation which reveals the satisfaction of Φ, and *definitely* Φ holds if *all* observations observe that Φ holds. Note that *definitely* Φ implies *possibly* Φ (we may safely assume that the set of observations is not empty), and that $\neg$ *possibly* ($\neg$ Φ) implies *definitely* Φ. Note further that the term "definitely" is somewhat misleading, as it only refers to all observations of *one* particular computation, but — due to possible nondeterminism — not to *all* computations of a distributed algorithm.

Cooper's and Marzullo's predicate qualifiers are closely related to some modalities known from modal and temporal logic [MaP92]. For example, there is a direct correspondence between the two qualifiers *possibly* and *definitely* and the sequence quantifiers *EF* and *AF* of Katz's and Peled's interleaving set temporal logic [KaP90, KaP92]. Note, however, that we excluded conflicts from our conceptual framework. Therefore, we only deal with a *single* execution $(E, \rightarrow)$ of a distributed system and its possible observations. This differs from the approach generally taken in temporal logic; there, *all* possible executions of a nondeterministic algorithm are considered, and predicates typically contain additional qualifiers denoting the set of executions for which the predicate formula holds.

The predicates *definitely* Φ and *possibly* Φ are properties of a computation which do not depend on a specific observation. Therefore, characterizing a computation by finding out whether certain predicates can *possibly* or will *definitely* hold is an important aspect of distributed debugging. Typically, we would use *definitely* to monitor predicates which specify mandatory states of the computation, for instance: "In each process, the variable *Counter* must eventually decrease to zero"; *possibly* is suitable for the detection of constraint violations like, for example: "More than one traffic light shows 'green' at the same time". Recurrence to the Newtonian model of absolute global time[5] may be helpful to give a pragmatic meaning to *possibly* and *definitely*. Recall that it is generally impossible to decide whether an observation reflects the actual real-time order of event occurrences. Thus, if we assume that the events have an immediate effect on some "global environment" (e.g., traffic lights on the traffic), then it is not clear whether an observed sequence of global states is identical to the one that was actually experienced by the environment. With *possibly* or *definitely*, however, we can simulate an "omniscient" observer by considering all possible observations — i.e., all feasible real-time orders of event occurrences — simultaneously. In our traffic light scenario, for example, most observers may observe a signalling sequence where (just by chance) at most one traffic light shows 'green' at any instant of time, even though some feasible real-time order of events would disclose a lurking bug in the traffic light synchronization.

Interestingly, modal operators like, for example, *possibly* or *definitely* are dispensable for *stable predicates*, as the following lemma shows (see also [KaP90, KaP92, Cha92]). A predicate is called stable if it continues to hold once it has been satisfied.

Lemma 6.5 *For a stable predicate Φ defined on the global states of a distributed computation,* possibly Φ *and* definitely Φ *are equivalent.*

Proof. As remarked above, *definitely* Φ implies *possibly* Φ. Conversely, suppose that *possibly* Φ holds for a given computation. Hence, Φ is satisfied at some intersection point p of some path P, where p corresponds to the consistent cut $C(p)$. Consider an arbitrary complete path P' through the state lattice. As $C(p)$ is finite and P' is complete, there exists a point p' on P' such that $C(p) \subseteq C(p')$. Accord-

5. As opposed to the *relativistic* point of view in modern physics where the existence of absolute time is denied.

ing to Definition 6.2, $C(p)$ is left-closed with respect to the causality relation, hence the events in $C(p') \setminus C(p)$ do not causally precede any event in $C(p)$. Therefore, it is evident that successively removing minimal elements (with respect to $\rightarrow$) from the finite set $C(p') \setminus C(p)$ and adding them to $C(p)$ yields a cut sequence which corresponds to a continuation of path P from p to p'. From the stability of Φ it follows that Φ must hold at p' on P'. As P' was arbitrarily chosen, the same argument holds for any complete path through the lattice, which means that *possibly* Φ implies *definitely* Φ for stable predicates. ❑

Lemma 6.5 shows that a stable property that holds in *some* observation will eventually hold in *any* observation and is thus observer-independent. This fact can easily be understood by considering the N-dimensional state lattice. Obviously, a stable property holds for all valid intersection points in an "upper-right" N-dimensional subcube of the state lattice. If this subcube is not empty, then every complete path must eventually enter that subcube; if it is empty, then neither *possibly* Φ nor *definitely* Φ are satisfied. There exists another class of predicates for which it is possible to generalize from one observer to all observers, namely predicates which depend on a property *local* to a single process [KaP90, KaP92, Cha92]. An in-depth treatment of such "observer-independent" predicates may be found in [Cha92].

Because until recently only the detection of stable predicates was discussed in the literature, Lemma 6.5 might explain why modal operators such as *possibly* or *definitely* were not considered there. It should be noted, however, that detecting *possibly* Φ or *definitely* Φ is quite different from the classical stable predicate detection problem [ChL85]. Whereas in the latter case it is usually *required* that the predicate Φ be stable, and the problem is to detect the satisfaction of Φ as soon as possible in the course of a computation, the problem for *possibly* Φ and *definitely* Φ is to *decide* whether or not a distributed computation has these properties.

In [CoM91], two algorithms based on vector time for the detection of *possibly* Φ and *definitely* Φ in finite computations (i.e., computations where E is finite) are presented[6]. Let us call $|C(p)|$ the *level* of intersection point p. Basically, the algorithm for *definitely* Φ iteratively computes the sets $A_0, A_1, A_2, \ldots$ where A_i denotes the set of valid intersection points at level i, such that all p in A_i are accessible by a path not containing an intersection point at a smaller level that satisfies Φ. A_0 contains the origin of the state lattice; A_{i+1} comprises those valid intersection points p for which there exists an immediate predecessor $p' \in A_i$ along some path (i.e., $C(p') \subseteq C(p)$ and the levels of p and p' differ by 1) such that Φ is not satisfied at p'. If an A_j is reached which is empty, then *definitely* Φ holds. If, however, the maximum level $l = |E|$ of the state lattice is reached and all elements of A_l do still not satisfy Φ (in fact, A_l contains exactly one element), then a path through the state lattice exists such that Φ never holds, and therefore *definitely* Φ is not satisfied.

The algorithm for *possibly* Φ is similar; as soon as A_i contains a member for which Φ holds, *possibly* Φ is satisfied and the algorithm terminates. Otherwise, A_{i+1} is computed as above. Both algorithms are based on an efficient enumeration of the valid intersection points (essentially a breadth-first search through the lattice), thus they are linear in the number of valid intersections. Unfortunately, this can be of order $O(K^N)$, where K is the maximum number of local events per process, and N is the number of processes. The use of vector time and the immense number of valid intersections render an on-the-fly application of the above algorithms almost prohibitive.

As a final remark, it should be noted that *possibly* and *definitely* can be defined without referring to complete paths, i.e., observations. In [Och87], Ochmanski introduces the concept of *inevitable global states* — an equivalent to *definitely* — and extends this notion even to systems for which an observation in the sense of Definition 6.1 does not exist; it is, however, doubtful, whether an efficient algorithm for the detection of *inevitability* in non-observable systems is feasible. Furthermore, such systems seem to be of little practical relevance.

6. For (potentially) infinite computations, *possibly* Φ and *definitely* Φ are undecidable properties because deciding *definitely* ("the computation has terminated") for deterministic programs would mean to solve the halting problem.

6.3 Navigating through the State Lattice

Deciding *definitely* Φ conceptually requires to inspect *all* paths — or at least all consistent global states — of the state lattice in the worst case. It is therefore computationally expensive. While in general the situation for *possibly* Φ is not much better, there exist certain predicates Φ for which *possibly* Φ can be detected quite efficiently. Garg and Waldecker give a more formal characterization of these predicates in [GaW92]; in essence, their definitions comprise global predicates which are decomposable into locally detectable parts — such as conjunctions or disjunctions of local predicates — whose validity can be established in isolation. In the following we restrict our attention to such predicates Φ, and we present a simple algorithm for the detection of *possibly* Φ.

The basic idea is to navigate through the state lattice, searching for an intersection point where Φ holds. For an efficient realization, it is desirable to restrict the search to only one "dimension" of the lattice as long as possible, and to change the direction of search only if absolutely necessary. That is, we execute the events of one specific process until we "hit" a local state that may contribute to the satisfaction of Φ, or until causality constraints force us to interrupt the execution of that particular process; next, we freeze that process' local state and continue with a different process. By executing the computation in such a sequential fashion, we reduce the computational complexity of the detection scheme from $O(K^N)$ to $O(KN)$, or more precisely, to $O(E)$. Approaches similar to the one sketched here are described in [Cha92, GaW92, MaI92].

Executing a computation in the proposed manner is, however, somewhat difficult to achieve in a distributed system where computations are typically nondeterministic. Blocking all processes but one to obtain the required sequential execution would generally cause an unbearable distortion of the system's "normal" behavior. That is, the so-called *probe effect* [Gai86] induced by such a method may lead to a completely abnormal behavior of the system which would render the conclusions drawn from its observation almost irrelevant. One way to overcome the problems induced by observing the processes *during execution* might be to collect event-traces in an otherwise undisturbed run of the system, and to apply the algorithm sketched above *after the execution*. For instance, one could put all traced events in event queues, one for each process, and fetch the next event from the respective queue instead of performing an execution step of a single process. This approach was taken, for example, by Garg and Waldecker [GaW92]. The number of relevant events produced in the course of a distributed computation could be quite substantial, however, and therefore the queues may rapidly grow. Furthermore, in a large distributed system the management of all process queues is likely to become a bottleneck, and tracing all events may already lead to an intolerable probe effect.

In order to avoid undue distortions during the original execution, one solution is to re-execute the computation, and to generate the events "on demand" only *during replay*. Since distributed computations are usually nondeterministic, an identical reproduction of the system's behavior requires special precautions. One might, for example, try to employ a deterministic scheduling discipline to enforce reproducibility of the execution. Unfortunately, centralized scheduling is not appropriate in a distributed setting as it would severely limit the potential for parallelism. Therefore, a better solution is to provide an execution replay facility [LeM87, Leu91]. This mechanism is based on a trace of the outcome of all nondeterministic steps which each process took during an original execution of the distributed system (e.g., the selection of an incoming message, or reading some volatile data). During replay, each process simply consults its trace records whenever a nondeterministic decision has to be taken. Thus, by forcing all processes to reproduce their exact sequence of nondeterministic execution steps, the original behavior of the system — including its communication pattern — is preserved. Tracing only "nondeterministic events" (instead of all events which may affect Φ) diminishes the probe effect, reduces the amount of trace data, and allows to detect global predicates during replay with virtually no (logical) detection delay [MaI92]. Moreover, during replay the execution speed may be reduced in order to match the observer's processing capacity, and on each re-execution the observer may concentrate on particular aspects, thus reducing the space requirements for each analysis.

Execution replay is particularly valuable for the evaluation of global predicates which typically causes substantial overhead in communication and computation. For the subsequent discussion, we will therefore assume that either some kind of deterministic replay of the original computation, or at least a facility for the collection of event traces is available, such that we can safely study the effect of the events of each process in isolation, without changing the observed overall behavior of the system. It should be noted, however, that the problem of replaying distributed computations is difficult in its own right, and may require substantial computational effort. For a more detailed discussion, see [LeM87, Leu91].

To continue our discussion of the navigation scheme sketched above, consider, for example, the distributed computation depicted in Figure 14 and the global predicate $\Phi \equiv ((x = 1) \wedge (y = 1))$, where x and y are local variables of P_1 and P_2, respectively. Note that $x = 1$ and $y = 1$ are two predicates whose truth can be established locally. To detect *possibly* Φ, any assignment to the variables x or y is a significant event that may affect Φ. We propose the following algorithm which detects whether *possibly* Φ holds for such a "locally decomposable" predicate Φ:

(1) Put the system in its initial state.

(2) Check if Φ is satisfied for the current global state. If so, the detection algorithm terminates with *possibly* $\Phi \equiv$ TRUE.

(3) Select some executable process (i.e., a process whose next execution step does not causally depend on the occurrence of a non-local event that has not yet been executed and thus blocks further local execution) according to the following preferences:

  a) select a process that fails to satisfy its local predicate[7], or else

  b) select a process that causes the blocking of some other process.

  If no selectable process exists, the detection algorithm terminates, yielding *possibly* $\Phi \equiv$ FALSE.

(4) Execute the next step of the selected process, and continue until one of the following conditions is met:

  a) P's local predicate holds, or

  b) P becomes blocked at a receive event, waiting for the corresponding send event to occur, or

  c) P terminates.

  In case a), continue with step (2). In case b), continue with step (3). Otherwise (case c), the detection algorithm terminates, yielding *possibly* $\Phi \equiv$ FALSE.

Note that in step (3) of the algorithm, we may be forced to select a process which already satisfies its local predicate. However, should such a situation arise, this means that there exists some process which is currently blocked and still has not reached its local predicate. Under these circumstances, we have no other choice but to continue with the process that causes the blocking, even if the local predicate of that process is then invalidated. The preferences for the selection of an executable process stated in step (3) ensure that the algorithm will detect the satisfaction of *possibly* Φ at the earliest possible "logical moment", i.e., at a minimal consistent cut which satisfies Φ.

Figure 15 illustrates the application of the algorithm for the computation depicted in Figure 14. In terms of the state lattice, we select a path through the lattice such that we move in one dimension (to the right, say) as long as we can, until we find $x = 1$ to hold. Next, we move in an upward direction until $y = 1$ holds. Only if there is no valid intersection in the current direction, then we are forced to circumvent the barrier (i.e., the shaded areas in Figure 15), and change the current direction. The generalization of this method for N-dimensional lattices is straightforward, and interpreting the algorithm as a directed walk through the state lattice guarantees that it is free of cycles and will eventually terminate. It should also be noted that one could easily derive a more sophisticated navigation scheme, where several processes that fail to satisfy their local predicate are executed in parallel. We leave this optimization to the interested reader.

---

7. It is assumed that each process whose local state does not affect the truth of Φ has a local dummy predicate which is always satisfied but does not contribute to the satisfaction of Φ.

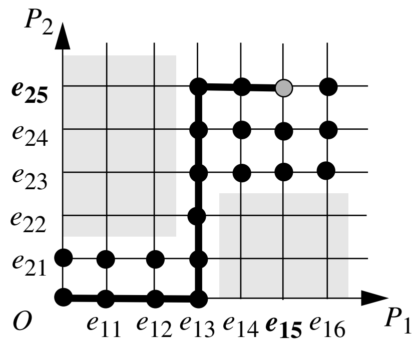

Figure 15: Detecting *possibly* $((x = 1) \wedge (y = 1))$ in the computation of Figure 14

For simple global predicates like the one used in the example above, our algorithm is quite efficient. Provided we are able to execute the computation in the required deterministic fashion, no message time-stamps and no time vectors are needed. Using execution replay, there is no need to explicitly construct the state lattice, which would require too much space in general; the required causality information is implicitly represented by the specific way in which the global state evolves. In the simple example shown in Figure 14 and Figure 15, the navigation algorithm will inevitably lead us to the global state (e_{15}, e_{25}) where Φ holds.

One drawback of the navigation algorithm is that it can only check for one global predicate at a time. Another, more important restriction of this approach is that it will fail for slightly more sophisticated predicates, as shown in Figure 16. Here, each process reaches a local state which may contribute to the desired predicate Φ. However, the local states are contradictory with respect to Φ, i.e., they mutually exclude each other for the predicate to hold. Thus, as soon as we reach $x = 1$ in P_1 and $y = 2$ in P_2, we have to decide which process should be continued next. If we resume P_1, then Φ will never hold, but if we resume P_2, then the predicate is eventually satisfied. The example shows that the simple navigation approach is not generally applicable for the detection of *possibly* Φ for arbitrary global predicates Φ.

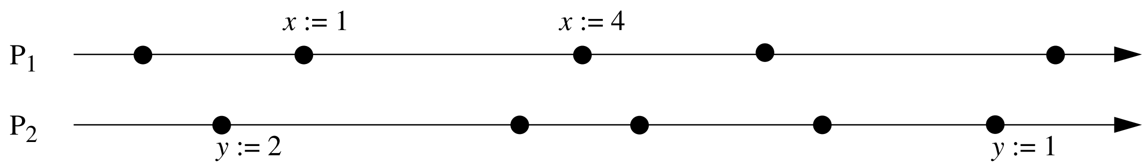

Figure 16: The problem of detecting *possibly* $((x = 1) \wedge (y = 1) \vee (x = 2) \wedge (y = 2))$

6.4 Currently Φ

The discussion in the previous sections revealed that useful modalities such as *possibly* are in general computationally intractable. Except for some special cases where we were able to derive quite efficient detection schemes, we have to resort to the general algorithm proposed by Cooper and Marzullo, as the scenario of Figure 16 indicates. The fact that their algorithm considers all possible observations and may therefore require an immense number of steps rules out the detection of *possibly* Φ in many practically relevant situations. If we consider such intractable predicates, we have to confine ourselves to simpler, although maybe less powerful modalities. In [CoM91], Cooper and Marzullo propose a modality which is based on *a single* observation, the real-time observation of the computation, which we define as follows:

Definition 6.6 *The total order (E, <) of the events of a distributed computation ordered according to* *their real-time occurrence is called the* real-time observation *of the computation.*

Note that the real-time observation is an observation in the sense of Definition 6.1. In Section 6.1, we pointed out that from *within* the system, the real-time observation is indistinguishable from any other observation of the computation; however, as a distributed computation usually affects its global external environment, an *external* observer might nevertheless be able to identify the real-time order of all events. Of course, implementing an external real-time observer might be difficult or even impossible if one has no control over the observed system. It might, on the other hand, be possible that the observer is able to force the system to produce events only in such a way that they are faithfully observable. This, of course, raises the question to what degree such an influence of the observer on the observed system is tolerable — we would certainly not accept a central scheduler that forces a synchronized sequential execution of the computation. In [CoM91], a qualified global predicate called *currently* Φ is defined as follows:

Definition 6.7 *The global predicate* currently Φ *defined on the local process states of a distributed computation is said to hold, if* Φ *is still satisfied at the moment at which it is reported by some dedicated monitoring process.*

Thus, monitoring *currently* Φ addresses both detection accuracy and detection *delay*; it aims at a reliable *on-the-fly* detection of a global predicate Φ, such that the unavoidable notification delay will not allow Φ to vanish before it is recognized. In this respect, *currently* Φ is superior to *possibly* Φ which may be detected long after Φ was first satisfied. To match its intended meaning, *currently* Φ should eventually be satisfied if Φ holds at some global consistent state occurring in the real-time observation of the computation.

Similar to Spezialetti and Kearns mentioned earlier, Cooper and Marzullo define events as state changes that might affect Φ, and they assume a dedicated central monitoring process M which does not participate in the computation, but is only responsible for the predicate detection. In order to detect *currently* Φ — i.e., to simulate a real-time observation — certain processes are temporarily blocked by the monitor M. This may, of course, affect the behavior of the distributed computation. Therefore, the monitor may cause the system to perform a computation that is very unlikely to occur in an unmonitored execution, although the monitor's intrusion will *never* lead to a computation that is not feasible in principle in the unmonitored system. Thus, monitoring will only yield possible, although maybe improbable cases. For application domains like, e.g., debugging, the effects of intrusion are clearly undesirable — they are the price we have to pay for the efficiency of the detection algorithm. In cases, however, where the detection of global states is an integral part of the system (e.g., in distributed reactive systems [HaP85, MaN91] where the system itself is essentially a monitor receiving stimuli from its environment through a network of sensors, and reacting to these stimuli through actuators) a moderate amount of intrusion may be tolerable as long as sufficient potential for concurrency is retained. Cooper's and Marzullo's algorithm for the detection of *currently* Φ can be outlined as follows:

(1) Before the computation starts, the monitor is informed about the initial state of each process.

(2) Whenever a process executes an event e that could make Φ *true*, it (asynchronously) sends the relevant part of its current state to the central monitor. The monitor maintains the latest received state information for each process of the distributed computation. This rule does only apply if e is not an invalidating event, see next rule.

(3) Whenever a process reaches an event e that could make Φ *false* (a so-called *invalidating event*), it first transfers the relevant part of its local state to the monitor and blocks before executing the event. On notification, the monitor then flushes all links from the processes to the monitor, thereby collecting the most recent local states of all processes. This is done by sending a REQ message to all other processes and requiring an immediate ACK. If Φ is not found to hold by the time all replies have returned, then the monitor releases the blocked process by sending an "unblock" message and updates the recorded state of the blocked process to "undefined". "Undefined" signifies that the monitor must not draw any conclusions until it receives new state information from that process. If, however, Φ holds, then *currently* Φ is reported and the algorithm terminates.

(4) On receiving an "unblock" message, a blocked process executes the event it was waiting at and sends the relevant part of its new state to the monitor.

The details of the algorithm may be found in [CoM91]. Interestingly, the monitor M does not actually perform a real-time observation because, according to rule (2), the event notification messages are not necessarily received in real-time order by M. However, by flushing all channels on every invalidating event, the algorithm tries to retain all *essential properties* of a a real-time observation with respect to the detection of Φ.

Note that flushing the communication links with REQ - ACK pairs requires FIFO channels. Apart from the rather high number of control messages, the proposed algorithm is computationally cheap — in particular, it does not require vector time. Figure 17 shows an example of how the algorithm works. The reason why a process is blocked when it tries to (potentially) invalidate Φ is to allow all other processes enough time to send their latest local states to the monitor, thus to enable the detection of a temporary holding of Φ before it can vanish again.

Cooper's and Marzullo's method is highly intrusive and may substantially slow down the distributed computation. The proposed algorithm tries to reduce the monitoring overhead by restricting the blocking of processes to invalidating events. However, if predicates like, for example, $\Phi \equiv (x_1 + x_2 = 5)$ are considered, then each assignment to x_1 or x_2 which changes the value of the variable is an invalidating event. Thus, invalidating events may occur very frequently, causing a lock-stepped execution with many control messages.

Besides these practical issues, there is an even more important conceptual objection to mention. The proposed protocol is incomplete in that it can miss predicates that hold in a true real-time observation of the computation. Figure 18 shows such a case where $\Phi \equiv (x_1 + x_2 \geq 5)$ occurs but is not detected. In the given scenario, P_2 changes its local state to $x_2 = 3$ just after it has replied to a REQ message, but before the invalidating event $x_1{:=}\ 1$ in process P_1 occurs. For the short interval marked on the time line of the monitor the *current state* of the system is $x_2 = 3$ and (still) $x_1 = 2$; hence Φ holds, but is missed because at that time the monitor has recorded "undefined" as the current state of P_1. One could try to "fix" the problem by immediately updating the recorded local state to the new value as soon as the "block" message arrives at the monitor. But then again the algorithm misses certain predicates, as Figure 17 shows, if we replace the initial assignment $x_1 = 2$ in P_1 by $x_1 = 3$.

Obviously, the deficiency remains whether the effect of an invalidating event is defined to occur already *during* or only *after* the interval in which the process is blocked. Avoiding this problem would require to introduce some additional blocking. For example, one could block each process on *every* relevant event — not just invalidating events as suggested in [CoM91]; alternatively, each process could be blocked after sending an ACK message, until Φ has been decided. Both methods would, of course, mean

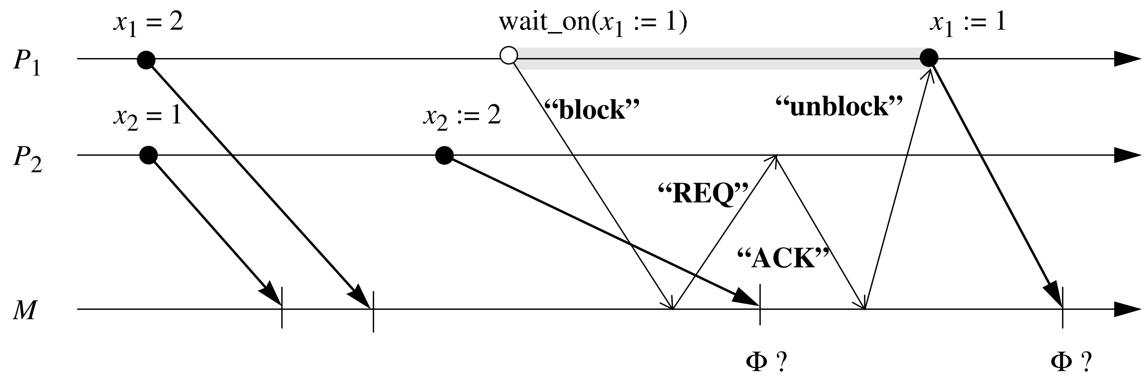

Figure 17: Monitoring *currently* $(x_1 + x_2 \geq 5)$ according to Cooper and Marzullo

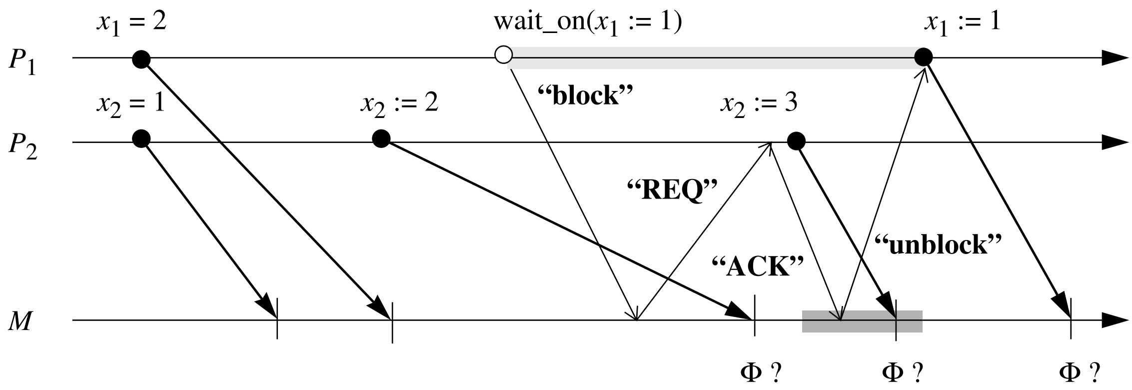

Figure 18: Counter-example: $\Phi \equiv (x_1 + x_2 \geq 5)$ holds but is not detected

to substantially increase the intrusiveness of the algorithm. If such an overhead is unacceptable, we have to pretend that the current state is "undefined" until the next state update is received after unblocking, thus leaving the observer with a "blind spot" for some predicates which occur. This is exactly what Cooper and Marzullo do. Then, however, the notion of a *currently true* predicate becomes rather vague. If *currently* Φ is detected, it is not sure whether Φ (or even *possible* Φ) would have occurred in an *actual* unmonitored computation. If, on the other hand, *currently* Φ is not detected, then it might still be possible that a true real-time observation of the computation would yield the truth of Φ. As a consequence, we do not reach sufficiently trustworthy and meaningful conclusions by trapping predicate occurrences with the proposed algorithm.

In summary, the *currently* qualifier seems only to be appropriate if the underlying system already comprises a (possibly distributed) monitor which is responsible for the collection of global state information, and if the only global states of interest are those seen by this monitoring agent. If, however, our aim is to analyze a given distributed system by making observations with a minimal impact on the system's "natural" behavior, then the detection of *currently* Φ is too intrusive, and the occurrence of *currently* Φ (or the lack thereof) is probably not meaningful. The discussion of *currently* Φ gives further evidence for our claim that in general real time is not appropriate for the analysis of asynchronous distributed systems.

7 Detecting Behavioral Patterns

The approaches for global predicate evaluation discussed so far concentrated on properties of the global state. Alternatively, we may focus our attention on *state transitions* rather than on actual *states*. Recall that every event entails a transition from one global state to another. Thus, by detecting the satisfaction of predicates describing the relative causal order in which certain events occur, we may gain sufficient insight into the resulting system state.

Consider, for example, the distributed traffic lights control system sketched above, and let event e_i denote "light i turns green". If the predicate $\Phi = (e_1 \parallel e_2)$ is satisfied at some instant of time during the execution of the control system, then the system is unsafe because it fails to guarantee mutual exclusion, even if the *actual* global state sequence which is observed by the environment is correct. For many applications — in particular, for the analysis of synchronization in concurrent systems — it suffices to determine the order in which certain events occur in a computation. Detecting such basic patterns of a system's behavior and combining them into high-level abstractions of activity is generally referred to as the *behavioral abstraction* approach [BaW83]. In practice, the detection of *behavioral patterns* and the detection of *global states* can be combined by enriching the events with appropriate local state information which is passed to a central monitor for evaluation. In fact, current approaches typically apply such hybrid techniques [BrH83, MiC88, Pon91].

One important step towards the detection of behavioral patterns appears in [MiC88]. In this seminal paper, Miller and Choi define a class of *distributed predicates*, and they present a detection algorithm for that class. Their work is influenced by the *event description language EDL* proposed by Bates and Wileden [BaW83], but in contrast to EDL Miller's and Choi's specifications do not require global time. Furthermore, not only the relative order, but also the causal relationship between events can be expressed in their formalism. However, conjunction and disjunction operators are restricted to combine *simple predicates* based only on the state local to a single process. So-called *linked predicates* that specify event sequences ordered according to the causality relation can be specified, but concurrency of events cannot be expressed. Thus, their algorithm can only detect a limited class of behavioral specifications. In fairness to their work, it should be noted that Miller's and Choi's approach works on-the-fly and does not require complex mechanisms such as vector time.

Haban and Weigel [HaW88] address the problem of more sophisticated specifications. They aim at the detection of arbitrary causal relations between events, and they assume that vector time is available. Based on some *primitive event specifications* denoting local event classes of a single process, the authors define *global event specifications* recursively as follows:

(1) Every primitive event specification is a global event specification.
(2) If G_1, G_2, G_3 denote global event specifications, then $G_1 \vee G_2$, $G_1 \wedge G_2$, $G_1 \rightarrow G_2$, $G_1 \parallel G_2$, G_1 @ G_2, and @$G_3(G_1, G_2)$ denote global *alternative*, *conjunctive*, *happened-before*, *concurrent*, *negation*, and *between* specifications, respectively.

Note the difference between *event specifications* which denote certain classes of events that may occur repeatedly, and *events* which belong to a specific event class and have a unique occurrence. For the rest of this section, we use capital letters to denote event classes or specifications; if required, several instances of the same event class are distinguished by using *upper* indices, while *lower* indices denote different event classes or specifications.

The satisfaction of a specification is defined recursively. A primitive event specification G is satisfied if a local event e of class G has occurred, and the vector time $V(G) = V(e)$ is assigned to G. A specification $G_1 \vee G_2$ holds as soon as one of its operands is satisfied, and it inherits the timestamp of that operand. $G_1 \wedge G_2$ requires both operands to be satisfied, and inherits the timestamp of the operand most recently detected. $G_1 \parallel G_2$ is treated in the same way, but additionally requires that $V(G_1) \parallel V(G_2)$ holds. Likewise, $G_1 \rightarrow G_2$ requires $V(G_1) \rightarrow V(G_2)$ to hold, and $V(G_1 \rightarrow G_2)$ is defined to be $V(G_2)$. G_1 @ G_2 is satisfied if G_1 holds while G_2 does not; $V(G_1$ @ $G_2)$ inherits the timestamp of G_1. And finally, @$G_3(G_1, G_2)$ requires that $G_1 \rightarrow G_2$ be satisfied, while no G_3 exists which satisfies both $G_1 \rightarrow G_3$ and $G_3 \rightarrow G_2$; the timestamp inherited is that of G_2.

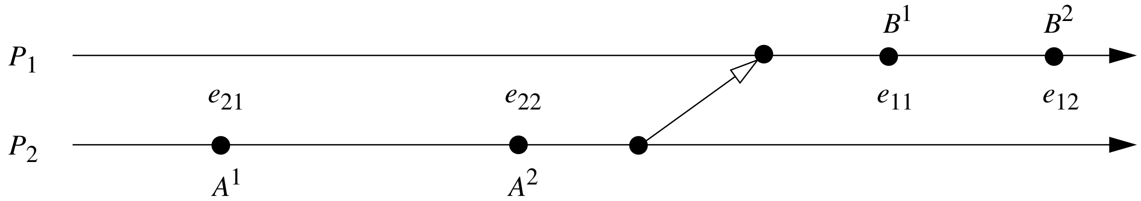

Figure 19: Single detection of the global event specification $(A \rightarrow B)$

The details of the detection scheme may be found in [HaW88]. An important aspect of the algorithm is its requirement that each event may contribute to a global event specification at most once; i.e., as soon as an event has been used to satisfy part of a specification, this event is *consumed* with respect to that specification. It may, of course, contribute to the satisfaction of several, distinct specifications. This rule has a pragmatic background: It reduces the number of event occurrences that have to be stored, and it prevents the detection of "redundant" event occurrences, as Figure 19 demonstrates, where only a *single* occurrence of $A \rightarrow B$ is detected (i.e., $A^2 \rightarrow B^1$ as soon as B^1 occurs) instead of all four combinations of

$A^i \rightarrow B^j$ that hold.

Unfortunately, by consuming events and also by allowing the *absence* of events to denote a global event occurrence (i.e., by introducing the negation operator), the meaning of certain syntactically valid specifications is defined in a counterintuitive way, as is shown in Figure 20, where the processes P_1 and P_2 are observed by the monitor M. According to the definition given in [HaW88], the specification $@B(C, D)$, which reads “there is no event of type B between an event of type C and an event of type D”, is satisfied, because the first occurrence C^1 of C is concealed by the second, C^2. Nevertheless, C^1 and D form an interval such that $C^1 \rightarrow B$ and $B \rightarrow D$ hold, and therefore $@B(C, D)$ should rather *not* hold. The example shows that the pragmatic decision to consume events may lead to situations where crucial events are simply ignored.

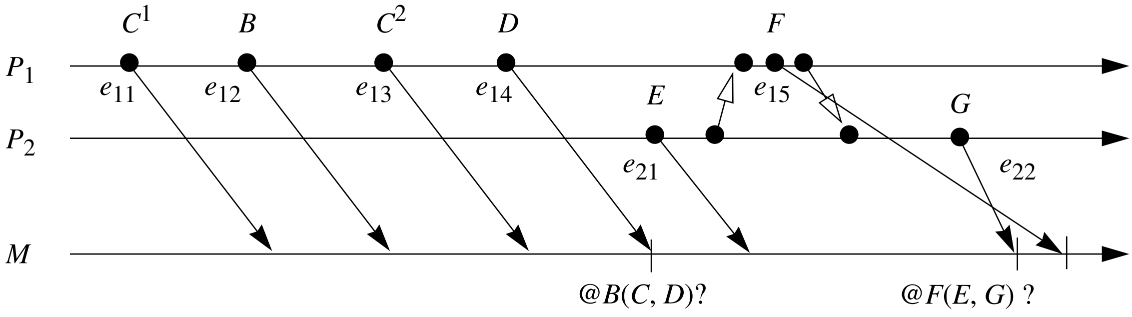

Figure 20: $@B(C, D)$ and $@F(E, G)$ are detected, but do they really hold?

The transmission delays between the local processes and the event monitor raise another problem. Consider, for example, the specification $@F(E, G)$ in the scenario of Figure 20. Here, $E \rightarrow F \rightarrow G$ holds (hence, the specification is not satisfied), but the *notification* of the event monitor M about the occurrence of F suffers from a significant delay such that a direct transmission from P_1 to M takes longer than a transmission from P_1 to M via P_2, violating the so-called triangle inequality [Pra92]. Thus, as soon as $E \rightarrow G$ is detected — and F is not detected in between — the monitor will reach the conclusion that $@F(E, G)$ holds, which is, of course, wrong.

Fortunately, this problem can be avoided by using a *causal order delivery protocol* to inform the monitor about event occurrences. That is, if a notification about the occurrence of event e reaches the monitor M, it must only be delivered after the notifications about all events belonging to the causal history $C(e)$ have been delivered. In Figure 20, for example, the monitor should delay the delivery of G until F — which clearly belongs to $C(G)$ — has been delivered. Causal delivery order at M guarantees that M has always a consistent view of the global state [SaS92], i.e., that the sequence of observed events is a linear extension of the causality relation. There is a straightforward protocol which implements causal delivery order [KeK89, Sch89, Ray91]:

(1) For each process P_i, M maintains a counter *observed*[i], initialized to 0.

(2) On receiving a notification message $m = (e, i)$ indicating the occurrence of event e at process P_i with vector timestamp $V(e)$, the delivery of m is delayed until m becomes deliverable.

(3) $m = (e, i)$ is deliverable iff (*observed*[i] $= V(e)[i] - 1$) and (*observed*[j] $\geq V(e)[j]$ for all $j \neq i$).

(4) If $m = (e, i)$ becomes deliverable, it is actually delivered and *observed*[i] := *observed*[i] + 1.

To understand why this algorithm is correct recall that according to Observation 2.3, the vector timestamp $V(e)$ is just a shorthand notation for the causal history $C(e)$ of event e. The vector *observed* maintained by the above algorithm counts the number of events at each process which have been observed so far. What step (3) essentially requires is that all events of process P_i locally preceding e have already been observed (thus implementing the FIFO property for notification messages), and that at least those non-local events at P_j belonging to the causal history of e (i.e., $e_{j1}, \ldots, e_{jV(e)[j]}$) have already been

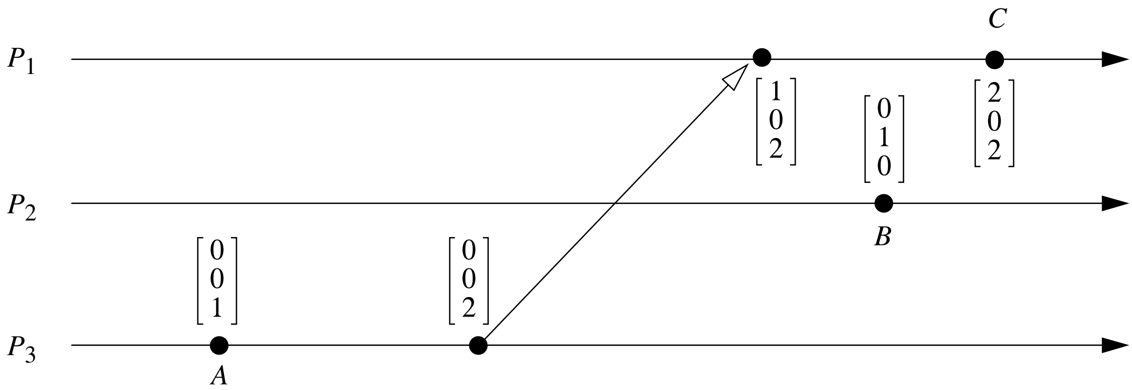

Figure 21: Is $(A \parallel B) \rightarrow C$ satisfied, or $A \rightarrow (B \parallel C)$?

observed, too. It follows by induction that this delivery rule ensures causal delivery order. It is also easy to see that eventually every notification message becomes deliverable. Thus, by adding this algorithm to the protocol described in [HaW88], the satisfaction of specifications like @F(E, G) in Figure 20 can indeed be correctly detected.

There is a general problem with the occurrence of global events which are non-atomic — when, exactly, does such an event "happen"? That is, what is the appropriate (logical) timestamp that should be assigned to its occurrence? Consider, for example, Figure 21, and the specification $(A \parallel B) \rightarrow C$. Is it satisfied? If we suppose that B is detected later than A, then — according to the definition given in [HaW88] — it is not because the subexpression $(A \parallel B)$ inherits the timestamp $V(B)$, which means that $V(A \parallel B) < V(C)$ does not hold as required. If, however, B is detected before A, then $V(A \parallel B) = \mathrm{V}(A)$, and $(A \parallel B) \rightarrow C$ is satisfied.

To exclude such ambiguities, Haban et al. [Hab91] revised the original definitions of [HaW88]. In particular, they define $V(A \parallel B) = \sup\{V(A), V(B)\}$ which means that $(A \parallel B) \rightarrow C$ does not hold in Figure 21 — regardless of the order in which A and B are detected. In this new version, however, the definition lacks symmetry, because now $A \rightarrow (B \parallel C)$ holds in the example above, whereas $(A \parallel B) \rightarrow C$ does not. Intuitively, the meaning of $(A \parallel B) \rightarrow C$ should probably be defined as $(A \parallel B) \wedge ((A \rightarrow C) \vee (B \rightarrow C))$, or maybe $(A \parallel B) \wedge (A \rightarrow C) \wedge (B \rightarrow C)$. Unfortunately, it is non-trivial to extend such definitions to arbitrary compound specifications in a meaningful way, and the resulting specifications tend to become rather bulky.

The reason why timestamp inheritance yields counterintuitive semantics in some cases is because global specifications do generally not describe *atomic* events; rather, they denote activities which have a non-zero duration, as illustrated in Figure 21. Assigning a *single* timestamp to a behavioral pattern essentially means to deny its non-atomic nature. Consequently, the timestamps of *all* subexpressions should in some way or the other affect the satisfaction of a global specification. This, however, rules out simple timestamp inheritance such as those considered above.

The question of how to specify the occurrence of non-atomic events is addressed by Fidge in [Fid89, Fid89b]. Like Haban and Weigel, he aims at the detection of significant global state changes which are characterized by specifications based on local (i.e., primitive) predicate expressions. However, instead of assigning unique time instants to event specifications as in [HaW88], he proposes to determine appropriate *state intervals* instead. More specifically, two events s and t are assigned to the occurrence of a primitive specification G (i.e., to a local state change of a single process which causes the satisfaction of G); s denotes the event that leads to the satisfaction of G, and t denotes the next local event that invalidates G again. G is said to be *satisfied in the interval* $I(G) = [s, t]$. Following this approach, $[s, t]$ can safely be regarded as denoting an interval of time — namely, $[V(s), V(t)]$ — during which G is satisfied. Note that the intervals belonging to primitive specifications are strictly local to one process. It is straightforward to define the following relations between local intervals $I_1 = [s, t]$ and $I_2 = [u, v]$:

- I_1 *precedes* $I_2 \equiv t \to u$
- I_1 *includes* $I_2 \equiv s \to u \wedge v \to t$
- I_1 and I_2 *may overlap* $\equiv \neg$ (I_1 *precedes* I_2) $\wedge \neg$ (I_2 *precedes* I_1)

Based on these relations, and given two primitive specifications G_1 and G_2 with corresponding intervals $I(G_1)$ and $I(G_2)$ during which the respective specifications are satisfied, we may now define $G_1 \to G_2 \equiv$ ($I(G_1)$ *precedes* $I(G_2)$), and $G_1 \parallel G_2 \equiv$ ($I(G_1)$ and $I(G_2)$ *may overlap*). Similar definitions for $G_1 \wedge G_2$ as well as for $G_1 \vee G_2$ are feasible as long as G_1 and G_2 are primitive specifications local to the same process. The details may be found in [Fid89, Fid89b].

Unfortunately, it is rather difficult — if not impossible — to extend the relations between primitive specifications to arbitrary global specifications in a sensible way. In particular, assigning meaningful intervals to compound specifications is an open problem. There is, for example, no obvious choice for the interval which should be assigned to $G_1 \to G_2$, given that $I(G_1)$ and $I(G_2)$ are known local intervals. Note, for instance, that the "natural" choice for $I(G_1 \to G_2)$ — the interval formed by the lower bound of $I(G_1)$ and the upper bound of $I(G_2)$ — may comprise logical time instants (between the upper bound of $I(G_1)$ and the lower bound of $I(G_2)$) at which neither G_1 nor G_2 holds, which is different from what one would typically expect. Another problem occurs if a compound specification leads to interval fragmentation. Consider, for instance, the specification $G_1 \wedge \neg G_2$, in a situation where G_1 and G_2 are local to the same process, with $I(G_1)$ including $I(G_2)$. Under these circumstances, one would expect that $I(G_1 \wedge \neg G_2)$ denotes not a unique interval, but should in general rather comprises *two* intervals, both of which are contained in $I(G_1)$, adjacent to $I(G_2)$. But what if $I(G_1 \wedge \neg G_2)$ actually denotes two intervals and occurs as a subexpression of a more complex specification? These examples show that it is generally impossible to reasonably combine primitive specifications in order to obtain more general global specifications. It seems that such problems are inherent to all specifications based on atomic event occurrences, even if time intervals instead of time instants are used.

Another approach to behavioral pattern detection is due to Hseush and Kaiser. They propose a formalism called *data path expressions* [HsK90] which bears a strong resemblance to Haban's and Weigel's global event specifications, but avoids most of their problematical aspects — in particular, negation (like, e.g., Haban's and Weigel's @ operator) is excluded. (Negation is problematic because it is often difficult or even impossible to define when exactly a negated event first "occurs", in particular, if upper bounds for transmission delays are not known.) Basically, data path expressions extend generalized path expressions [BrH83] with a concurrency operator such that both causal dependence and causal independence between event occurrences can be expressed. However, instead of the "→" operator used by Haban and Weigel, only the weaker *sequencing operator* ";" is provided, with "A; B" defined as "A is an *immediate* causal predecessor of B". As an example, Haban's and Weigel's specification ($A \to B \to C$) corresponds to the equivalent data path expression "A; $(A \vee C)^*$; B; $(A \vee B)^*$; C", where "X^*" denotes zero or more occurrences of subexpression X. Note that the transitive closure implicit in the causality relation must be explicitly stated by the data path expression. Consequently, global event specifications are more compact than their data path equivalent; an automated conversion from the former to the latter is, of course, feasible as long as a specification does not contain the negation operator. Dealing only with event sequences prevents the need for interval specifications as have been proposed by Fidge.

For a given data path expression, Hseush and Kaiser construct an equivalent *predecessor automaton* which is able to recognize that expression. Predecessor automata are similar to, but extend the concept of finite-state automata. In [HsK90], a rule set for recursively transforming data path expressions into their recognizing automata is presented. For brevity, we do not further discuss the concept of predecessor automata and the recognition process. It should be noted, however, that predecessor automata can become quite complex; for example, if we have recognizers for the data path expressions X and Y which comprise n (or m, respectively) internal states, then the predecessor automaton recognizing

"*X concurrent Y*" requires $n \times m$ internal states. Ponamgi et al. have implemented a debugging tool for multithreaded programs based on data path expressions and predecessor automata [Pon91]. However, their prototype tool lacks support for a more convenient specification of high-level patterns of behavior. In [Pon91], it is also noted that an automatic reduction of data path expressions would be desirable in order to obtain more compact predecessor automata, thus making recognition more efficient. This is particularly important because the "$\rightarrow$" operator is not supported directly, but has to be converted into a series of sequencing operators, yielding rather complex expressions.

It should be noted that the scheme proposed by Hseush and Kaiser does not require vector time. In particular, there is no need to assign time instants (or time intervals) to each specification as is required by Haban and Weigel or Fidge. Furthermore, expressing causal dependence with only the sequencing operator avoids ambiguities like the one depicted in Figure 19 (note that "A; $(A \vee C)^*$; B" is matched exactly once by "A^1; A^2; B^1"), so there is no need for event consumption. And as Hseush and Kaiser exclude negation from their formalism, they preventing the potential problems depicted in Figure 20. And finally, defining "$(A \parallel B)$; C" to require both "A; C" and "B; C" yields a symmetrical solution for the situation shown in Figure 21.

As a final remark, it should be noted that the detection of behavioral patterns requires an *anticipation* of the system's behavior, i.e., a pattern must be specified in advance to be observable. Unexpected behavior — even if it has the same effect on the global state as the expected one — is not captured by the observation. Thus, only selected aspects of the complex causality structure are revealed. This is quite different from the general global predicate detection techniques described in previous sections.

8 Conclusions

Distributed programs are difficult to develop and to analyze. This is due to their inherent characteristics such as parallelism, nondeterminism, and the unavailability of global state and global time. The fact that these aspects have still not been completely mastered at the conceptual level is one of the reasons for the lack of adequate tools for the design and analysis of distributed systems. However, distributed computing is almost ubiquitous today. Thus, there is an urgent demand for more powerful and more sophisticated programming environments which are able to overcome the problems arising from distribution in order to exploit its potential benefits like increased speed, availability, and reliability. Much work has been dedicated to this issue, but surprisingly little has been achieved so far.

As we tried to show in this paper, the lack of practical realizations of adequate tools is — among other reasons — due to the fact that we still lack appropriate methods to deal with the complex causality structure of distributed programs which is the key to understanding their behavior. Fortunately, it seems that the situation is improving now. At least from a theoretical point of view, causality in distributed computations is being increasingly well understood. It is now widely accepted that the traditional Newtonian model of distributed computations, which is based on the notion of absolute global time, is insufficient to reflect the relativistic aspects of systems which are asynchronous, physically distributed, and suffer from noticeable communication delays [Pra86]. The partial order semantics of distributed computations expressed by the "happened before" relation [Lam78] — as opposed to the traditional interleaving semantics where an underlying total order of event occurrences (i.e., the "real-time order") is implicitly assumed — triggered off major progress in the theory of distributed computing. As a result, different types of logical clocks were proposed to capture some notion of causality, culminating in the advent of vector time and a general definition of consistent global states. Unfortunately, the theoretical insight into the relativistic nature of distributed computations failed to entail a corresponding stimulus on the development of actual tools. This reluctance to apply the new findings has several reasons.

First of all, there exists no well-established, agreed-upon formalism for reasoning about causality in distributed systems, and the system models found in literature often lack conciseness and differ substantially. This *"Babel of languages"* impedes the exchange of knowledge and experience, and makes published results difficult to assess. As a consequence, the relativistic nature of the compound system formed by the observer and the observed is not yet sufficiently understood by the computing community. Therefore, many approaches suffer from slight misconceptions. For example, it is often not taken into account that different observers typically observe different global states of the system; states and state transitions, or atomic and non-atomic events are often confused, causing severe shortcomings. A second reason for the lack of suitable tools is the complexity inherent to the causality structure, which leads to tool designs dominated by efficiency considerations. In the past, this prevented, for example, the widespread use of vector time, and provoked dubious "optimizations" like monitoring the absence of events, or consuming event occurrences. Finally — and maybe most importantly — the theoretical insights gained so far are almost discouraging. The intricacy of distributed computations exceeded common expectations. For example, there seems to be no representation of causality more compact than vector time. Instead of simplifying matters, the known results rather seem to muddy the waters, and the lack of global control in distributed computations, the inherent nondeterminism preventing their reproducibility, as well as the overwhelming amount of information which is essential for their analysis further exacerbates the problems. Thus, current experience confirms our claim that distributed programming is still an art rather than a well-established technique.

Our discussion indicates several possible directions of future work. One aim could be a relaxation of the causality relation. Recall that $\rightarrow$ indicates *potential*, but not *actual* causal relationships. Events occurring at the same process, for example, are totally ordered by the causality relation, although some of them are presumably not causally related. Therefore, Ahuja et al. [Ahu91] propose a timestamping scheme which reflects *actual* causality — or *essential causality*, as it is called in [KaP90] — more accurately. The benefits of this approach are questionable, though, for two reasons. Firstly, from a theoretical point of view it can be argued that the local event order in each process — although not causally enforced — is in fact total; it is uniquely determined by the local thread of control, and thus not subject to relativistic effects. That is, *all* observers will agree on this order. Hence, there seems to be little need to determine alternative local event orderings. Secondly, our discussion showed that in practice even the more "unsophisticated" causality relation as defined in Definition 1.1 causes immense problems, the situation is even worse for actual causality.

In contrast to these considerations, one might try to find a timestamping scheme which yields a partial event order somewhat stricter than the order induced by vector time, but which relaxes the (total) order of Definition 3.6 derived from Lamport time. The aim is to trade accuracy for ease of computation. In [DiJ92], Diehl and Jard propose *interval orders* [Fis85] as a means to obtain event timestamps of pairs of integers with relatively little computational effort. If the causal structure of a distributed computation is in fact that of an interval order, then their scheme yields timestamps which actually characterize causality. In general, however, this condition is not satisfied. Nevertheless, it might be fruitful to develop new programming paradigms which induce causal orders that are guaranteed to be as easy to handle as, for example, interval orders.

A different approach is pursued by Meldal et al. in [Mel91]. Like Diehl and Jard, they aim at a more efficient computation of the causality relation by restricting the problem domain. Their work is based on the observation that for some applications causal relationships are only of interest for messages that are sent to the *same* destination process; furthermore, communication paths are often static and known at compile time. Thus, by exploiting the logical structure of the computation, and also the physical structure of the network on which the computation is executed, substantial savings in communication cost and storage requirements are achievable. Only part of the causality information is required because some *dynamic* causal relationships can be inferred from the given *static* structures, while others are known to be irrelevant for the matter at hand. The feasibility of this technique depends, however, on the particular system under consideration.

In this paper, we surveyed some representative approaches to the problem of determining causal relationships in distributed computations. The discussion shed some light on the main problems and some fundamental limitations arising in this research area. We saw that none of the presented schemes is sufficiently mature to serve as a general-purpose mechanism for the analysis of causality. Ideally, a tool should combine the speed and reliability of automated detection with the human intuition and flexibility. The problem of anticipating the relevant behavior, assigning meaningful semantics to general global predicates, and finding correct and efficient algorithms for their detection, remains to be a challenge.

It seems that distributed computations are intrinsically difficult to understand, and perhaps a simple way to describe their behavior does not even exist. Anyhow, the holy grail of causality analysis has not been found yet.

Acknowledgments

The work presented in this paper has been stimulated by many colleagues who provided us with insightful suggestions for improvement and pointers to further relevant work. In particular, we would like to thank the anonymous referees, and also Ken Birman, Bernadette Charron-Bost, Claire Diehl, Stefan Fünfrocken, Claude Jard, Horst Mehl, and Michel Raynal for their valuable comments on earlier versions of this paper.

References

[Aha90] M. Ahamad, P. Hutto, and R. John. "Implementing and Programming Causal Distributed Shared Memory". *Technical Report GIT-CC-90-49, College of Computing, Georgia Institute of Technology, 1990.*

[Ahu91] M. Ahuja, T. Carlson, A. Gahlot, and D. Shands. "Timestamping Events for Inferring 'Affects' Relation and Potential Causality". *Proc. 15th IEEE Int. Computer Software and Application Conference COMPSAC 91, 1991.*

[Bal92] P. Baldy, H. Dicky, R. Medina, M. Morvan, and J.-F. Vilarem. "Efficient Reconstruction of the Causal Relationship in Distributed Computations". *Technical Report 92-013, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, France, June 1992.*

[BaW83] P.C. Bates and J.C. Wileden. "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach". *Journal of Systems and Software, Vol. 4, No. 3, pp. 255-264, Dec. 1983.*

[BiJ87] K. Birman and T. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". *Operating Systems Review, Vol. 22, No. 1, pp. 123-138, Dec. 1987.*

[Bir91] K. Birman, A. Schiper, and P. Stephenson. "Lightweight Causal and Atomic Group Multicast". *ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 272-314, Aug. 1991.*

[Bir91b] K. Birman. "The Process Group Approach to Reliable Distributed Computing". *Technical Report*, *Computer Science Department, Cornell University, Ithaca, New York, July 1991.*

[BrH83] B. Bruegge and P. Hibbard. "Generalized Path Expressions". *The Journal of Systems and Software, Vol. 2, No. 2, pp. 265-276, 1983.*

[Cha89] B. Charron-Bost. "Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory". In *Distributed Algorithms, J.-C. Bermond and M. Raynal (eds.), Springer-Verlag, LNCS 392, pp. 45-56, 1989.*

[Cha91] B. Charron-Bost. "Concerning the Size of Logical Clocks in Distributed Systems". *Information Processing Letters 39, pp. 11-16, July 1991.*

[Cha92] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. "Local and Temporal Predicates in Distributed Systems". *Technical Report, LITP, IBP, Université Paris 7, France, April 1992.*

[CMT92] B. Charron-Bost, F. Mattern, and G. Tel. "Synchronous and Asynchronous Communication in Distributed Computations". *Technical Report LITP 92.77, IBP, Université Paris 7, France, Oct. 1992.*

[ChL85] K.M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems, Vol. 3, No. 1, pp. 63-75, Feb. 1985.*

[CoM91] R. Cooper and K. Marzullo. "Consistent Detection of Global Predicates". *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, pp. 163-173, May 1991.*

[DiJ92] C. Diehl and C. Jard. "Interval Approximations and Message Causality in Distributed Systems". *Proc. of the 9th Annual Symposium on Theoretical Aspects of Computer Science STACS '92, A. Finkel, M. Jantzen (eds.), Springer-Verlag, LNCS 577, pp. 363-374, 1992.*

[Fid88] C.J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering". *Proc. 11th Australian Computer Science Conference, University of Queensland, pp. 55-66, Feb. 1988.*

[Fid89] C.J. Fidge. "Partial Orders for Parallel Debugging". *ACM SIGPLAN Notices, Vol. 24, No. 1, pp. 183-194, Jan. 1989.*

[Fid89b] C.J. Fidge. "Dynamic Analysis of Event Orderings in Message-Passing Systems". *PhD Thesis, Department of Computer Science, Australian National University, March 1989.*

[Fid91] C.J. Fidge. "Logical Time in Distributed Computing Systems". *IEEE Computer, Vol. 24, No. 8, pp. 28-33, Aug. 1991.*

[FiM82] M.J. Fischer and A. Michael. "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network". *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Database Systems, pp. 70-75, 1982.*

[Fis85] P.C. Fishburn. "Interval Orders and Interval Graphs". *Wiley, New York, 1985.*

[Flo62] R.W. Floyd. "Algorithm 97, Shortest Path". *Communications of the ACM, Vol. 5, p. 345, 1962.*

[FoZ90] J. Fowler and W. Zwaenepoel. "Causal Distributed Breakpoints". *Proc. 10th Int. Conference on Distributed Computing Systems, Paris, France, pp. 134-141, May 1990.*

[Gai86] J. Gait. "A Probe Effect in Concurrent Programs". *Software - Practice and Experience, Vol. 16, No. 3, pp. 225-233, March 1986.*

[GaW92] V.K. Garg and B. Waldecker. "Detection of Unstable Predicates in Distributed Programs". *Technical Report TR-92-07-82, Computer and Vision Research Center, University of Texas at Austin, March 1992.*

[Hab91] D. Haban, S. Zhou, D. Maurer, and R. Wilhelm. "Specification and Detection of Global Breakpoints in Distributed Systems". *Technical Report SFB124 - 08/1991, Universität des Saarlandes, Saarbrücken, Germany, April 1991.*

[HaP85] D. Harel and A. Pnueli. "On the Development of Reactive Systems". In *Logics and Models of Concurrent Systems, K. Apt (ed.), Springer-Verlag, NATO ASI Series F, Vol. 13, pp. 477-498, 1985.*

[HaW88] D. Haban and W. Weigel. "Global Events and Global Breakpoints in Distributed Systems". *Proc. 21th Annual Hawaii Int. Conference on System Sciences, pp. 166-175, Jan. 1988.*

[HsK90] W. Hseush and G.E. Kaiser. "Modeling Concurrency in Parallel Debugging". *ACM SIGPLAN Notices, Vol. 25, No. 3, pp. 11-20, March 1990.*

[HuA90] P. Hutto and M. Ahamad. "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories". *Proc. 10th Int. Conference on Distributed Computing Systems, Paris, France, pp. 302-309, May 1990.*

[JoZ90] D.B. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing". *Journal of Algorithms, Vol. 11, No. 3, pp. 462-491, Sep. 1990.*

[KaP90] S. Katz and D. Peled. "Interleaving Set Temporal Logic". *Theoretical Computer Science 75, pp. 263-287, 1990.*

[KaP92] S. Katz and D. Peled. "Verification of Distributed Programs Using Representative Interleaving Sequences". *Distributed Computing, Vol. 6, pp. 107-120, 1992.*

[KeK89] J.P. Kearns and B. Koodalattupuram. "Immediate Ordered Service in Distributed Systems". *Proc. 9th Int. Conference on Distributed Computing Systems, Newport Beach, California, pp. 611-618, June 1989.*

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978.*

[LeM87] T.J. LeBlanc and J.M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay". *IEEE Transactions on Computers, Vol. 36, No. 4, pp. 471-482, April 1987.*

[Leu91] E. Leu, A. Schiper, and A. Zramdini. "Efficient Execution Replay for Distributed Memory Architectures". *Proc. 2nd European Distributed Memory Computing Conference, Munich, Germany, A. Bode (ed.), Springer-Verlag, LNCS 487, pp. 315-324, April 1991.*

[MaI92] Y. Manabe and M. Imase. "Global Conditions in Debugging Distributed Programs". *Journal of Parallel and Distributed Computing, No. 15, pp. 62-69, 1992.*

[MaN91] K. Marzullo and G. Neiger. "Detection of Global State Predicates". *Proc. 5th Workshop on Distributed Algorithms (WDAG-91), Delphi, Greece, S.Toueg, P.G. Spirakis, L. Kirousis (eds.), Springer-Verlag, LNCS 579, pp. 254-272, Oct. 1991.*

[MaP92] Z. Manna and A. Pnueli. "The Temporal Logic of Reactive and Concurrent Systems". *Springer-Verlag, 1992.*

[Mat87] F. Mattern. "Algorithms for Distributed Termination Detection". *Distributed Computing, Vol. 2, pp. 161-175, 1987.*

[Mat89] F. Mattern. "Virtual Time and Global States in Distributed Systems". *Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, Oct. 1988, M. Cosnard et al. (eds.), Elsevier / North Holland, pp. 215-226, 1989.*

[Mat93] F. Mattern. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation". *Journal of Parallel and Distributed Computing, to appear, 1993.*

[Mel91] S. Meldal, S. Sankar, and J. Vera. "Exploiting Locality in Maintaining Potential Causality". *Proc. 10th Annual*

ACM Symposium on Principles of Distributed Computing, Montreal, Canada, pp. 231-239, 1991.

[MiC88] B.P. Miller and J.-D. Choi. “Breakpoints and Halting in Distributed Programs”. *Proc. 8th Int. Conference on Distributed Computing Systems, pp. 316-323, June 1988.*

[Nie81] M. Nielsen, G. Plotkin, and G. Winskel. “Petri Nets, Event Structures and Domains, Part I”. *Theoretical Computer Science, Vol. 13, pp. 85-108, 1981.*

[Och87] E. Ochmanski. “Inevitability in Concurrent Systems”. *Information Processing Letters 25, pp. 221-225, June 1987.*

[PaT92] P. Panangaden and K. Taylor. “Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems”. *Distributed Computing, Vol. 6, No. 2, pp. 73-93, 1992.*

[Pon91] M.K. Ponamgi, W. Hseush, and G.E. Kaiser. “Debugging Multithreaded Programs with MPD”. *IEEE Software, pp. 37-43, May 1991.*

[Pra86] V. Pratt. “Modeling Concurrency with Partial Orders”. *Int. Journal of Parallel Programming, Vol. 15, No. 1, pp. 33-71, Feb. 1986.*

[Pra91] V. Pratt. “Modeling Concurrency with Geometry”. *Proc. 18th Annual Symposium on Principles of Programming Languages (POPL-91), pp. 311-322, Jan. 1991.*

[Pra92] V. Pratt. “Arithmetic + Logic + Geometry = Concurrency”. *Proc. LATIN’92, I. Simon (ed), Springer-Verlag, LNCS 583, pp. 430-447, 1992.*

[Ray91] M. Raynal, A. Schiper, and S. Toueg. “The Causal Ordering Abstraction and a Simple Way to Implement it”. *Information Processing Letters 39, pp. 343-350, Sep. 1991.*

[Rei88] W. Reisig. “Temporal Logic and Causality in Concurrent Systems”. *Proc. Concurrency’88, F.H. Vogt (ed.), Springer-Verlag, LNCS 335, pp. 121-139, 1988.*

[Rei91] W. Reisig. “Parallel Composition of Liveness”. *Technical Report SFB342/30/91A, Technische Universität München, Germany, Oct. 1991.*

[Ren92] R. Renesse. “Causal Controversy at the 5th ACM SIGOPS Workshop 1992 (Le Mont St.-Michel, France)”. *Submitted to ACM Operating Systems Review, 1992.*

[SaS92] A. Sandoz and A. Schiper. “A Characterization of Consistent Distributed Snapshots Using Causal Order”. *Technical Report 92-14, Département d’Informatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland, Oct. 1992.*

[Sch89] A. Schiper, J. Eggli, and A. Sandoz. “A New Algorithm to Implement Causal Ordering”. *Proc. Workshop on Distributed Algorithms, Nice, France, J.-C. Bermond and M. Raynal (eds.), Springer-Verlag, LNCS 392, pp. 219-232, 1989.*

[SiK92] M. Singhal and A. Kshemkalyani. “An Efficient Implementation of Vector Clocks”. *Information Processing Letters 43, pp. 47-52, Aug. 1992.*

[SpK89] M. Spezialetti and J.P. Kearns. “Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Systems”. *Proc. 9th Int. Conference on Distributed Computing Systems, Newport Beach, California, pp. 61-68, June 1989.*

[Sto87] J.M. Stone. “Visualizing Concurrent Processes”. *Technical Report RC 12973, IBM T.J. Watson Research Center, July 1987.*

[Sto88] J.M. Stone. “Debugging Concurrent Processes: A Case Study”. *Proc. SIGPLAN Conf. on Programming Language Design and Implementation, Atlanta, Georgia, pp. 145-153, June 1988.*

[Sto89] J.M. Stone. “A Graphical Representation of Concurrent Processes”. *ACM SIGPLAN Notices, Vol. 24, No. 1, pp. 226-235, Jan. 1989.*

[StY85] R. Strom and S. Yemini. “Optimistic Recovery in Distributed Systems”. *ACM Transactions on Computer Systems, Vol. 3, No. 3, pp. 204-226, 1985.*

[Szp30] E. Szpilrajn. “Sur l’extension de l’ordre partiel”. *Fund. Math. 16, pp. 386-389, 1930.*

[War62] S. Warshall. “A Theorem on Boolean Matrices”. *Journal of the ACM, Vol. 9, pp. 11-12, 1962.*

[Win88] G. Winskel. “An Introduction to Event Structures”. *Proc. Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, The Netherlands, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (eds.), Springer-Verlag, LNCS 354, pp 364-397, 1988.*

[WuB84] G.T.J. Wuu and A.J. Bernstein. “Efficient Solutions to the Replicated Log and Dictionary Problems”. *Proc. ACM Symposium on Principles of Distributed Computing, pp. 233-242, 1984.*