Vector Time and Causality among Abstract Events in Distributed Computations *

Twan Basten[1], Thomas Kunz[2], James P. Black[2], Michael H. Coffin[2], and David J. Taylor[2]

[1] Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands

[2] Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

Abstract

An important problem in analyzing distributed computations is the amount of information. In event-based models, even for simple applications, the number of events is large and the causal structure is complex. Event abstraction can be used to reduce the apparent complexity of a distributed computation.

This paper discusses one important aspect of event abstraction: causality among abstract events. Following Lamport [24], two causality relations are defined on abstract events, called weak and strong precedence. A general theoretical framework based on logical vector time is developed in which several meaningful timestamps for abstract events are derived. These timestamps can be used to efficiently determine causal relationships between arbitrary abstract events. The class of *convex* abstract events is identified as a subclass of abstract events that is general enough to be widely applicable and restricted enough to simplify timestamping schemes used for characterizing weak precedence. We explain why such a simplification seems not possible for strong precedence.

Key words: Distributed systems – Event abstraction – Causality – Precedence relation – Partial order – Vector time – Logical time

1 Introduction

A distributed application consists of a number of autonomous sequential processes, cooperating to achieve a common goal. Cooperation includes both communication and synchronization, and is achieved by exchanging messages. A distributed computation is modeled as a set of *events*. An event represents some activity performed by some process and is considered to take place at an instant in time. Typically, the lowest-level observable events, or *primitive* events, are computations local to processes and interprocess-communication events.

What is important in an event-based view of distributed computations is how events are causally related to each other. Causality can be expressed in terms of precedence. Sending a message, for example, always precedes receiving the message. However, sending a message might be unrelated to a write action on a local file in another process. Neither event precedes the other and they are said to be concurrent. In [23], Lamport argues that causality among primitive events is a *partial order*.

To determine causal relationships between events, logical-timestamp schemes have been proposed [16, 23, 27]. Logical time has been used for many different purposes: implementing causal broadcasts [9], measuring concurrency [10], detecting global predicates [14, 26], implementing distributed breakpoints [4, 19], computing consistent global snapshots [27], and visualizing program

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

behavior [32]. A good starting point for an introduction to several of these issues that play an important role in distributed computing is [3].

Experience shows that even for simple distributed applications, the amount of behavioral information is very large, and the causality structure is very complex. It is well known that human beings have difficulties managing too much information at once. Therefore, in analyzing distributed applications, it is desirable to reduce the amount of information that must be considered at a single point in time and, thus, to reduce the apparent complexity of a computation. A powerful way to achieve such a reduction is *abstraction.*

This paper focuses on one type of abstraction, namely *event abstraction.* Primitive events are grouped together into high-level abstract events, hiding their internal structure and creating an abstract view of the computation. Given a hierarchy of abstract views of program behavior, a distributed application can be analyzed at different levels of abstraction. As Schwarz and Mattern have observed [30], to date, there has been no sound treatment in the literature of causality and logical time for arbitrary abstract events. Therefore, in this paper, we study vector time, which is one particular type of logical time, and causality among abstract events. Our goal is to present a general theoretical framework that is useful for a wide variety of applications using vector time. Following Lamport [24], two precedence relations on abstract events are defined, called *weak* and *strong* precedence. Together they capture all important aspects of causality among abstract events. The main contribution of this paper is that timestamp schemes and accompanying precedence tests are derived to efficiently determine causal relationships between abstract events. Each timestamp scheme is formally proven correct. Using timestamps, program behavior can be visualized at any level of abstraction while faithfully depicting causal relations between abstract events.

The main motivation for this paper comes from the area of analyzing and debugging distributed programs, in particular, visualizing abstract views of distributed computations. Although we do not think that the theoretical framework presented in this paper is restricted to this particular application, Section 2 discusses some results achieved in this area to show the practical applicability of the theory.

The remainder of the paper is organized as follows. Section 3 presents a formal model of distributed computations and recalls some basic definitions and results about logical vector time. Section 4 discusses causality among abstract events. The weak and strong precedence relations on abstract events are defined. Section 5 explains the basic issues that are important when timestamping abstract events. In general, at least two timestamps are necessary to characterize precedence relations between arbitrary abstract events. In Section 6, two timestamps and two precedence tests that characterize strong precedence among arbitrary abstract events are derived. Section 7 gives timestamps and precedence tests for determining weak precedence relations among abstract events. Section 8 deals with an important subclass of abstract events, called convex abstract events. It is shown that for this class of abstract events, a single timestamp is sufficient to characterize weak precedence relations. Unfortunately, no such result seems to exist for the strong precedence relation. Finally, Section 9 summarizes the results.

2 Motivating Example

The research presented in this paper originated in the area of monitoring, analyzing, and debugging distributed applications, in particular, the visualization of the causality structure of a distributed computation. For this purpose, an *event-based* representation of distributed computations is most convenient, whereas for other purposes, such as for example distributed-predicate detection, a *state-based* representation is more appropriate. We do not discuss the advantages and disadvantages of both representations in detail. In [17], timestamps and causality relations are defined for a state-based representation of computations. The two representations yield very similar formulas. The results presented in this paper can easily be translated to a state-based representation of distributed computations.

Our approach to debugging and analyzing distributed applications is one of post-mortem analysis, which conceptually consists of the following three steps. First, a minimum amount of event information is collected with the least possible perturbation of the distributed computation. Second, vector timestamps for the collected events are calculated separately. Finally, the causal relationships between events in the computation can be visualized and analyzed. This approach guarantees that the program behavior is influenced as little as possible by the monitoring and analysis process. In the actual debugger [32], steps are not as clearly separated as described above. Only timestamps needed for visualizing the part of the computation under consideration are calculated. A checkpoint mechanism is used to allow a fast reconstruction of timestamps for other parts of the computation when needed.

One of the main features of the debugger is that it allows the user to construct abstract visualizations of program behavior consisting of abstract events and causal relations between these events. For this purpose, it is important to have a faithful representation of causal relations among abstract events as well as an efficient way of determining and visualizing such relations. The remainder of this paper studies these two aspects in more detail.

First, however, we show the abstract visualization of a small distributed computation to motivate the use of event abstraction and to show an actual implementation of the results presented in this paper. The computation visualized here is an execution of the `boundedbuffer` application described in the Hermes tutorial [31]. It implements a simple bounded buffer for text strings and conceptually consists of two processes. Process `boundedbuffer` implements the bounded buffer, and process `bbintf` provides a line-oriented user interface to the bounded buffer. In the sample computation, one string is put into the buffer, fetched from the buffer and displayed, after which the computation is terminated. During the execution, four additional processes are created and a number of processes within the Hermes runtime system are used. The execution creates an event file containing 1874 primitive events from a total of 126 processes. (The first 120 processes form the standard Hermes runtime system.)

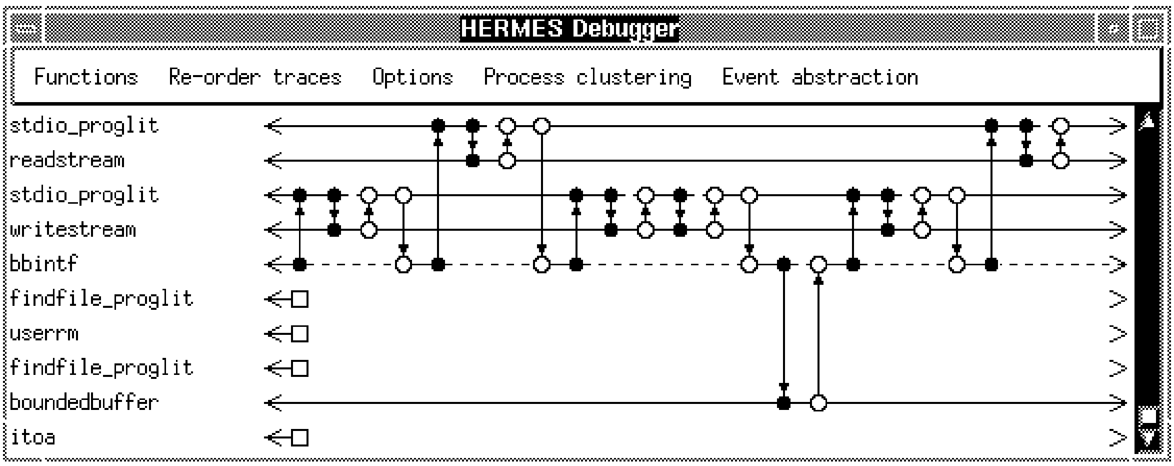

Figure 1: The `boundedbuffer` computation.

Figure 1 depicts part of the sample computation, using the display provided by the original Hermes debugger [32]. The display is similar to a standard process-time diagram with slightly more information about an execution, such as event types (indicated by the symbol used to draw a primitive event) and an approximation of the process states (indicated by the line style). Since even for this small example the numbers of processes and events are already very large, Figure 1 shows only a subset of all processes and events, namely that part of the computation where a string is entered and put into the buffer.

Figure 2 shows a visualization of the computation that starts with the same events as the ones shown in Figure 1 at an intermediate event-abstraction level. In general, abstract events contain primitive events from different processes. We therefore chose the following representation to visualize abstract events. An abstract event is depicted by an open, vertical rectangle, stretching over the range of all processes involved. The intersection of this rectangle with a process is drawn as a filled square if primitive events from this process are constituents of the abstract event (see also [21]).

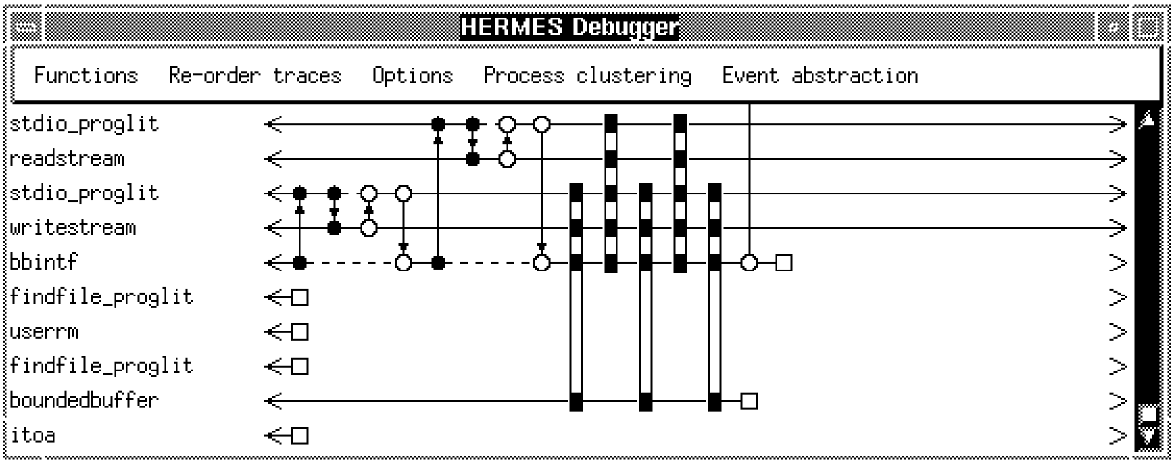

Figure 2: An intermediate event-abstraction view of the `boundedbuffer` computation.

The first abstract event corresponds to the action "put a string into the buffer." The second abstract event corresponds to getting the next command from the user. The third abstract event fetches a string from the buffer and displays it. The fourth obtains the next user command, and the fifth abstract event contains the termination activities. Because of the use of event abstraction, Figure 2 depicts a much larger sequence of the execution history than Figure 1. Moreover, it depicts the computation in meaningful "units of work," facilitating understanding the program behavior.

In Figure 2, two abstract events are connected if and only if the leftmost event *weakly* precedes the rightmost event. As explained in more detail in Section 4, this means that *part* of the first abstract event causally affects *part* of the second event. In our opinion, such causal relationships are very important in debugging distributed applications, which is confirmed by, for example, [20]. The abstract visualization shown in Figure 2 is built using the timestamp scheme of Section 8.

The above example was deliberately kept simple. However, we have traced distributed applications that generate many thousands of events and built event-abstraction hierarchies with many hundreds of abstract events. We are currently in the process of applying the visualization tool to long-running distributed applications, exploring issues such as managing the growth of the trace files. A more detailed explanation of the current state of the implementation of our debugger for distributed programs can be found in [22].

3 Basic Definitions and Results

In this paper, a *distributed system* is a collection of many loosely-coupled machines. These machines do not share any system resources and are only connected by a communication network. Communication channels may be lossy and delivery order may or may not be guaranteed.

A *distributed application* is a set of independent, cooperating program modules. Information is exchanged only by message passing. Both synchronous and asynchronous communication are allowed. It is assumed that communication is point-to-point. However, it is straightforward to extend the results to multicast and broadcast schemes.

At runtime, program modules are instantiated as *processes* which do not share memory. For the sake of simplicity, it is assumed that the number of processes is fixed and known in advance. Each process performs a *local computation*. A *distributed computation* is the collection of all local computations.

3.1 Distributed Computations

The model of distributed computations used in this paper is based on the notion of *primitive events*. Primitive events are considered to be atomic. Therefore, a primitive event is modeled as if it occurred instantaneously. A distributed computation is a pair $(E, \prec)$, where E is the set of primitive events and $\prec$ is an irreflexive partial order on the set of events that models causal precedence.

The detailed model given in the remainder of this subsection is based mainly on previous definitions of Mattern [27, 28], Charron-Bost [10, 11, 12], and Fidge [16]. Their definitions are, in turn, based on the "happens before" relation introduced by Lamport [23].

The set of primitive events E is the union of N mutually disjoint sets of events, $E_0, \ldots, E_{N-1}$, where N is the number of processes. Each of these sets represents a local computation. It is assumed that E is *finite*. This is not a real restriction since this paper discusses timestamp schemes and in practice, only finite (prefixes of) computations can be timestamped. The set of process identifiers, $\{0, \ldots, N-1\}$, is denoted $\mathcal{P}$.

As mentioned, both synchronous and asynchronous communication are allowed. Every communication is modeled by a send event and a corresponding receive event. The sets of send and receive events are denoted by $\mathcal{S}$ and $\mathcal{R}$ respectively. The two sets are disjoint subsets of the set of events E. A relation $\Gamma \subseteq \mathcal{S} \times \mathcal{R}$ relates send events to receive events. This relation is left- and right-unique. Furthermore, it is required that for every receive event in $\mathcal{R}$, there is a corresponding send event in $\mathcal{S}$. The absence of the converse condition means that messages might be lost or might still be in transit. A subset of Γ, Γ_s, denotes the set of synchronous message communications.

For every $i \in \mathcal{P}$, the set E_i is totally ordered by a relation $\prec_i$. This models the fact that processes are sequential. The relation $\prec_l$ is defined as the union of all $\prec_i$. It expresses the local ordering of events. The precedence relation $\prec$ that models the causal ordering of events is defined as the smallest transitive relation that satisfies the following two conditions.

C1 The relation $\prec_l \cup \Gamma$ is a subset of $\prec$.

C2 For every $(s, r) \in \Gamma_s$ and $e \in E \setminus \{s, r\}$, $e \prec s \Leftrightarrow e \prec r$ and $s \prec e \Leftrightarrow r \prec e$.

The definitions given so far model a distributed computation if and only if the precedence relation is an irreflexive partial order.

The relation $\prec$ extends the "happens before" relation as defined by Lamport [23] to synchronous communication in a natural way. Condition **C2**, originally given by Fidge [16], means that a synchronous communication can be interpreted as if it occurred atomically. That is, no other events can occur causally between the two events participating in a synchronous communication. Distinguishing a send and a receive event such that the send event precedes the corresponding receive conforms to physical reality: a synchronous communication is initiated by one process and received by the other after a small but non-zero delay. We prefer this model of synchronous communication over another model of synchronous communication in the literature [12, 13, 16], which models a synchronous communication as two unrelated events; this model has the disadvantage that a synchronous communication cannot be distinguished from a pair of concurrent events.

Let $\preceq$ denote the reflexive closure of the precedence relation $\prec$. The relation $\preceq$ can be used to express concurrency among events. Two events $e_0, e_1 \in E$ are concurrent if and only if $e_0 \not\preceq e_1$ and $e_1 \not\preceq e_0$. That is, two events are concurrent if and only if they are unrelated by (the reflexive closure of) the precedence relation.

Using the definitions above, it is possible to formalize the notion of *cuts*. A cut is the event-based equivalent of a global state. Formalizing the notion of cuts is useful to better understand the causality structure of a distributed computation. The following definitions and theorems are due to Mattern [27].

Definition 3.1. (Cut) A set $C \subseteq E$ is called a *cut* of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \prec_l e_0 \Rightarrow e_1 \in C$. A cut is said to be *left-closed* under $\prec_l$. The set of all cuts is denoted by $C_{\prec_l}$.

Theorem 3.2. (Structure of cuts) *The set of all cuts of a distributed computation, with the ordering defined by the subset relation $\subseteq$, is a complete lattice. The infimum and supremum of sets of cuts are defined by set intersection and set union respectively.*

In distributed computing, the subset of *consistent* cuts is of particular interest. Consistent cuts characterize the set of global states that might actually occur during a distributed computation.

Definition 3.3. (Consistent cut) A set $C \subseteq E$ is called a *consistent cut* of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \prec e_0 \Rightarrow e_1 \in C$. A consistent cut is left-closed under $\prec$. The set of all consistent cuts of a distributed computation is denoted by $C_{\prec}$.

Theorem 3.4. (Structure of consistent cuts) *The set of consistent cuts, with the ordering defined by $\subseteq$, is a complete lattice.*

3.2 Vector Time

For many applications in distributed computing, it is useful to have a characterization of causality. Since the precedence relation is a partial order, it is not possible to use physical time or any other totally ordered set as a characterization. For this reason, Mattern [27] and Fidge [16] independently introduced partially ordered vector time. Vector time extends the idea of logical clocks introduced by Lamport [23].

In this subsection, we summarize some definitions and results given by Mattern [27, 28] and Schwarz and Mattern [30]. (Note that [28] is a revised version of [27]. The reason for mentioning it here is its clear presentation; it also contains some results which are not present in [27].) The definitions and results given in this subsection form the theoretical framework which is needed to prove the correctness of the timestamp schemes and accompanying precedence tests for abstract events given in Sections 6 through 8.

The intuition of vector timestamps can be best explained using the reflexive variant of the precedence relation $\preceq$. An event e_0 causally precedes another event e_1 if and only if all predecessors of e_0

are also predecessors of e_1, where predecessors are defined by means of $\preceq$. That is, e_0 precedes e_1 if and only if the cut containing all predecessors of e_0 is a subset of the cut containing all predecessors of e_1. The idea behind timestamps is to associate with each event e a value $T.e$, the *timestamp* of e, and, in addition, to define a relation $\leq$ on timestamps in such a way as to ensure that for any $e_0, e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0 \leq T.e_1$. The intent is to make $\leq$ relatively inexpensive to calculate, thus avoiding expensive set-inclusion calculations. Figure 3 illustrates this interpretation of precedence between two events. Definitions of the function $\mathbf{p}$, which defines the cut containing all predecessors of an event, and T, the timestamp of an event, are given below. Event e_0 precedes event e_1 since all the predecessors of e_0 are also predecessors of e_1.

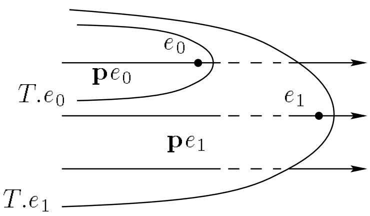

Figure 3: Precedence between primitive events.

Definition 3.5. (Causal past [10, 28, 30]**)** The function $\mathbf{p}. : E \longrightarrow 2^E$ defines the causal past of an event as follows. For any $e \in E$, $\mathbf{p}e = \{e_0 \in E \mid e_0 \preceq e\}$. Note that $\mathbf{p}e$ is a consistent cut.

Definition 3.6. (Strict causal past) The function $\mathbf{p}^-. : E \longrightarrow 2^E$ defines the *strict* causal past of an event. For any $e \in E$, $\mathbf{p}^- e = \{e_0 \in E \mid e_0 \prec e\}$. Note that $\mathbf{p}^- e = \mathbf{p}e \setminus \{e\}$.

The causal past in some process i of an event is the set of all its predecessors in i.

Definition 3.7. (Causal past in a process [10]**)** For any $i \in \mathcal{P}$, the function $\mathbf{p}_i. : E \longrightarrow 2^{E_i}$ defines the causal past in process i of an event. For any $e \in E$, $\mathbf{p}_i e = \mathbf{p}e \cap E_i = \{e_0 \in E_i \mid e_0 \preceq e\}$.

Definition 3.8. (Strict causal past in a process) For any $i \in \mathcal{P}$, the function $\mathbf{p}_i^-. : E \longrightarrow 2^{E_i}$ defines the strict causal past in process i of an event. For any $e \in E$, $\mathbf{p}_i^- e = \mathbf{p}^- e \cap E_i = \{e_0 \in E_i \mid e_0 \prec e\}$. Note that $\mathbf{p}_i^- e = \mathbf{p}_i e \setminus \{e\}$; if $e \notin E_i$, then $\mathbf{p}_i^- e = \mathbf{p}_i e$.

The following corollaries are a direct result of the above definitions.

Corollary 3.9. *For any events $e_0, e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow \mathbf{p}e_0 \subseteq \mathbf{p}e_1$.*

Corollary 3.10. *For any events $e_0, e_1 \in E$, $e_0 \prec e_1 \Leftrightarrow \mathbf{p}e_0 \subseteq \mathbf{p}^- e_1 \Leftrightarrow \mathbf{p}e_0 \subset \mathbf{p}e_1$.*

Corollary 3.11. *For any process $i \in \mathcal{P}$ and events $e_0 \in E_i$ and $e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow \mathbf{p}_i e_0 \subseteq \mathbf{p}_i e_1$.*

Corollary 3.12. *For any process $i \in \mathcal{P}$ and events $e_0 \in E_i$ and $e_1 \in E$, $e_0 \prec e_1 \Leftrightarrow \mathbf{p}_i e_0 \subseteq \mathbf{p}_i^- e_1$.*

Observe that Corollary 3.12 does not have a counterpart in terms of only the causal past in a process, as Corollary 3.10 has. For any process $i \in \mathcal{P}$ and events e_0 and e_1 as above, $e_0 \prec e_1 \not\Rightarrow \mathbf{p}_i e_0 \subset \mathbf{p}_i e_1$. It is easy to choose e_0 and e_1 such that $\mathbf{p}_i e_0 = \mathbf{p}_i e_1$.

The introduction of the causal past is sufficient to formalize the notion of vector timestamps. A vector timestamp of size N is assigned to every event such that component $i \in \mathcal{P}$ of the timestamp is equal to the number of predecessors of the event in process i.

Definition 3.13. (Timestamp function [11, 28]**)** The function $T : E \longrightarrow \mathbb{N}^N$ defines a timestamp for every event as follows. For any event $e \in E$ and process $i \in \mathcal{P}$, $T.e.i = |\mathbf{p}_i e|$.

The vector representation of timestamps is possible only because the number of processes is known. However, vector representation of timestamps is not essential. If the number of processes is not known, the timestamp of an event can be defined as a set of pairs, where each pair consists of a process identifier and the corresponding timestamp component [16].

An example of the assignment of vector timestamps to events is given in Figure 4, showing a standard process-time diagram. Horizontal lines represent processes. Time increases from left to right. Events are depicted as dots. Arrows represent the communication relation Γ. A synchronous communication is represented by a vertical arrow, an asynchronous communication by a slanted arrow. Note that in Process P_1, the increment of the third component of the vector timestamp from 0 to 1 and from 1 to 2 is the result of the synchronous communication between P_1 and P_2.

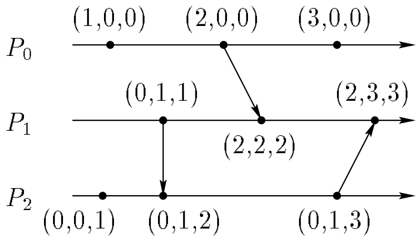

Figure 4: Timestamping events in a distributed computation.

The following two well-known theorems show that timestamps can be used to determine the causal relation between primitive events. For two vectors $t_0, t_1 \in \mathbb{N}^N$, we define $t_0 \leq t_1$ if and only if $t_0.i \leq t_1.i$ for all i, $0 \leq i < N$, and $t_0 < t_1$ if and only if $t_0 \leq t_1$ and $t_0 \neq t_1$.

Theorem 3.14. (Precedence test [27, 28, 30]**)** *For any events* $e_0, e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0 \leq T.e_1$ *and* $e_0 \prec e_1 \Leftrightarrow T.e_0 < T.e_1$.

This precedence test formalizes the visualization of precedence given in Figure 3. It provides an efficient way to determine precedence among primitive events; at most N integer comparisons are necessary. Precedence can be determined even more efficiently if it is known in which process an event occurs.

Theorem 3.15. (Precedence test [27, 30]**)** *For any* $i \in \mathcal{P}$ *and events* $e_0 \in E_i$ *and* $e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0.i \leq T.e_1.i$.

This theorem shows that only one integer comparison is needed to decide whether an event precedes another event if the process in which the event occurs is known. For the example of Figure 4, the validity of the precedence tests given in Theorems 3.14 and 3.15 is easily checked.

Theorem 3.15 does not have a counterpart for the irreflexive precedence relation $\prec$. Corollary 3.12 suggests a solution to this problem. We introduce another timestamp function, defined in terms of the *strict* causal past of events.

Definition 3.16. (Timestamp function T^-) The function $T^- : E \longrightarrow \mathbb{N}^N$ defines a timestamp for every event as follows. For any event $e \in E$ and process $i \in \mathcal{P}$, $T^-.e.i = |\mathbf{p}_i^- e|$.

Corollary 3.17. *For any $i, j \in \mathcal{P}$ and any event $e \in E_i$,*

$$T^{\text{-}}.e.j = \begin{cases} T.e.i - 1, & \text{for } j = i \\ T.e.j, & \text{otherwise} \end{cases}$$

This corollary shows that the timestamp $T^{\text{-}}$ of an event can be easily calculated from the timestamp T, *provided* that it is known in which process the event occurs. The introduction of timestamp $T^{\text{-}}$ and Corollary 3.12 yield the following theorem.

Theorem 3.18. (Precedence test) *For any $i \in \mathcal{P}$, $e_0 \in E_i$, and $e_1 \in E$, $e_0 \prec e_1 \Leftrightarrow T.e_0.i \leq T^{\text{-}}.e_1.i$.*

The precedence test in this theorem is a characterization of the *irreflexive* precedence relation such that only one integer comparison is needed to determine whether an event precedes another event. In order to use the test, it is necessary to know in which process events occur. Therefore, by Corollary 3.17, timestamp $T^{\text{-}}$ can be simply calculated from timestamp T; it is not necessary to keep track of two sets of timestamps. We have introduced $T^{\text{-}}$ for the reason of clarity. It proves to be convenient in Section 6. (Note that it is also possible to formulate a variant of the precedence test in Theorem 3.14 in terms of $T^{\text{-}}$. However, such a test is not of any practical use.)

For implementation purposes, it is important to know that vector timestamps can be calculated algorithmically during or after the execution of a distributed program. There exists a well-known, straightforward algorithm based on counters. In [30], Schwarz and Mattern present this algorithm and they discuss techniques to implement vector timestamps efficiently.

In the remainder of this subsection, the notion of global time is formalized. Global-time vectors have a structure that is isomorphic to the structure of cuts.

Definition 3.19. (Global time of a cut [28]**)** Function $\mathcal{T} : C_{\prec_l} \longrightarrow \mathbb{N}^N$ defines the global time of a cut. For any cut C, component i, with $0 \leq i < N$, of the time vector is defined as $\mathcal{T}.C.i = |C \cap E_i|$. The set of global-time vectors of a computation, $\{\mathcal{T}.C \mid C \in C_{\prec_l}\}$, is denoted $T_{\prec_l}$.

The following corollaries follow immediately from the definitions given so far. Corollary 3.20 states that the timestamp of an event reflects its causal past (see Figure 3).

Corollary 3.20. [28] *For any event $e \in E$, $\mathcal{T}.\mathbf{p}e = T.e$.*

Corollary 3.21. *For any event $e \in E$, $\mathcal{T}.\mathbf{p}^{\text{-}}e = T^{\text{-}}.e$.*

Function $\mathcal{T}$ is an isomorphism between the two lattices $(C_{\prec_l}, \subseteq)$ and $(T_{\prec_l}, \leq)$, i.e., for any cuts $C_0, C_1 \in C_{\prec_l}$, $C_0 \subseteq C_1 \Leftrightarrow \mathcal{T}.C_0 \leq \mathcal{T}.C_1$. This yields the following theorems.

Theorem 3.22. (Structure of time vectors [28]**)** *The set of time vectors, $T_{\prec_l}$, with the ordering defined by $\leq$, forms a complete lattice. It is isomorphic to the lattice $(C_{\prec_l}, \subseteq)$.*

Theorem 3.23. (Structure of consistent time vectors [27]**)** *The set of* consistent *time vectors $T_{\prec} = \{\mathcal{T}.C \mid C \in C_{\prec}\}$, with the ordering $\leq$, forms a complete lattice. It is isomorphic to $(C_{\prec}, \subseteq)$.*

The last two results are established by defining an isomorphism between the lattices of cuts and time vectors. The infimum and supremum of a set of global-time vectors corresponding to a set C of cuts, are therefore implicitly defined as $\mathcal{T}.(\cap\, c :\ c \in C :\ c)$ and $\mathcal{T}.(\cup\, c :\ c \in C :\ c)$. Let the quantifier SUP (and the corresponding binary operator "sup") on time vectors be defined as the componentwise maximum, and the quantifier INF (and binary operator "inf") as the componentwise minimum. It follows from Definitions 3.1 (Cut) and 3.19 (Global time) that $\mathcal{T}.(\cap\, c :\ c \in C :\ c) = (\text{INF}\ c :\ c \in C :\ \mathcal{T}.c)$ and $\mathcal{T}.(\cup\, c :\ c \in C :\ c) = (\text{SUP}\ c :\ c \in C :\ \mathcal{T}.c)$. In other words, the infimum and supremum of sets of time vectors are defined by INF and SUP respectively.

4 Abstract Events and Causality

An abstract description of program behavior is a set of abstract events plus a characterization of causality among abstract events. In a hierarchy of such abstract descriptions, an abstract event is described uniquely by its constituents in the previous level. However, to avoid recursive definitions and inductive proofs, in the following, abstract events are represented by non-empty sets of primitive events. Obviously, the latter representation can always be derived from the former.

Definition 4.1. (Abstract event) An abstract event is a non-empty set of primitive events.

The causality structure in an abstract description of program behavior is defined in terms of precedence. An important question is what is a meaningful definition of precedence among abstract events. In Section 3.1, where precedence among primitive events has been discussed, we have already seen a characteristic property of a precedence relation, namely that it is a(n irreflexive) *partial order*. This implies that the precedence relation has the desirable properties of anti-symmetry (or asymmetry) and transitivity. It also provides a very natural way to express concurrency among events. Two events are concurrent if and only if they are unrelated by the precedence relation. Intuitively, a precedence relation on abstract events should also have these properties. An important observation when trying to define causality among abstract events is that, as opposed to primitive events, abstract events are no longer atomic. Abstract events are composed of primitive events (or lower-level abstract events).

It seems natural to define causality among abstract events in terms of the causal relations between their primitive events. Let us return to the intuition behind the precedence relation on primitive events. One plausible interpretation of this relation is that an event precedes another event if and only if the latter causally depends on the completion of the former. In terms of abstract events this can be phrased as follows. An abstract event precedes another abstract event if and only if *all* the primitive events in the first abstract event precede *all* the primitive events in the other one. Only then is it guaranteed that the second abstract event cannot start before the first one is completed. This leads to the following definition of a precedence relation on abstract events, first defined by Lamport in [24].

Definition 4.2. (Strong precedence relation on abstract events) For any abstract events A and B, $A \prec B \Leftrightarrow (\forall\, a : a \in A : (\forall\, b : b \in B : a \prec b))$.

Property 4.3. *The strong precedence relation is an irreflexive partial order on abstract events.*

Proof. It is easy to verify that the strong precedence relation is irreflexive and transitive. □

Note that it is essential that the above definition is formulated in terms of the irreflexive precedence relation on primitive events. If it had been defined in terms of the reflexive precedence relation, the relation would no longer be irreflexive, which can be seen by considering a singleton abstract event. Also note that no matter what variant of the precedence relation on primitive events is used, the strong precedence relation is not reflexive.

At a first glance, the strong precedence relation might seem to satisfy the properties of the precedence relation on primitive events mentioned above. However, this is not the case. It does not conform to the intuitive meaning of concurrency. Concurrency between primitive events is defined as their being unrelated by the precedence relation. If a similar definition is used for abstract events, two abstract events can be concurrent while some primitive events in one abstract event precede some primitive events in the other, which is clearly counterintuitive.

This observation inspires another definition of causality among abstract events. An abstract event precedes another abstract event if and only if *some* primitive events in the former precede *some* primitive events in the latter. This definition conforms to an interpretation of the precedence relation on primitive events that is subtly different from the interpretation given above, namely that a primitive event precedes another primitive event if and only if it can causally affect the other event. Seen in the light of our earlier observation that abstract events are no longer atomic, this second definition of causality among abstract events should not come as a surprise. In the words of Lamport [24]:

> "Nonatomicity introduces the possibility that an operation execution A can influence an operation execution B without preceding it; it is necessary only that some action of A precede some action of B. Hence in addition to the precedence relation [...], one needs an additional relation [...] "can affect," where A can affect B means that some action of A precedes some action of B."

Definition 4.4. (Weak precedence relation on abstract events) For any abstract events A and B, $A \rightarrow B \Leftrightarrow (\exists\, a : a \in A : (\exists\, b : b \in B : a \preceq b))$.

Note that the weak precedence relation allows a natural definition of concurrency among abstract events: Two abstract events are concurrent if and only if they are unrelated by the weak precedence relation. Only then is there absolutely no causal relation between two concurrent abstract events. Note that this would not be true if weak precedence had been defined in terms of the irreflexive variant of the precedence relation on primitive events. Unfortunately, the weak precedence relation also has a drawback. Figure 5 shows that the weak precedence relation is neither an irreflexive partial order nor a reflexive one. In Figure 5(a), $A \rightarrow B$ and $B \rightarrow A$, which is a violation of the asymmetry requirement of an irreflexive partial order. Note that the asymmetry requirement is a direct consequence of the irreflexivity and transitivity requirements. Since clearly A is not equal to B, the fact that $A \rightarrow B$ and $B \rightarrow A$ also contradicts the anti-symmetry requirement of a reflexive partial order. In Figure 5(b), $A \rightarrow B$ and $B \rightarrow C$. However, we do not have $A \rightarrow C$, which implies that the weak precedence relation is also not transitive.

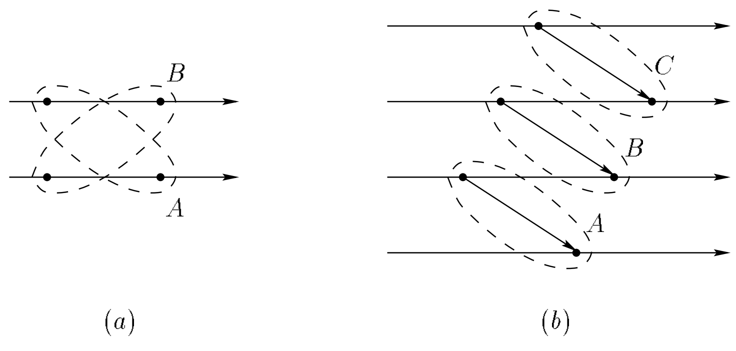

Figure 5: The weak precedence relation is not a partial order on abstract events: (a) A violation of the asymmetry resp. anti-symmetry requirement; (b) a violation of the transitivity requirement.

Although the strong and weak precedence relations each have shortcomings, their combination seems to be a good characterization of precedence among abstract events. It is possible to express that an abstract event as a whole precedes another abstract event. It is possible to express that part of an abstract event precedes part of another abstract event. Finally, concurrency among abstract events can be expressed in a natural way. The conclusion that weak and strong precedence are meaningful indeed is supported by the fact that (variants of) both weak and strong precedence appear in many places in the literature. As already mentioned, the definitions given above are taken from the work of Lamport [24, 25], in which an extensive motivation for both relations is given. A variant of the weak precedence relation, applied in the context of debugging distributed programs, appears in [20]. As already mentioned in Section 2, we also believe that weak precedence plays an important role in this area. Another variant of the weak precedence relation appears in the area of distributed databases [29], where abstract events correspond to transactions. Restricted versions of both weak and strong precedence formulated in terms of states instead of events appear in [15, 17, 18]. They are restricted in the sense that, in the terminology of this paper, each abstract event can contain primitive events from only a single process.

A different approach is taken in [1, 2]. In these papers, the notions of atoms and molecules as abstract events are introduced, as well as precedence relations on such abstract events. Without going into detail, we will explain what we believe to be a serious shortcoming of this approach to event abstraction. Consider the example of Figure 5(b). In the terminology of [1, 2], abstract events A, B, and C are all atoms. Since the precedence relation in [1, 2] is transitive, this implies that A precedes C, although none of the primitive events in A is related to any of the events in C. In our opinion, this is undesirable. An abstract representation of program behavior should not suggest causal relations that are not present when considering a lower level of abstraction. Note that it follows immediately from the definitions given above that both weak and strong precedence satisfy this requirement. The price that we have paid is that weak precedence is not a partial order.

So far, we have only mentioned work that explicitly addresses the question of causality among (relatively) general abstract events. Some papers describing event abstraction simply ignore the issue of causality [6, 7, 19]. Others limit their attention to abstract events with specific structural properties [8, 33]. While this allows them to prove certain desirable properties for their abstract events, it severely limits the modeling power.

5 Timestamping Abstract Events

5.1 Timestamps and Precedence Tests for Abstract Events

This subsection discusses some basic issues with respect to timestamps and precedence tests for abstract events. Two criteria for timestamps and precedence tests are introduced: efficiency and hierarchical applicability. Furthermore, it is argued that, in general, one timestamp is not sufficient to determine precedence among abstract events.

The basic work on timestamping abstract events with vector timestamps is the paper of Haban and Weigel [19]. However, as mentioned, this work lacks a good analysis of causality among abstract events. The causality relation among abstract events is defined implicitly by their timestamps. As shown by Schwarz and Mattern in [30], this leads in some cases to counterintuitive and undesirable precedences among abstract events. They believe that the reason for these counterintuitive precedences is the fact that abstract events are assigned only a single timestamp, which denies their non-atomic nature. We agree with this conclusion and, below, we give an example showing that indeed at least two timestamps are needed to faithfully characterize precedence among arbitrary abstract events. Furthermore, Sections 6 and 7 show that each of the two precedence relations defined in the previous section can be characterized by two timestamps. Since the two characterizations share one timestamp, three timestamps are sufficient to characterize the combination of weak and strong precedence among arbitrary abstract events. Hence, Sections 6 and 7 present a solution—or at least a partial one—to one of the open problems stated in [30], namely that of assigning meaningful timestamps to arbitrary abstract events. The main reason that we did not encounter the problems stated by Schwarz and Mattern is that we separated the issues of specification and detection of abstract events, on the one hand, and assigning timestamps to abstract events, on the other hand. In our work, timestamps of abstract events are not derived from their specifications, as in the work of Haban and Weigel, but solely from the timestamps of their constituents. For a more detailed discussion of specifying, detecting, and timestamping abstract events, as well as an overview of some related work, see [30].

Before explaining our criteria for timestamps and precedence tests in more detail and substantiating our claim that a single timestamp is not sufficient to characterize causality among arbitrary abstract events, we make one assumption about precedence tests and a few assumptions about timestamps for abstract events.

A very obvious, but nonetheless important assumption about any test for strong or weak precedence among abstract events is that it should be a *correct and complete* characterization of (strong or weak) precedence. That is, a causal relation between two abstract events is derivable from a precedence test *if and only if* it is derivable from the corresponding definition (Definition 4.2 or 4.4). In particular, we do not consider any tests which are not complete. That is, we do not consider tests that do not yield all causal relations among abstract events. This assumption does not differ from the assumption made at the beginning of Section 3.2 for precedence tests for primitive events.

For timestamps, we make the following four assumptions. First, for the sake of simplicity, any timestamp should contain at most one integer entry per process.[1] Second, when calculating a timestamp for an abstract event, no information about any primitive event is used other than the process in which the event occurs, whether it is part of the abstract event, and its timestamp(s). Third, the only information about a process that may be used is whether (part of) the abstract event occurs in it. Finally, all abstract events are assigned timestamps that are calculated by means of the same timestamping algorithm(s). The last three assumptions ensure that no specific information about the structure of a distributed computation, its processes, or its primitive events is used. Thus, any timestamping scheme discussed in this paper is as general as possible and applicable to a wide variety of applications.

Timestamps and precedence tests not satisfying the above assumptions are not discussed in this paper. However, timestamps and tests that do satisfy the assumptions can be better or worse than others. Therefore, we introduce the following two criteria to evaluate timestamps and precedence tests. The first criterion is *efficiency*. Timestamps and precedence tests must be reasonably efficient in storage and computation time. That is, storage and computation time must be similar to storage and computation time needed for determining precedence between primitive events. Given the first assumption for timestamps above, the amount of storage needed for a precedence test is determined by the number of timestamps needed. Computation time is determined by the algorithms used to calculate timestamps and the number of comparisons needed to determine a causal relation between any two abstract events given their timestamps.

The second criterion is *hierarchical applicability*. In a hierarchy of true abstract descriptions of program behavior, a characterization of causality should not depend on any other levels than the level immediately below. This means that it must be possible to calculate a timestamp for an

[1] Since the timestamps in any timestamp scheme are countable, it is always possible to encode timestamps by single integers. However, we assume that the timestamps are derived from the timestamps of primitive events in a reasonable way, which excludes complex encodings into the integers. Moreover, such an encoding would complicate the comparison of timestamps for the purpose of determining causality, making any precedence test virtually impossible to use in practice.

abstract event from the timestamps of its constituents in the previous level. It also means that precedence tests must be defined in terms of the timestamps in the level being described. If tests depend on lower levels, determining precedence between two abstract events becomes computationally expensive, because the abstraction hierarchy must be traversed to a level that contains the desired information. Although for reasons of mathematical simplicity we have defined an abstract event as a set of primitive events and not as a set of lower-level abstract events, for most of the definitions and results in the remaining sections, it is clear whether they satisfy the second criterion. If not, some additional explanation is given.

The example in Figure 6 shows that under the above assumptions, for both weak and strong precedence, at least two timestamps are needed to properly characterize precedence relations among arbitrary abstract events. Note that this implies a lower bound on the amount of storage needed for a characterization of precedence, which is twice as high as the amount of storage needed to characterize precedence among primitive events.

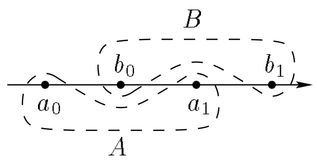

Figure 6: Why at least two timestamps are necessary.

It is obvious that in the simple sequential computation of Figure 6, consisting of only a single process, both $A \rightarrow B$ and $B \rightarrow A$. The first assumption for timestamps above implies that any timestamp for abstract events is simply an integer. In order to reflect the two weak causal precedences between A and B with only a single integer timestamp for each abstract event, the timestamps for A and B should be equal. However, under the above assumptions, it is not difficult to see that any reasonable timestamp for A is always smaller than *the same* timestamp for B.[2] Hence, one needs at least two timestamps to characterize weak precedence among abstract events.

For strong precedence, the reasoning is similar. Obviously, $A \not\prec B$ and $B \not\prec A$. That is, A and B are unrelated by the strong precedence relation. To reflect this with only a single timestamp, the timestamps for A and B must be unordered, which is impossible under the above assumptions.

One might argue that the example of Figure 6 is a degenerate case, but it is not hard to construct more complex examples for distributed computations with abstract events having constituents in more than one process.

5.2 Basic Definitions and Results

This subsection introduces some new definitions and adapts some previous definitions to abstract events. It also presents some basic results that are useful in the next sections.

Definition 5.1. (Location set) The location set of a primitive or abstract event is defined by a function $\boldsymbol{l}. : E \cup 2^E \longrightarrow 2^{\mathcal{P}}$ as the set of processes in which the event occurs. For any $e \in E$,

[2] It is possible to assign equal timestamps to A and B that satisfy all four assumptions: For example, any abstract event gets timestamp 481. However, this timestamp does not characterize weak precedence correctly for abstract events other than A and B. It is also possible to assign timestamps to A and B satisfying all the assumptions such that the timestamp of A is larger than the timestamp of B. For example, if for any abstract event C in the computation of the example, timestamp T is defined as $481 - (\mathrm{MAX}\, c : c \in C : T.c)$, then $T.A > T.B$. Apart from the fact that this timestamp does not solve the problem, we do not think it is reasonable. Note that Section 6 shows that $(\mathrm{MAX}\, c : c \in C : T.c)$ *is* a reasonable timestamp to determine precedence in the example computation.

$\boldsymbol{l}e = \{i \in \mathcal{P} \mid e \in E_i\}$. For any $A \subseteq E$, $\boldsymbol{l}A = \{i \in \mathcal{P} \mid A \cap E_i \neq \emptyset\}$. Note that the location set of a primitive event is always a singleton.

As mentioned before, a property of primitive events that no longer holds for abstract events is atomicity. This is expressed by the following two functions.

Definition 5.2. (Beginning and end of an abstract event) The beginning of an abstract event A is defined by a function $\lfloor . \rfloor : 2^E \longrightarrow 2^E$ as $\lfloor A \rfloor = \{a_0 \in A \mid \neg(\exists\, a_1 : \, a_1 \in A : \, a_1 \prec a_0)\}$. The end of an abstract event A is defined by a function $\lceil . \rceil : 2^E \longrightarrow 2^E$ as $\lceil A \rceil = \{a_0 \in A \mid \neg(\exists\, a_1 : \, a_1 \in A : a_0 \prec a_1)\}$.

Note that to determine precedence among abstract events, it is sufficient to consider the beginning and end of the events instead of the events in their entirety.

The following two definitions lift the notions of causal past and strict causal past to abstract events. The question is when a primitive event is an element of the (strict) causal past of an abstract event. The most general answer to this question leads to the definition of the causal past: A primitive event is in the causal past of an abstract event if and only if it precedes *any* of the primitive events in the abstract event. That is, the causal past of an abstract event is defined as the union of the causal pasts of all its primitive events. The second, more restricted answer to the above question yields the definition of the strict causal past: A primitive event is an element of the strict causal past of an abstract event if and only if it precedes *all* primitive events in the abstract event. The strict causal past of an abstract event is defined as the intersection of all *strict* causal pasts of its primitive events. Using the strict causal past of primitive events instead of their causal past yields a more natural result. If the latter had been used, the causal past of an abstract event might overlap with the abstract event itself.

Definition 5.3. (Causal past of an abstract event) The causal past of an abstract event is defined by a function $\mathbf{p}. : 2^E \longrightarrow 2^E$ as follows. For any $A \subseteq E$, $\mathbf{p}A = (\cup\, a : \, a \in A : \, \mathbf{p}a)$.

Definition 5.4. (Strict causal past of an abstract event) The strict causal past of an abstract event is defined by a function $\mathbf{p}^{-}. : 2^E \longrightarrow 2^E$ as follows. For any $A \subseteq E$, $\mathbf{p}^{-}A = (\cap\, a : \, a \in A : \, \mathbf{p}^{-}a)$. Note that $\mathbf{p}^{-}A$ is not necessarily equal to $\mathbf{p}A \setminus A$.

Definition 5.5. (Causal past of an abstract event in a process) For any $i \in \mathcal{P}$, the function $\mathbf{p}_i. : 2^E \longrightarrow 2^{E_i}$ defines the causal past in process i of an abstract event as follows. For any $A \subseteq E$, $\mathbf{p}_i A = \mathbf{p}A \cap E_i = (\cup\, a : \, a \in A : \, \mathbf{p}_i a)$.

Definition 5.6. (Strict causal past of an abstract event in a process) For any $i \in \mathcal{P}$, the function $\mathbf{p}_i^{-}. : 2^E \longrightarrow 2^{E_i}$ defines the strict causal past in process i of an abstract event as follows. For any $A \subseteq E$, $\mathbf{p}_i^{-}A = \mathbf{p}^{-}A \cap E_i = (\cap\, a : \, a \in A : \, \mathbf{p}_i^{-}a)$.

The following two corollaries are a direct result of the previous definitions. They show that the definitions of the beginning and end of abstract events are intuitively correct. They also show that the causal past and strict causal past are useful notions in reasoning about causality among abstract events, just as they proved to be useful in characterizing precedence among primitive events. Corollary 5.7 states that the past of the end of an abstract event corresponds to the past of the completed event. Corollary 5.8 shows that each event that precedes all events in the beginning of an abstract event precedes all the events in the abstract event.

Corollary 5.7. *For an abstract event A, $\mathbf{p}\lceil A\rceil = \mathbf{p}A$.*

Corollary 5.8. *For an abstract event A, $\mathbf{p}^{\text{-}}\lfloor A\rfloor = \mathbf{p}^{\text{-}}A$.*

The next corollary gives an expression for the global time of the causal past of an event in some process in terms of the global time of its causal past. It is a direct consequence of Definition 5.5 (Causal past of an abstract event in a process) and Definition 3.19 (Global time of a cut).

Corollary 5.9. *For any process $i \in \mathcal{P}$ and abstract event A,*

$$\mathcal{T}.\mathbf{p}_i A.j = \begin{cases} \mathcal{T}.\mathbf{p}A.i, & \text{for } j = i \\ 0, & \text{otherwise} \end{cases}$$

We have a similar result for the global time of the strict causal past of an abstract event in some process.

Corollary 5.10. *For any process $i \in \mathcal{P}$ and abstract event A,*

$$\mathcal{T}.\mathbf{p}_i^{\text{-}} A.j = \begin{cases} \mathcal{T}.\mathbf{p}^{\text{-}}A.i, & \text{for } j = i \\ 0, & \text{otherwise} \end{cases}$$

The last two results of this section give expressions for the global time of the causal past and strict causal past of abstract events. The global time of the causal past of an abstract event is equal to the componentwise maximum of the timestamps of its constituents; the global time of its strict causal past is equal to the componentwise minimum.

Property 5.11. *For any abstract event A,*

i) $\mathcal{T}.\mathbf{p}A = (\mathrm{SUP}\, a : a \in A : T.a)$

ii) $\mathcal{T}.\mathbf{p}^{\text{-}}A = (\mathrm{INF}\, a : a \in A : T^{\text{-}}.a)$

Proof. We prove only Property *i)*. The proof of Property *ii)* is similar.

$$\begin{aligned}
& \mathcal{T}.\mathbf{p}A \\
= \quad & \{ \text{ Definition 5.3 (Causal past of an abstract event) } \} \\
& \mathcal{T}.(\cup\, a : a \in A : \mathbf{p}a) \\
= \quad & \{ \text{ Theorem 3.23 (Structure of consistent time vectors) } \} \\
& (\mathrm{SUP}\, a : a \in A : \mathcal{T}.\mathbf{p}a) \\
= \quad & \{ \text{ Corollary 3.20 } \} \\
& (\mathrm{SUP}\, a : a \in A : T.a)
\end{aligned}$$

□

6 Characterizations of Strong Precedence

This section presents two timestamps and two precedence tests for the strong precedence relation on abstract events. One test does not use information about the location set of abstract events; the other does and is, therefore, more efficient. Consider the following derivation. Note that in this derivation, the introduction of timestamp $T^{\text{-}}$ for primitive events is very convenient.

$A \prec B$

$\Leftrightarrow$ { Definition 4.2 (Strong precedence) }

$(\forall a : a \in A : (\forall b : b \in B : a \prec b))$

$\Leftrightarrow$ { Corollary 3.10 }

$(\forall a : a \in A : (\forall b : b \in B : \mathbf{p}a \subseteq \mathbf{p}^{\text{-}}b))$

$\Leftrightarrow$ { Definition 5.3 (Causal past) }

$(\forall b : b \in B : \mathbf{p}A \subseteq \mathbf{p}^{\text{-}}b)$

$\Leftrightarrow$ { Definition 5.4 (Strict causal past) }

$\mathbf{p}A \subseteq \mathbf{p}^{\text{-}}B$

$\Leftrightarrow$ { Theorem 3.23 (Structure of consistent time vectors) }

$\mathcal{T}.\mathbf{p}A \leq \mathcal{T}.\mathbf{p}^{\text{-}}B$

$\Leftrightarrow$ { Property 5.11 }

$(\mathrm{SUP}\, a : a \in A : T.a) \leq (\mathrm{INF}\, b : b \in B : T^{\text{-}}.b)$

Summarizing,

$$A \prec B \Leftrightarrow (\mathrm{SUP}\, a : a \in A : T.a) \leq (\mathrm{INF}\, b : b \in B : T^{\text{-}}.b).$$

This result shows that it is useful to extend the two timestamp functions T and $T^{\text{-}}$ on primitive events to abstract events as follows.

Definition 6.1. (Timestamps for abstract events) The functions $T, T^{\text{-}} : 2^E \longrightarrow \mathbb{N}^N$ define timestamps for abstract events as follows. For any $A \subseteq E$, $T.A = (\mathrm{SUP}\, a : a \in A : T.a)$ and $T^{\text{-}}.A = (\mathrm{INF}\, a : a \in A : T^{\text{-}}.a)$.

Timestamp T is an efficient encoding of the causal past and, hence, the end of an abstract event (see also Corollary 5.7). Timestamp $T^{\text{-}}$ is an encoding of the strict causal past of an abstract event. It represents the cut containing all events that precede the beginning of the abstract event and, hence, all the primitive events in the abstract event (Corollary 5.8). In case of a hierarchy of abstract descriptions of program behavior, the associativity of the quantifiers SUP and INF implies that the timestamps of an abstract event are equal to the supremum and infimum of the timestamps of its constituents in the previous level of abstraction, which means that they satisfy the requirement of hierarchical applicability. Another criterion for timestamps is that their construction must be efficient. Given an abstract event A which consists of k primitive or lower-level abstract events, calculating $T.A$ needs $(k-1)\cdot N$ (binary) max operations on integers; calculating $T^{\text{-}}.A$ takes $(k-1)\cdot N$ min operations. Hence, the construction of both timestamps is indeed efficient. The introduction of the timestamps and the derivation above yield the following precedence test on abstract events. Figure 7 visualizes the meaning of both timestamps for abstract events and the precedence test.

Theorem 6.2. (Precedence test) *For any abstract events A and B, $A \prec B \Leftrightarrow T.A \leq T^{\text{-}}.B$.*

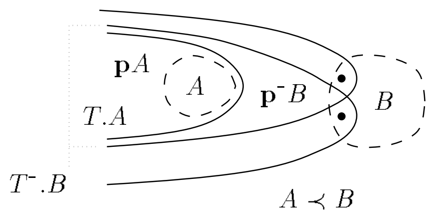

Figure 7: The meaning of the timestamps and the precedence test for abstract events.

As a simple example, consider the computation of Figure 6. The timestamps for events A and B are as follows: $T^{-}.A = 0$, $T.A = 3$, $T^{-}.B = 1$, and $T.B = 4$. It follows that $A \not\prec B$, because $T.A \not\leq T^{-}.B$. Since $T.B \not\leq T^{-}.A$, it also follows that $B \not\prec A$. Note that $T.A$ is smaller than $T.B$; the same is true for timestamp T^{-}, which conforms to the observation made in Section 5.1 that any reasonable timestamp for event A is always smaller than the corresponding timestamp for B.

Since the timestamps T and T^{-} for abstract events satisfy the requirement of hierarchical applicability, the precedence test of Theorem 6.2 also does. Concerning the efficiency of the test, the following can be said. First, it uses two different timestamps, which is the minimum number of timestamps. Hence, the required amount of storage is minimal. Second, we have already seen that the construction of T and T^{-} is very efficient. Finally, the maximum number of integer comparisons needed to determine whether some abstract event A precedes some abstract event B is equal to the number of processes N. To illustrate the meaning of this upper bound, simply checking whether all primitive events in A precede all primitive events in B leads to, using the precedence test of Theorem 3.14, an upper bound of $|A| \cdot |B| \cdot N$ integer comparisons. Keeping track of the beginning and end of A and B and only comparing primitive events in the end of A to events in the beginning of B leads to $|\lceil A \rceil| \cdot |\lfloor B \rfloor| \cdot N$ comparisons which implies an upper bound of $|\boldsymbol{l}A| \cdot |\boldsymbol{l}B| \cdot N$. However, calculating the beginning and end of an abstract events is computationally expensive, so the last test is not as efficient as it may seem. Summarizing, the test of Theorem 6.2 satisfies the two criteria for precedence tests. It is hierarchically applicable and its efficiency is very reasonable.

However, the test in Theorem 6.2 does not use the location set of an abstract event. It is possible to reduce the maximum number of integer comparisons if this information is available.

Assume A and B are abstract events. The derivation below yields a precedence test using location information. The second step might need some extra explanation. Let a and b be events in A and B respectively; let a occur in process j. If $a \prec b$, then Corollary 3.10 implies that $\mathbf{p}a \subseteq \mathbf{p}^{-}b$ and, hence, that for all $i \in \mathcal{P}$, $\mathbf{p}_i a \subseteq \mathbf{p}_i^{-} b$. On the other hand, if we know that for all $i \in \boldsymbol{l}A$, $\mathbf{p}_i a \subseteq \mathbf{p}_i^{-} b$, then obviously $\mathbf{p}_j a \subseteq \mathbf{p}_j^{-} b$, which by Corollary 3.12 implies that $a \prec b$. The other steps are all very similar to the derivation above.

$$
\begin{array}{ll}
 & A \prec B \\
\Leftrightarrow & \{ \text{ Definitions 4.2 (Strong precedence) } \} \\
 & (\forall\, a : a \in A : (\forall\, b : b \in B : a \prec b)) \\
\Leftrightarrow & \{ \text{ Corollaries 3.10 and 3.12 } \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : (\forall\, a : a \in A : (\forall\, b : b \in B : \mathbf{p}_i a \subseteq \mathbf{p}_i^{-} b))) \\
\Leftrightarrow & \{ \text{ Definition 5.5 (Causal past in a process) } \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : (\forall\, b : b \in B : \mathbf{p}_i A \subseteq \mathbf{p}_i^{-} b)) \\
\Leftrightarrow & \{ \text{ Definition 5.6 (Strict causal past in a process) } \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : \mathbf{p}_i A \subseteq \mathbf{p}_i^{-} B) \\
\Leftrightarrow & \{ \text{ Theorem 3.23 (Structure of time vectors) } \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : \mathcal{T}.\mathbf{p}_i A \leq \mathcal{T}.\mathbf{p}_i^{-} B) \\
\Leftrightarrow & \{ \text{ Corollaries 5.9 and 5.10 } \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : \mathcal{T}.\mathbf{p}A.i \leq \mathcal{T}.\mathbf{p}^{-}B.i) \\
\Leftrightarrow & \{ \text{ Property 5.11; Definition 6.1 (Timestamps } T \text{ and } T^{-}) \} \\
 & (\forall\, i : i \in \boldsymbol{l}A : T.A.i \leq T^{-}.B.i)
\end{array}
$$

This derivation leads to the following precedence test for strong precedence among abstract events, which is a generalization of the precedence test for primitive events given in Theorem 3.18.

Theorem 6.3. (Precedence test) *For any abstract events A and B, $A \prec B \Leftrightarrow (\forall\, i : i \in \boldsymbol{l}A : T.A.i \leq T^{-}.B.i)$.*

Obviously, this test also satisfies the two criteria for timestamps and precedence tests. As promised, it is even more efficient in terms of the maximum number of integer comparisons than the previous test of Theorem 6.2. The maximum number of integer comparisons needed to determine whether abstract event A precedes abstract event B is equal to $|\boldsymbol{l}A|$. Of course, the price to be paid for the improvement in the number of comparisons is that one has to keep track of the location information for primitive and abstract events.

To conclude, the formal framework presented in the previous sections leads in a fairly straightforward way to two characterizations of strong precedence that satisfy the two criteria of Section 5. One characterization uses location information, whereas the other does not. The question of which precedence test is most useful can be answered only in the context of a particular application.

7 Characterizations of Weak Precedence

7.1 Another Timestamp for Abstract Events

In this subsection, we give two characterizations of weak precedence. The first one is the result of a straightforward derivation and uses the timestamp T introduced in the previous section. It does not use location information for events. Unfortunately, this test does not satisfy the two criteria for precedence tests. The second half of this subsection introduces a new timestamp for abstract events and a precedence test in terms of this timestamp. This second test does satisfy the criteria. However, both the new timestamp and the precedence test depend on location information for events. In the framework developed so far, we have not been able to find a test for weak precedence that does not use location information. In the next subsection, we return to this point.

Let A and B be two abstract events. Consider the following derivation.

$$
\begin{array}{ll}
 & A \rightarrow B \\
\Leftrightarrow & \{ \text{ Definitions 4.4 (Weak precedence) and 5.2 (Beginning/end of abstract events) } \} \\
 & (\exists\, a : a \in \lfloor A \rfloor : (\exists\, b : b \in \lceil B \rceil : a \preceq b)) \\
\Leftrightarrow & \{ \text{ Corollary 3.9 } \} \\
 & (\exists\, a : a \in \lfloor A \rfloor : (\exists\, b : b \in \lceil B \rceil : \mathbf{p}a \subseteq \mathbf{p}b)) \\
\Leftrightarrow & \{ \text{ Definition 5.3 (Causal past); Corollary 5.7 } \} \\
 & (\exists\, a : a \in \lfloor A \rfloor : \mathbf{p}a \subseteq \mathbf{p}B))
\end{array}
$$

For any primitive event a,

$$
\begin{array}{ll}
 & \mathbf{p}a \subseteq \mathbf{p}B \\
\Leftrightarrow & \{ \text{ Theorem 3.23 (Structure of consistent time vectors) } \} \\
 & \mathcal{T}.\mathbf{p}a \leq \mathcal{T}.\mathbf{p}B \\
\Leftrightarrow & \{ \text{ Corollary 3.20; Property 5.11; Definition 6.1 (Timestamp } T \text{) } \} \\
 & T.a \leq T.B
\end{array}
$$

This derivation leads to the following precedence test for weak precedence, illustrated in Figure 8.

Theorem 7.1. (Precedence test) *For any abstract events A and B, $A \rightarrow B \Leftrightarrow (\exists\, a : a \in \lfloor A \rfloor : T.a \leq T.B)$.*

The test in Theorem 7.1 has two disadvantages. First, it is not very efficient. In the worst case, the timestamp of each primitive event in the beginning of abstract event A must be compared to the timestamp of the other abstract event, yielding $|\lfloor A \rfloor| \cdot N$ comparisons which implies an upper bound

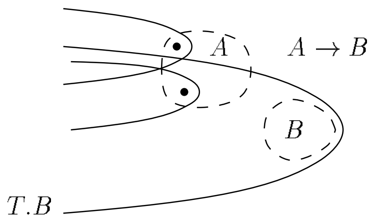

Figure 8: The meaning of the precedence test for weak precedence.

of $|\boldsymbol{l}A| \cdot N$ comparisons. Second, the test depends on the primitive level of the computation. Two or more abstract events cannot be merged into a higher-level abstract event without using information from the primitive level to compute the beginning of the newly formed abstract event. Therefore, the test does not satisfy either of the criteria for precedence tests. Note that the test in Theorem 7.1 uses only a single timestamp for abstract events. However, this does not contradict our conclusion of Section 5 that at least two timestamps are needed to determine precedence among abstract events. The use of timestamps from the primitive level of the computation is an implicit second timestamp.

The most important contribution of the above derivation and the resulting precedence test of Theorem 7.1 is that they illustrate the following interesting point. There is an asymmetry in the way the beginning and the end of the abstract events are used. The beginning of an abstract event is used explicitly. The end of an abstract event is encoded nicely in its timestamp T. If the asymmetry can be resolved, this might lead to a precedence test that does not depend on the primitive level of the computation.

So the question is whether we can find an encoding of the beginning of an abstract event. Note that timestamp T^- is too restrictive. It is possible to formulate a precedence test in terms of T^- (and T) that only yields a causal relation between two abstract events if there is such a relation, but that does not always give a relation if there is one. As mentioned in Section 5.1, we do not consider such precedence tests in this paper. We have not been able to answer the above question in the current framework *without using location information*. In the next subsection, the framework is extended with so-called *reversed vector time* which can be considered as the dual of vector time. With this extension, it is possible to find an encoding of the beginning of an abstract event, similar to the encoding of the end by timestamp T, that does not use location information. In the remainder of this section, we derive another timestamp for abstract events and a precedence test for weak precedence in terms of this timestamp. The new timestamp as well as the precedence test use information about the location set of abstract events; they (partially) resolve the asymmetry between the use of the beginning and the end of abstract events in the test of Theorem 7.1. It yields a precedence test which is independent of the primitive level of the computation and which is more efficient than the test of Theorem 7.1.

Let A and B be abstract events. It follows immediately from Definitions 5.1 (Location set) and 4.4 (Weak precedence) that

$$A \to B \Leftrightarrow (\exists\, i : i \in \boldsymbol{l}A : (\exists\, a : a \in A \cap E_i : (\exists\, b : b \in B : a \preceq b))).$$

Figure 8 may be helpful in understanding the following derivation, which is numbered for the purpose of future reference.

Derivation 7.2. Let i be a process in $\boldsymbol{l}A$.

$$(\exists\, a : a \in A \cap E_i : (\exists\, b : b \in B : a \preceq b))$$
$$\Leftrightarrow \quad \{\ \text{Corollary 3.11}\ \}$$

$(\exists\, a : a \in A \cap E_i : (\exists\, b : b \in B : \mathbf{p}_i a \subseteq \mathbf{p}_i b))$

$\Leftrightarrow$ { E_i is totally ordered by $\preceq$; Definition 3.7 (Causal past in a process) }

$(\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b)$

$\Leftrightarrow$ { Definition 5.5 (Causal past in a process) }

$(\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i B$

$\Leftrightarrow$ { Theorem 3.22 (Structure of time vectors) }

$\mathcal{T}.(\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \leq \mathcal{T}.\mathbf{p}_i B$

$\Leftrightarrow$ { Lemma 7.3 (see below); Corollary 5.9 }

$(\mathrm{MIN}\, a : a \in A \cap E_i : T.a.i) \leq \mathcal{T}.\mathbf{p}B.i$

$\Leftrightarrow$ { Property 5.11; Definition 6.1 (Timestamp T) }

$(\mathrm{MIN}\, a : a \in A \cap E_i : T.a.i) \leq T.B.i$

Lemma 7.3. *For any abstract event A, process $i \in \boldsymbol{l}A$, and process $j \in \mathcal{P}$,*

$$\mathcal{T}.(\cap\, a : a \in A \cap E_i : \mathbf{p}_i a).j = \begin{cases} (\mathrm{MIN}\, a : a \in A \cap E_i : T.a.i), & \textit{for } j = i \\ 0, & \textit{otherwise} \end{cases}$$

Proof. First, observe that for any $i \in \mathcal{P}$ and any event $e \in E$,

$$\mathcal{T}.\mathbf{p}_i e.j = \begin{cases} \mathcal{T}.\mathbf{p}e.i, & \text{for } j = i \\ 0, & \text{otherwise} \end{cases}$$

The following derivation proves the desired result.

$\mathcal{T}.(\cap\, a : a \in A \cap E_i : \mathbf{p}_i a).j$

$=$ { Theorem 3.22 (Structure of time vectors) }

$(\mathrm{INF}\, a : a \in A \cap E_i : \mathcal{T}.\mathbf{p}_i a).j$

$=$ { Definition (Componentwise minimum); observation above }

$$\begin{cases} (\mathrm{MIN}\, a : a \in A \cap E_i : \mathcal{T}.\mathbf{p}a.i), & \text{for } j = i \\ 0, & \text{otherwise} \end{cases}$$

$=$ { Corollary 3.20 }

$$\begin{cases} (\mathrm{MIN}\, a : a \in A \cap E_i : T.a.i), & \text{for } j = i \\ 0, & \text{otherwise} \end{cases}$$

□

Summarizing, the above derivations yield the following test:

$$A \to B \Leftrightarrow (\exists\, i : i \in \boldsymbol{l}A : (\mathrm{MIN}\, a : a \in A \cap E_i : T.a.i) \leq T.B.i\ ).$$

This result suggests the introduction of another timestamp for abstract events. The components corresponding to a process in the location set of an abstract event must be equal to the minimum calculated above; the other components can be chosen arbitrarily, since they are not used in the test. In order to define the new timestamp on abstract events, we also define it on primitive events.

Definition 7.4. (Timestamp T^w) The function $T^w : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows. For any $e \in E$,

$$T^w.e.i = \begin{cases} T.e.i, & \text{if } e \in E_i \\ \infty, & \text{otherwise} \end{cases}$$

For any $A \subseteq E$,

$$T^w.A = (\mathrm{INF}\, a : a \in A : T^w.a).$$

The value of all the components of timestamp T^w corresponding to processes outside the location set of a primitive event is set to infinity. The reason for this is mathematical convenience. In an actual implementation, one would choose some large integer value, for example, `MaxInt`.

It follows from the following derivation that timestamp T^w satisfies the above requirement concerning timestamp components corresponding to a process in the location set of some abstract event.

Derivation 7.5. For any abstract event A and process $i \in \boldsymbol{l}A$,

$$
\begin{array}{ll}
 & T^w.A.i \\
= & \quad \{ \text{ Definition 7.4 (Timestamp } T^w\text{); Definition INF(Componentwise minimum) } \} \\
 & (\text{MIN}\, a : a \in A : T^w.a.i) \\
= & \quad \{ \text{ Algebra } \} \\
 & (\text{MIN}\, a : a \in A \cap E_i : T^w.a.i) \min (\text{MIN}\, a : a \in A \setminus E_i : T^w.a.i) \\
= & \quad \{ \text{ Definition 7.4 (Timestamp } T^w\text{) } \} \\
 & (\text{MIN}\, a : a \in A \cap E_i : T.a.i)
\end{array}
$$

The definition of T^w immediately yields that for processes outside the location set of A, the corresponding component of $T^w.A$ equals infinity. The following precedence test follows from the calculations above and the introduction of the new timestamp.

Theorem 7.6. (Precedence test) *For abstract events A and B, $A \to B \Leftrightarrow (\exists\, i : i \in \boldsymbol{l}A : T^w.A.i \leq T.B.i)$.*

Let us consider again the example of Figure 6. For abstract events A and B, timestamp T^w is equal to 1 and 2 respectively. Since $T.A$ equals 3 and $T.B$ equals 4, it follows from the above precedence test that $A \to B$ and $B \to A$.

It remains to be verified whether this test satisfies the two criteria for timestamps and precedence tests. It follows from the associativity of the operator INF that, in a hierarchy of abstract descriptions of program behavior, for any abstract event, timestamp T^w can be calculated from the timestamps of its constituents in the level immediately below. Hence, the timestamp and the test satisfy the criterion of hierarchical applicability. Concerning the efficiency of the test, the following can be said. As for the two characterizations of strong precedence, it is necessary to maintain two timestamps for every abstract event. Note that it is not necessary to maintain a second set of timestamps for *primitive* events. For primitive events, timestamp T^w can be calculated immediately from timestamp T, provided, of course, that it is known in what process the events occur. However, in order to use the new timestamp in a precedence test for abstract events, this information is necessary anyway, so it is not a real restriction. The construction of the new timestamp T^w is as efficient as the construction of T and T^-. As for the test in Theorem 6.3, the maximum number of integer comparisons needed to determine whether some abstract event A precedes another event is $|\boldsymbol{l}A|$.

7.2 Reversed Vector Time

This section presents a weak-precedence test for abstract events that does not use location information for events. For this purpose, we extend the framework developed so far with the causal future of events and *reversed vector time*. These notions are the duals of the causal past and ordinary vector time, respectively. They lead to an encoding of the beginning of an abstract event in terms of a reversed vector timestamp which is similar to the encoding of the end of an abstract event in terms of its timestamp T. The notion of the causal future of an event was already mentioned in [28]. However, the idea to use it as the basis for a timestamp is new. A drawback of reversed vector time is that it is only suitable for post-mortem analysis of distributed computations. The whole set of primitive events is needed to calculate reversed timestamps. Thus, precedence tests in terms of reversed vector time are computationally more expensive than any of the tests presented so far. However, as explained in Section 2, in applications such as distributed debugging, the restriction to post-mortem analysis is not necessarily a limitation. More practical experience with event abstraction is needed to determine whether reversed vector time is a practically useful notion. For now, its main contribution is a better insight into the meaning of causality among abstract events. It also completes the results presented in this paper in the sense that it yields a precedence test for weak precedence which does not use location information and which satisfies the criterion of hierarchical applicability. A more extensive treatment of reversed vector time than presented in this subsection can be found in [5]. An application of reversed vector time to distributed breakpoints is described in [4].

The notions of causal future and reversed vector time are based on the *successor relation* $\succ$, which is defined as the dual of $\prec$. That is, for any $e_0, e_1 \in E$, $e_0 \succ e_1$ if and only if $e_1 \prec e_0$. The local successor relation $\succ_l$ is the dual of $\prec_l$. The relations $\succeq$ and $\succeq_l$ are the reflexive closures of $\succ$ and $\succ_l$ respectively. The following definition introduces the dual notion of cuts.

Definition 7.7. ((Consistent) successor cut) A set $C \subseteq E$ is called a successor cut, or $\succ$-cut, of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \succ_l e_0 \Rightarrow e_1 \in C$. A $\succ$-cut is left-closed under $\succ_l$. The set of all $\succ$-cuts is denoted by $C_{\succ_l}$. Set C is called a *consistent* $\succ$-cut of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \succ e_0 \Rightarrow e_1 \in C$. A consistent $\succ$-cut is left-closed under $\succ$. The set of all consistent $\succ$-cuts is denoted by $C_{\succ}$.

The next corollary states the obvious relation between cuts and $\succ$-cuts.

Corollary 7.8. *For any $C \subseteq E$, $C \in C_{\prec_l} \Leftrightarrow E \setminus C \in C_{\succ_l}$ and $C \in C_{\prec} \Leftrightarrow E \setminus C \in C_{\succ}$.*

The counterpart of the causal past of events is the so-called causal future.

Definition 7.9. (Causal future) Function $\mathbf{f}. : E \cup 2^E \longrightarrow 2^E$ defines the causal future of primitive and abstract events as follows. For any $e \in E$, $\mathbf{f}e = \{e_0 \in E \mid e_0 \succeq e\}$. For any $A \subseteq E$, $\mathbf{f}A = (\cup\, a : a \in A : \mathbf{f}a)$. Note that $\mathbf{f}e$ and $\mathbf{f}A$ are consistent $\succ$-cuts.

Definition 7.10. (Causal future in a process) For any process $i \in \mathcal{P}$, function $\mathbf{f}_i. : E \longrightarrow 2^E$ defines the causal future in process i of an event. For any $e \in E$, $\mathbf{f}_i e = \{e_0 \in E_i \mid e_0 \succeq e\}$.

Since we are only interested in finite (prefixes of) computations, it is appropriate to define the following.

Definition 7.11. (Reversed vector time of a successor cut) Function $\mathcal{T}^R : C_{\succ_l} \rightarrow \mathbb{N}^N$ defines the *reversed vector time* of a $\succ$-cut. For any $\succ$-cut C, component i, where $0 \leq i < N$, of the reversed time vector is defined as $\mathcal{T}^R.C.i = |C \cap E_i|$.

The following corollary is a direct result of this definition, the definition of the time of a cut (3.19), and Corollary 7.8. It states the relation between vector time and reversed vector time. The binary operator "$-$" on vectors denotes componentwise subtraction. Time vector $\boldsymbol{E}$ is a constant vector whose i^{th} component is equal to the number of events in process i, i.e., $|E_i|$.

Corollary 7.12. *For any successor cut $C \in C_{\succ_l}$, $\mathcal{T}.(E \setminus C) = \boldsymbol{E} - \mathcal{T}^R.C$. For any cut $C \in C_{\prec_l}$, $\mathcal{T}^R.(E \setminus C) = \boldsymbol{E} - \mathcal{T}.C$.*

Finally, we define reversed vector timestamps for primitive events.

Definition 7.13. (Timestamp function T^R) The function $T^R : E \longrightarrow \mathbb{N}^N$ defines a timestamp in reversed vector time for primitive events. For any $e \in E$ and $i \in \mathcal{P}$, $T^R.e.i = |\mathbf{f}_i e|$.

The function T^R encodes exactly the set of timestamps that is obtained by applying a timestamp algorithm for ordinary vector timestamps while traversing event information backwards. As mentioned, this is computationally expensive and it is restricted to post-mortem analysis of distributed computations. Also, it is necessary to maintain two sets of timestamps for primitive events, which requires substantial extra storage. (Recall that, for primitive events, timestamps T^- and T^w can be easily expressed in terms of timestamp T, which means that it is not necessary to store them separately.)

The following results are a direct consequence of the duality of the relations $\preceq$ and $\succeq$. Therefore, they are given without proof.

Corollary 7.14. *For any event $e \in E$, $\mathcal{T}^R.\mathbf{f}e = T^R.e$.*

Corollary 7.14 shows that the reversed timestamp of an event encodes its causal future. Corollary 7.15 is the dual of Corollary 5.7. It states that the beginning of an abstract event and the event itself share the same causal future, which is a hint that the causal future is useful for determining precedence relations among abstract events.

Corollary 7.15. *For an abstract event A, $\mathbf{f}\lfloor A \rfloor = \mathbf{f}A$.*

The following property gives a simple expression in terms of reversed vector time for the beginning of an abstract event.

Property 7.16. *For any abstract event A, $\mathcal{T}^R.\mathbf{f}A = (\mathrm{SUP}\, a : a \in A : T^R.a)$.*

The following derivation shows the meaning of precedence between abstract events in terms of causal past and causal future. Let A and B be abstract events. Informally, A weakly precedes B if and only if the causal future of A shares some events with the causal past of B. Figure 9 clarifies the derivation.

$$
\begin{array}{ll}
 & A \rightarrow B \\
\Leftrightarrow & \{ \text{ The relation } \preceq \text{ is reflexive and transitive; Definition 4.4 (Weak precedence) } \} \\
 & (\exists\, e : e \in E : (\exists\, a : a \in A : a \preceq e) \wedge (\exists\, b : b \in B : e \preceq b)) \\
\Leftrightarrow & \{ \text{ Definitions 7.9 (Causal future) and 3.5 (Causal past) } \} \\
 & (\exists\, e : e \in E : (\exists\, a : a \in A : e \in \mathbf{f}a) \wedge (\exists\, b : b \in B : e \in \mathbf{p}b)) \\
\Leftrightarrow & \{ \text{ Definitions 7.9 (Causal future) and 5.3 (Causal past) } \} \\
 & (\exists\, e : e \in E : e \in \mathbf{f}A \wedge e \in \mathbf{p}B) \\
\Leftrightarrow & \{ \text{ Definition of set intersection } \} \\
 & \mathbf{f}A \cap \mathbf{p}B \neq \emptyset \\
\Leftrightarrow & \{ \text{ Set calculus; } \mathbf{f}A \subseteq E \text{ and } \mathbf{p}B \subseteq E \ \} \\
 & E \setminus \mathbf{f}A \not\supseteq \mathbf{p}B \\
\Leftrightarrow & \{ \text{ Corollary 7.8; Theorem 3.23 (Structure of consistent time vectors) } \} \\
 & \mathcal{T}.(E \setminus \mathbf{f}A) \not\geq \mathcal{T}.\mathbf{p}B \\
\Leftrightarrow & \{ \text{ Corollary 7.12 } \} \\
 & \boldsymbol{E} - \mathcal{T}^R.\mathbf{f}A \not\geq \mathcal{T}.\mathbf{p}B
\end{array}
$$

$\Leftrightarrow$ { Property 7.16; Property 5.11; Definition 6.1 (Timestamp T) }
$\boldsymbol{E} - (\mathrm{SUP}\, a : a \in A : T^R.a) \not\geq T.B$

Property 7.16 gives an expression for the *reversed* time of the beginning of an abstract events. Expression $\boldsymbol{E} - (\mathrm{SUP}\, a : a \in A : T^R.a)$ is an expression for the beginning of an abstract event in terms of ordinary vector time. It is not meaningful to compare times in the two different representations of time. The above result leads to the introduction of the following timestamp for abstract events.

Definition 7.17. (Reversed timestamp of an abstract event) Function $T^R : 2^E \longrightarrow \mathbb{N}^N$ defines the reversed timestamp of an abstract event. For any abstract event A, $T^R.A = (\mathrm{SUP}\, a : a \in A : T^R.a)$.

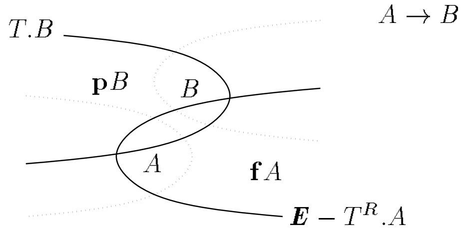

Figure 9: Weak precedence and reversed vector time.

The introduction of the reversed vector timestamp yields the following precedence test, which is illustrated in Figure 9.

Theorem 7.18. *For any abstract events A and B, $A \rightarrow B \Leftrightarrow \boldsymbol{E} - T^R.A \not\geq T.B$.*

Consider the computation of Figure 6 one last time. Reversed timestamp $T^R.A$ is equal to 4 and, hence, $\boldsymbol{E} - T^R.A$ equals 0; reversed timestamp $T^R.B$ equals 3, which implies that $\boldsymbol{E} - T^R.B$ is equal to 1. Recall that $T.A$ and $T.B$ are equal to 3 and 4 respectively. As before, this yields that $A \rightarrow B$ and $B \rightarrow A$. Note that the fact that $T^R.A$ is larger than $T^R.B$ does not contradict our argument of Section 5.1 that any reasonable timestamp for A is always smaller than the same timestamp for B. Timestamps $T^R.A$ and $T^R.B$ are times in *reversed* vector time. The corresponding timestamps in ordinary vector time, $\boldsymbol{E} - T^R.A$ and $\boldsymbol{E} - T^R.B$, confirm the argument of Section 5.1.

The precedence test of Theorem 7.18 is independent of the computation at the level of primitive events and does not use location information for primitive events. In this sense, it fills a gap which was left open in the previous subsection. As before, it is not difficult to see that it satisfies the criterion of hierarchical applicability. In terms of integer comparisons, it is reasonably efficient. Checking whether an abstract event precedes some other abstract event requires at most N comparisons. As already explained, it is more expensive in storage and computation time than any of the other tests given so far. It is also restricted to post-mortem analysis. Its main contribution is that it has a very clear intuitive meaning: Event A weakly precedes event B if and only if A begins before B ends.

8 Timestamping Convex Abstract Events

8.1 Convex Abstract Events

Up to this point, no restrictions have been imposed on the structure of abstract events. However, applications do not necessarily use arbitrary subsets of events. In this section, the subclass of *convex* abstract events is studied. The main result is that for mutually disjoint, convex abstract events, a single timestamp is sufficient to characterize *weak* precedence. This result has been used in the implementation of the tool that was used to visualize the sample computation in Section 2.

Definition 8.1. (Convex abstract events) An abstract event A is called *convex* if and only if $(\forall\, a_0, a_1, e : a_0, a_1 \in A \wedge e \in E : a_0 \preceq e \wedge e \preceq a_1 \Rightarrow e \in A)$.

Convexity is a meaningful requirement for abstract events for the following reason. For a convex abstract event A, there is no (primitive or abstract) event α in the previous level of abstraction that is not a constituent of A but that depends on the completion of part of A such that, in turn, the completion of A depends on α. In other words, there is no outside interference; a convex abstract event describes a *complete unit* of work.

Convexity is useful as well. First, convex abstract events are easier to recognize automatically than arbitrary abstract events, because it is not necessary to filter out interfering events. Second, they are more general and therefore more widely applicable than, for example, contractions [8, 13]. A contraction is an abstract event whose internal structure is restricted in such a way that it may be considered to occur atomically. Third, since there are no interfering events, convex abstract events are considerably easier to display than arbitrary abstract events, which is very important in an application such as distributed debugging. Finally, this section shows that determining *weak* precedence relations among *mutually disjoint*, convex abstract events requires less timestamping effort than determining weak precedence relations among arbitrary abstract events. For disjoint, convex abstract events, a single timestamp proves to be sufficient to characterize the weak precedence relation. Although weaker conditions than convexity and disjointness might exist, convexity and disjointness are sufficient. For most applications, mutual disjointness of abstract events is not a real restriction. On the contrary, it is often a useful requirement. For example, in distributed debugging, it is not meaningful if an abstract view of a computation has overlapping abstract events.

There does not seem to be any other meaningful class of abstract events that is as general as the class of convex abstract events and that combines so many useful properties. Hence, in this section, we investigate timestamps and precedence tests for convex abstract events.

The example of Figure 6 in Section 5.1 already shows that for arbitrary non-convex abstract events, a single timestamp is not sufficient to characterize weak precedence. Note that abstract events A and B are indeed not convex. To show that convexity alone is not a sufficient condition, consider the same computation, but with abstract events $A' = \{a_0, b_0, a_1\}$ and $B' = \{b_0, a_1, b_1\}$. It is clear that A' and B' are convex. However, a single timestamp is still not sufficient. Since both $A' \to B'$ and $B' \to A'$, the timestamps for A' and B' should be equal. Given the assumptions of Section 5.1, this is not possible, because, as for A and B, any reasonable timestamp for A' is smaller than the same timestamp for B'.

Unfortunately, for the *strong* precedence relation, a single timestamp does not seem to be sufficient, as can be explained by means of the example in Figure 10.

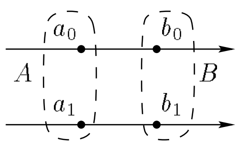

Figure 10: Why a single timestamp appears to be insufficient for characterizing strong precedence among disjoint, convex abstract events.

Obviously, abstract events A and B are convex. Furthermore, because events a_0 and b_1 are concurrent, A and B are unrelated by the strong precedence relation. Hence, if abstract events have only a single timestamp to characterize strong precedence, the timestamps of A and B must also be unrelated. Assuming that the only operators available to calculate timestamps are minimum and maximum operators such as "min," "max," "inf," and "sup," in combination with the assumptions about timestamps mentioned in Section 5.1, any reasonable timestamp assigned to A is always smaller than or equal to the timestamp of B. Hence, the timestamps are not unrelated and, thus, we may conclude that a single timestamp cannot be sufficient. Of course, there may be some ingenious timestamping scheme which is sufficient to characterize strong precedence among convex abstract events by only a single timestamp, but such a timestamp, in all likelihood, would have to be fundamentally different from the ones discussed here. This example raises the question of what conditions on abstract events are needed to characterize strong precedence by only a single timestamp. Although this is an interesting question, we do not try to answer it in this paper, but leave it for future work.

8.2 Characterizing Weak Precedence among Convex Abstract Events

It is a nice exercise for the reader to give examples showing that any one of the timestamps for abstract events given so far, T, T^{-}, T^{w}, and T^{R} cannot be used to characterize weak precedence among mutually disjoint, convex abstract events. Instead, we introduce a new timestamp T^{c} which is a combination of T and T^{w}.

Definition 8.2. (A single timestamp for convex abstract events) The function $T^{c} : E \cup 2^{E} \longrightarrow \mathbb{N}^{N}$ defines a timestamp for primitive and abstract events as follows. For any $\alpha \in E \cup 2^{E}$, $T^{c}.\alpha = T.\alpha \inf T^{w}.\alpha$.

For processes inside the location set of a convex abstract event, timestamp T^{c} more or less conforms to the beginning of the event. For processes outside the location set, it represents the end. This is formalized in the following two corollaries, which follow immediately from the definitions of T, T^{w}, and T^{c}. Note that the corollaries are true for arbitrary primitive or abstract events.

Corollary 8.3. *For any primitive or abstract event $\alpha \in E \cup 2^{E}$ and any process $i \in \boldsymbol{l}\alpha$, $T^{c}.\alpha.i = T^{w}.\alpha.i$.*

Corollary 8.4. *For any primitive or abstract event $\alpha \in E \cup 2^{E}$ and process $i \notin \boldsymbol{l}\alpha$, $T^{c}.\alpha.i = T.\alpha.i$.*

It follows from these corollaries and the definition of T^{w} (Definition 7.4) that for primitive events, timestamp T^{c} is equal to timestamp T.

Timestamp T^{c} can be used to formulate the following precedence test for mutually disjoint, convex abstract events.

Theorem 8.5. (Precedence test for disjoint, convex abstract events) *For any disjoint, convex abstract events A and B, $A \rightarrow B \Leftrightarrow (\exists\, i : i \in \boldsymbol{l}A : T^{c}.A.i \leq T^{c}.B.i)$.*

Proof. The implication from right to left follows immediately from Theorem 7.6, Corollaries 8.3 and 8.4, and the observation that for any abstract event C, by definition, $T^{c}.C$ is always at most $T.C$. The other implication is more involved.

$A \rightarrow B$

$\Rightarrow$ { Derivation 7.2 }
$(\exists\, i : i \in \boldsymbol{l}A : (\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b))$
$\Rightarrow$ { Set calculus }
$(\exists\, i : i \in \boldsymbol{l}A \setminus \boldsymbol{l}B : (\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b)) \vee$
$(\exists\, i : i \in \boldsymbol{l}A \cap \boldsymbol{l}B : (\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b))$

Let i be a process in $\boldsymbol{l}A \setminus \boldsymbol{l}B$.

$(\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b)$
$\Rightarrow$ { Derivation 7.2; Definition 7.4 (Timestamp T^w) }
$T^w.A.i \leq T.B.i$
$\Rightarrow$ { Definition 8.2 (Timestamp T^c); Corollaries 8.3 and 8.4 }
$T^c.A.i \leq T^c.B.i$

Let i be a process in $\boldsymbol{l}A \cap \boldsymbol{l}B$.

$(\exists\, b : b \in B : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b)$
$\Rightarrow$ { E_i is totally ordered; $B \cap E_i \neq \emptyset$; B is convex }
$(\exists\, b : b \in B \cap E_i : (\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq \mathbf{p}_i b)$
$\Rightarrow$ { B is convex; A and B disjoint }
$(\cap\, a : a \in A \cap E_i : \mathbf{p}_i a) \subseteq (\cap\, b : b \in B \cap E_i : \mathbf{p}_i b)$
$\Rightarrow$ { Similar to Derivation 7.2; Definition 7.4 (Timestamp T^w) }
$T^w.A.i \leq T^w.B.i$
$\Rightarrow$ { Definition 8.2 (Timestamp T^c); Corollary 8.3; $i \in \boldsymbol{l}A \cap \boldsymbol{l}B$ }
$T^c.A.i \leq T^c.B.i$

Hence, $A \rightarrow B \Rightarrow (\exists\, i : i \in \boldsymbol{l}A : T^c.A.i \leq T^c.B.i)$, which completes the proof. Note that both convexity and disjointness are indeed used in the proof, although only convexity of B is needed. □

Timestamp T^c cannot be used to formulate a precedence test for disjoint, convex abstract events that does *not* use location information. Consider again the computation in Figure 5(a). Since $A \rightarrow B$, we would like to have that $T^c.A \leq T^c.B$. That is, the timestamps should reflect the weak precedence relation between A and B. However, it is not difficult to see that for abstract event A, $T.A = T^w.A = (1,2)$. Hence, also $T^c.A = (1,2)$. For abstract event B, $T.B = T^w.B = T^c.B = (2,1)$, which means that $T^c.A \not\leq T^c.B$. Note that we do have that $T^c.A.1 \leq T^c.B.1$, which conforms to the precedence test of Theorem 8.5 that does use location information. It is an interesting question whether there exists a precedence test for weak precedence formulated in terms of a single timestamp that does not use location information. The above example suggests that the answer is negative. Since both $A \rightarrow B$ and $B \rightarrow A$, any such timestamping scheme should assign equal timestamps to A and B. It is very unlikely that such a scheme exists which is also correct for any other computation.

It remains to be shown that the test of Theorem 8.5 satisfies the two criteria for precedence tests. Since timestamp T^c is defined in terms of T and T^w, this is not immediately clear. It would be inefficient if it would be necessary to maintain both T and T^w for all abstract events. Even worse, it would invalidate our claim that a single timestamp is sufficient to determine weak precedence among abstract events. Therefore, we give an inductive definition of timestamp T^c that is independent of T and T^w. For this purpose, assume we have an abstraction hierarchy where α ***co*** A denotes that primitive or abstract event α is a constituent of abstract event A in the abstraction level immediately below the level containing A.

Definition 8.6. (Inductive definition of T^c) The function $T^{ci} : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows. For any $e \in E$,

$T^{ci}.e = T.e.$

For any $A \subseteq E$,

$$T^{ci}.A.i = \begin{cases} (\mathrm{MIN}\,\alpha : \alpha \,\boldsymbol{co}\, A \wedge i \in \boldsymbol{l}\alpha : T^{ci}.\alpha.i), & \text{for } i \in \boldsymbol{l}A \\ (\mathrm{MAX}\,\alpha : \alpha \,\boldsymbol{co}\, A : T^{ci}.\alpha.i), & \text{otherwise} \end{cases}$$

Property 8.7. $T^c = T^{ci}$.

Proof. We already observed that for primitive events, T^c is equal to T. Hence, it follows from the definition of T^{ci} that for primitive events, T^c is equal to T^{ci}. It follows from Corollaries 8.3 and 8.4, Definitions 6.1 (Timestamps T) and 7.4 (Timestamp T^w), and Derivation 7.5 that for any abstract event A,

$$T^c.A.i = \begin{cases} (\mathrm{MIN}\,a : a \in A \cap E_i : T.a.i), & \text{for } i \in \boldsymbol{l}A \\ (\mathrm{MAX}\,a : a \in A : T.a.i), & \text{otherwise} \end{cases}$$

By means of induction, it is not difficult to show that $T^{ci}.A.i$ can be rewritten to this expression as well. Hence, also for abstract events T^c is equal to T^{ci}, which concludes the proof. (See [5] for the details of the induction proof.) □

Definition 8.6 and Property 8.7 show that timestamp T^c and, hence, the precedence test of Theorem 8.5 satisfy the criterion of hierarchical applicability. In addition, they show that indeed a single timestamp is sufficient to characterize weak precedence. For primitive events, it is sufficient to maintain timestamp T. For abstract events timestamp T^c is sufficient. Definition 8.6 gives an efficient algorithm to compute T^c, which uses the same number of min/max operations as needed for the construction of any of the other timestamps for abstract events given in this paper. The maximum number of integer comparisons to check whether a convex abstract event A weakly precedes another disjoint, convex abstract event B is equal to $|\boldsymbol{l}A|$.

9 Conclusions

In this paper, we have studied causality among abstract events and its characterization in terms of vector time. As for primitive events, causality among abstract events can be expressed by means of precedence relations. Following Lamport [24], in Section 4, we introduced two precedence relations on abstract events, namely strong precedence and weak precedence. An abstract event strongly precedes another abstract event if and only if all its constituents precede all constituents of the other event. That is, the abstract event as a whole precedes the other abstract event as a whole. The strong precedence relation on abstract events has the nice property that it is a partial order. However, it is not well suited to express concurrency among abstract events or to express the fact that only part of some abstract event precedes part of some other abstract event. The weak precedence relation complements the strong precedence relation in the sense that it expresses that part of an abstract event causally affects part of another event. It also allows for a natural definition of concurrency. Unfortunately, the weak precedence relation is not a partial order. The combination of strong and weak precedence seems to be a proper characterization of causality among abstract events (see also [24]).

In Section 5, we explained the main goal of this paper, namely finding characterizations of strong and weak precedence in terms of vector time. The characterization must make it possible to determine causal relationships between two abstract events efficiently in a hierarchy of abstract descriptions of program behavior. We have argued that, for both weak and strong precedence, a single timestamp cannot be sufficient (see also [30], where this conjecture is made as well).

In Sections 6 and 7, we have studied characterizations of strong and weak precedence among abstract events. Both precedence tests using location information of events and tests not using such information have been given. The following table gives an overview of the results.

	Strong precedence	**Weak precedence**
Location-independent precedence test	Thm 6.2 (T,T^-)	Thm 7.18 (T,T^R)
Location-dependent precedence test	Thm 6.3 (T,T^-)	Thm 7.6 (T,T^w)

It proved to be relatively straightforward to arrive at the results for strong precedence. Strong precedence can be characterized efficiently by means of two timestamps, T and T^-, both with and without using location information. Timestamp T is an encoding of the end of an abstract event. Timestamp T^- of an abstract event represents the set of primitive events preceding all constituents of the abstract event.

It proved to be more difficult to achieve equivalent results for weak precedence. Using location information, it is possible to define a timestamp T^w, whose components correspond more or less to the beginning of an abstract event. In combination with T, timestamp T^w gives an efficient characterization of weak precedence. In order to find a representation for weak precedence not using location information, we had to introduce yet another timestamp, namely T^R. This so-called reversed timestamp has a serious drawback. Since it is the dual of timestamp T, it is calculated by applying a timestamp algorithm while traversing event information backwards. This means that it is restricted to post-mortem analysis of distributed computations. While this may not be a problem for some applications, it may be for others. Despite this drawback, timestamp T^R is useful in obtaining a better understanding of causality among abstract events. It is a very intuitive encoding of the beginning of an abstract event, similar to the way timestamp T encodes the end. It is an interesting open problem whether it is possible to characterize weak precedence without using location information in a way that is suitable for on-the-fly analysis of distributed computations.

The results of this paper show that two timestamps are sufficient to characterize the strong or weak precedence relation on abstract events in isolation. Note, however, that *three* timestamps are needed to characterize the combination of strong and weak precedence. Either T, T^-, and T^w are needed when location information is available, or T, T^-, and T^R when location information is not available. The question of which timestamps and which precedence tests should be used can only be answered in a specific context. Note that some uses might even require (slightly) different formalizations of precedence. Although the results of this paper are then no longer directly applicable, the framework is general enough that it may be adapted to other precedence relations.

Finally, in Section 8 we studied convex abstract events. The class of convex abstract events is a meaningful class of events that is widely applicable and restricted enough to simplify timestamping. A single timestamp is sufficient to characterize weak precedence among mutually disjoint, convex abstract events, provided at least that location information is available. An example showing an implementation of this result in the context of distributed debugging is discussed in Section 2. Unfortunately, for strong precedence there is no such result. A simple example shows that it is unlikely that a single timestamp can be found characterizing strong precedence among mutually disjoint, convex abstract events. This raises the interesting question of what restrictions on abstract events are necessary to characterize strong precedence among abstract events by only a single timestamp. A related question is what restrictions are sufficient to allow a characterization of strong *or* weak precedence by means of only a single timestamp while *not* using location information.

Summarizing, the results presented in this paper are a step towards the solution of one of the open problems stated in [30], namely that of assigning meaningful timestamps to arbitrary abstract events. Some questions have been answered; some others have been raised.

Acknowledgments. We are grateful to the anonymous referees of an earlier version of this paper, whose comments improved our insight in the matter of causality among abstract events. This led to substantial changes and, more important, to substantial simplifications of the theory presented in this paper.

References

1. M. Ahuja, A.D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *IEEE Proceedings of the 10th. International Conference on Distributed Computing Systems*, pages 12–19, Paris, France, May/June 1990. IEEE Computer Society Press, Los Alamitos, CA.

2. M. Ahuja and S. Mishra. Units of computation in fault-tolerant distributed systems. In *IEEE Proceedings of the 14th. International Conference on Distributed Computing Systems*, pages 626–633, Poznan, Poland, June 1994. IEEE Computer Society Press, Los Alamitos, CA.

3. Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S.J. Mullender, editor, *Distributed Systems* (2nd. edition), chapter 4, pages 55–96. Addison–Wesley, 1993.

4. T. Basten. Breakpoints and time in distributed computations. In G. Tel and P.M.B. Vitányi, editors, *Distributed Algorithms, 8th. International Workshop, WDAG '94, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 340–355, Terschelling, The Netherlands, September/October 1994. Springer–Verlag, Berlin, Germany, 1994.

5. T. Basten, T. Kunz, J.P. Black, M.H. Coffin, and D.J. Taylor. Time and the order of abstract events in distributed computations. Computing Science Note 94/06, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, February 1994.

6. P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

7. P.C. Bates and J.C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *The Journal of Systems and Software*, 3(4):255–264, December 1983.

8. E. Best and B. Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93–124, 1981.

9. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

10. B. Charron-Bost. Combinatorics and geometry of consistent cuts: Application to concurrency theory. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms, 3rd. International Workshop, WDAG '89, Proceedings*, volume 392 of *Lecture Notes in Computer Science*, pages 45–56, Nice, France, September 1989. Springer–Verlag, Berlin, Germany.

11. B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

12. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, February 1996.

13. W.-H. Cheung. *Process and event abstraction for debugging distributed programs.* PhD thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989. Also appeared as CCNG Technical Report T-189, 1989.

14. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. The proceedings appeared also as *ACM SIGPLAN Notices*, 26(12), December 1991.

15. C.J. Fidge. Partial orders for parallel debugging. *ACM Sigplan Notices*, 24(1):183–194, January 1989.

16. C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

17. E. Fromentin and M. Raynal. Local states in distributed computations: A few relations and formulas. *ACM Operating Systems Review*, 28(2):65–72, 1994.

18. E. Fromentin and M. Raynal. Characterizing and detecting the set of global states seen by all observers of a distributed computation. In *IEEE Proceedings of the 15th. International Conference on Distributed Computing Systems*, pages 431–438. IEEE Computer Society Press, Los Alamitos, CA, 1995.

19. D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st. Annual Hawaii International Conference on System Sciences,* Volume II, pages 166–175, Kailua-Kona, Hawaii, January 1988.

20. J. Kundu and J.E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Frontiers '95. Proceedings of the 5th. Symposium on the Frontiers of Massively Parallel Computation*, pages 472–479, 1995.

21. T. Kunz. Visualizing abstract events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, Toronto, Ontario, Canada, November 1994. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

22. T. Kunz, J.P. Black, D.J. Taylor, and T. Basten. Target-system-independent visualizations of complex distributed-application executions. *The Computer Journal,* special issue on software engineering for distributed systems, 1997. To appear.

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

24. L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1:77–85, 1986.

25. L. Lamport. On interprocess communication, part II: Algorithms. *Distributed Computing*, 1:86–101, 1986.

26. K. Marzullo and L.S. Sabel. Efficient detection of a class of stable properities. *Distributed Computing*, 8:81–91, 1994.

27. F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms, International Workshop, Proceedings*, pages 215–226, Gers, France, October 1988. Elsevier Science Publishers B.V., Amsterdam, North-Holland, The Netherlands, 1989.

28. F. Mattern. On the relativistic structure of logical time in distributed systems. *Bigre*, 78:3–20, March 1992. Proceedings of the workshop: Datation et Contrôle des Exécutions Réparties, December 1991, Rennes, France. This paper is also available at URL: http://www.informatik.th-darmstadt.de/VS/Publikationen/.

29. S. Pilarski and T. Kameda. Checkpointing for distributed databases: Starting from the basics. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):602–610, 1992.

30. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, March 1994.

31. R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, B. Silvermann, D. Yellin, J. Russell, and S. Yemini. Hermes: Unix user's guide, version 0.8alpha. Technical report, IBM T.J.Watson Research Center, Yorktown Heights, NY, March 1992.

32. D.J. Taylor. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference,* Volume I, pages 29–42, Toronto, Ontario, Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

33. D. Zernik, M. Snir, and D. Malki. Using visualization tools to understand concurrency. *IEEE Software*, 9(3):87–92, May 1992.