EINDHOVEN UNIVERSITY OF TECHNOLOGY

Time and order of abstract events in distributed computations

Citation for published version (APA):
Basten, T., Kunz, T. H., Black, J. P., & Taylor, D. J. (1994). *Time and order of abstract events in distributed computations*. (Computing science notes; Vol. 9406). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1994

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Eindhoven University of Technology

Department of Mathematics and Computing Science

Time and the Order of Abstract Events in
Distributed Computations

by

T. Basten, T. Kunz, J. Black,
M. Coffin and D. Taylor

94/06

Computing Science Note 94/06
Eindhoven, February 1994

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Time and the Order of Abstract Events in Distributed Computations *

Twan Basten[1], Thomas Kunz[2], James Black[3], Michael Coffin[3], and David Taylor[3]

[1] Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany
[3] Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

Abstract

An important problem in event-based models of distributed computations is the amount of behavioral information. 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. Logical vector time and a derived notion called *reversed* vector time can be used to assign two meaningful timestamps to abstract events. 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. For this class, ordinary vector time is sufficient to determine causal relationships. At the cost of some extra computational effort, the timestamps derived for convex abstract events can also be used for arbitrary abstract events, overcoming the need for reversed timestamping.

Key words: Distributed systems – Event abstraction – Causality – Precedence relation – Partial order – Vector 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 the exchange of messages. Following Lamport [15], a distributed computation is modeled as an ordered 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. This is true even if, because of clock skew, the time at which the send event occurs is larger than the time at which the receive event occurs, as measured by their respective local clocks. However, sending a message might be unrelated to a write action on a file local to another process. Neither event precedes the other and they are said to be concurrent. Lamport [15] has shown that causality among primitive events can be accurately modeled by a *partial order*.

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

To determine causal relationships between events, logical-timestamp schemes have been proposed [10, 12, 15, 16, 17]. Logical time has been used for many different purposes: implementing causal broadcasts [4], measuring concurrency [6], detecting global predicates [9], implementing distributed breakpoints [13], computing consistent global snapshots [16], and visualizing program behavior [19].

However, experience shows that even for simple distributed applications, the amount of behavioral information is very large, and the causality structure is very complex. Applications that must cope with this huge amount of information become slow and often need many resources (disk storage, memory, processor time). In addition, the programmer often has difficulties managing too much behavioral information. Therefore, it is desirable to reduce the amount of information that must be considered at once.

A powerful way to reduce the apparent complexity of a computation is *abstraction* [5]. 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. Causality among abstract events is defined, and timestamp schemes are derived to efficiently determine causal relationships between abstract events. Each timestamp scheme is formally proven correct. Other important issues in event abstraction, such as specifying abstract behavior and the automatic recognition of abstract events to construct abstract views [1, 2], are beyond the scope of this paper.

Given a hierarchy of abstract descriptions of program behavior, an application can be evaluated at an arbitrary level. Using the timestamp schemes presented in this paper, program behavior can be visualized or even simulated at any level of abstraction. Most of the applications of timestamps mentioned above can be adapted to use abstract descriptions of program behavior instead of primitive descriptions. Doing so, performance can be increased and only information that is of interest to the user is taken into consideration.

To date, there has been no sound theoretical treatment in the literature of causality and logical time for arbitrary abstract events. Previous work at Waterloo [8, 14, 18] discusses abstract events with restrictive structural properties only. Event sets with these structural properties create abstract views of an execution that conform to the partial order among primitive events. Timestamping algorithms for these abstract events are given. While such structures are attractive because the set of abstract events is again partially ordered, it appears that they are not general enough to be useful; many apparently intuitive abstract events do not satisfy the constraints (see also [14]). Thus, this paper takes another approach, starting with no structural requirements on abstract events, and recognizing that it is not feasible in practice to maintain a partial order on them.

The paper is organized as follows. Section 2 presents a formal model of distributed computations and summarizes the basic definitions and results about logical vector time. Section 3 discusses causality among abstract events. In particular, it defines a precedence relation on abstract events. Section 4 derives a number of precedence tests for arbitrary abstract events, using vector time as introduced for primitive events. This section also introduces the notion of *reversed vector time*, which is essential to timestamping arbitrary abstract events efficiently. Section 5 deals with an important subclass of abstract events, called convex abstract events. It is shown that for this class of abstract events, timestamping schemes can be derived that do not depend on reversed vector time. At the cost of some extra computational effort, these timestamping schemes can also be used for arbitrary abstract events. Finally, Section 6 summarizes the results.

2 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 program* is a set of independent, cooperating program modules. Information is exchanged only by message passing. Both synchronous and asynchronous communication are allowed. To prevent unnecessarily complex formulas, 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. Again 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.

2.1 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. Essentially, a distributed computation is a pair $(E, \preceq)$, where E is the set of primitive events and $\preceq$ is an ordering relation that models causal precedence.

The detailed model given in the remainder of this subsection is based mainly on previous definitions of Mattern [16, 17], Charron-Bost [6, 7], and Fidge [11, 12]. Their definitions are, in turn, based on the "happens before" relation introduced by Lamport [15]. The algebraic structure underlying the model is Winskel's elementary event structure [20].

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.* Since this paper discusses timestamp schemes, this is not a real restriction: 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 $\preceq$ that models the causal ordering of events is defined as the smallest reflexive and transitive relation that satisfies the following two conditions.

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

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

Since the restrictions on Γ do not guarantee the absence of cycles in the precedence relation, the set of definitions given so far models a distributed computation if and only if the precedence relation

is a partial order.

The precedence relation extends the "happens before" relation as defined by Lamport [15] to synchronous communication in a natural way. The condition **C2**, originally given by Fidge [11, 12], means that a synchronous communication can be interpreted as if it occurred instantaneously. 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 other models of synchronous communication in the literature. Charron-Bost et al. [6], Cheung [8], and Fidge [11, 12] model synchronous communication as a pair of unrelated events. This has the disadvantage that a synchronous communication cannot be distinguished from a pair of concurrent events. Summers [18] models synchronous communication as a pair of mutually related events. A drawback of this model is that the precedence relation is no longer a partial order, which raises theoretical problems. In [2], synchronous communication is modeled as a single event occurring in two processes simultaneously. This model is more abstract than the model presented above. Unfortunately, it has some theoretical problems that will be discussed shortly.

The relation $\preceq$ can be used to express concurrency in a natural way. 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 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 proofs of many results to follow depend on the introduction of cuts. The following definition and theorem are due to Mattern [16, 17].

Definition 2.1. (Cut) A set $C \subseteq E$ is called a *cut* of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \preceq_l e_0 \Rightarrow e_1 \in C$, where $\preceq_l$ is the reflexive closure of the local ordering $\prec_l$. A cut is said to be *left-closed* under $\preceq_l$. The set of all cuts is denoted by $C_{\preceq_l}$.

Theorem 2.2. (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 2.3. (Consistent cut, [6, 16, 17, 20]) A set $C \subseteq E$ is called a *consistent cut* of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \preceq e_0 \Rightarrow e_1 \in C$. A consistent cut is left-closed under $\preceq$. The set of all consistent cuts of a distributed computation is denoted by $C_{\preceq}$.

Theorem 2.4. (Structure of consistent cuts, [16, 17, 20]) *The set of consistent cuts, with the ordering defined by $\subseteq$, is a complete lattice.*

2.2 Vector time

As explained before, 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 [16, 17] and Fidge [10, 12] independently introduced partially ordered vector time. Vector time extends the idea of logical clocks introduced by Lamport [15].

In this subsection, we summarize some definitions and results given by Mattern in [17]. They form the basis for the remaining sections of this paper that discuss timestamping abstract events or *sets* of primitive events.

An event causally precedes another event if and only if all its predecessors are also predecessors of the other event. That is, an event precedes another event if and only if the cut of all its predecessors is a subset of the cut of all predecessors of the other event. 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 1 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.

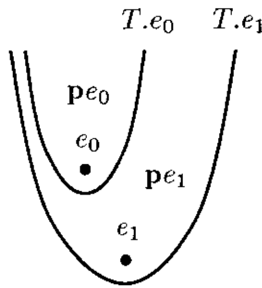

Figure 1: Precedence between primitive events.

Definition 2.5. (Causal past, [6, 17, 20]) 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.

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

Definition 2.6. (Causal past in a process, [6]) 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 as follows. For any $e \in E$, $\mathbf{p}_i e = \{e_0 \in E_i \mid e_0 \preceq e\}$. Note that $\mathbf{p}_i e = \mathbf{p}e \cap E_i$ and that $\mathbf{p}_i e$ is a cut, but not necessarily a consistent cut.

The two definitions of causal past and causal past in a process are essential to the results presented in this paper. In [2], some of these results are given for a slightly different model of distributed computations that models synchronous communication as a single event occurring in two processes simultaneously. As mentioned, this model raises some theoretical problems. In particular, since sets of events representing processes are not necessarily disjoint, the assumption that the causal past in a process is a cut is, in general, not true. Consider, for example, a synchronous-communication event e that is an element of two sets E_i and E_j for distinct i and j in $\mathcal{P}$. If e has a predecessor in process j, then the causal past in process i, $\mathbf{p}_i e$, is not a cut. The set $\mathbf{p}_i e$ is not left-closed under $\preceq_l$.

As a consequence, some definitions and derivations given in [2] are not correct. However, it is believed that the definitions and derivations can be adapted in such a way that the results remain valid. Another possibility is to adapt the model of synchronous communication so that sets of events representing processes are disjoint. This is the approach pursued in this paper.

Definition 2.5 yields the following useful result.

Corollary 2.7. [17, 20] *For any event* $e \in E$ *and any consistent cut* $C \in C_{\preceq}$, $e \in C \Leftrightarrow \mathbf{p}e \subseteq C$.

Proof. It follows from Definitions 2.3 (Consistent cut) and 2.5 (Causal past). □

Corollary 2.7 states that the causal past of an event is the smallest possible consistent cut that contains the event. This result is not very surprising, since the causal past of an event exactly reflects all its predecessors.

Corollary 2.8. *For any process* $i \in \mathcal{P}$, *event* $e \in E_i$, *and any cut* $C \in C_{\preceq_i}$, $e \in C \Leftrightarrow \mathbf{p}_i e \subseteq C$.

Proof. It follows from Definitions 2.1 (Cut) and 2.6 (Causal past in a process). □

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 every component $i \in \mathcal{P}$ of the timestamp is equal to the number of predecessors of the event in process i.

Definition 2.9. (Timestamp function) The function $T : E \longrightarrow \mathbb{N}^N$ defines a timestamp for every event as follows. For any event $e \in E$ and process $i \in \mathcal{P}$, $T.e.i = |\mathbf{p}_i e|$.

Note that 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 [12].

The following theorem shows that the timestamps as defined above can be used to determine the causal relation between primitive events. For two vectors $t_0, t_1 \in \mathbb{N}^N$, we define $t_0 \leq t_1$ to mean $t_0.i \leq t_1.i$ for all i, $0 \leq i < N$.

Theorem 2.10. (Precedence test for primitive events [16, 17]**)** *For any events* $e_0, e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0 \leq T.e_1$.

Proof. [6] Assume that $e_0 \preceq e_1$. Since $\preceq$ is transitive, it follows that, for any process $i \in \mathcal{P}$, $\mathbf{p}_i e_0 \subseteq \mathbf{p}_i e_1$. Definition 2.9 (Timestamp function) yields $T.e_0 \leq T.e_1$. It follows from the initial assumption that $e_0 \preceq e_1 \Rightarrow T.e_0 \leq T.e_1$.

Assume $e_0 \in E_i$, for some process $\subset\in \mathcal{P}$, and $e_1 \in E$ such that $e_0 \not\preceq e_1$. It follows that $\mathbf{p}_i e_1 \subset \mathbf{p}_i e_0$. Definition 2.9 (Timestamp function) yields $T.e_1.i < T.e_0.i$. This gives $T.e_0 \not\leq T.e_1$. It follows from the assumption that $T.e_0 \leq T.e_1 \Rightarrow e_0 \preceq e_1$. □

This precedence test formalizes the visualization of precedence given in Figure 1. 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 2.11. (Precedence test for primitive events [16]) *For any $i \in \mathcal{P}$ and events $e_0 \in E_i$ and $e_1 \in E$, $e_0 \preceq e_1 \Leftrightarrow T.e_0.i \leq T.e_1.i$.*

Proof. Assume that $e_0 \preceq e_1$. It follows that $\mathbf{p}_i e_0 \subseteq \mathbf{p}_i e_1$. This gives $T.e_0.i \leq T.e_1.i$. Assume that $e_0 \not\preceq e_1$. This yields $\mathbf{p}_i e_1 \subset \mathbf{p}_i e_0$. Hence $T.e_1.i < T.e_0.i$. □

This theorem shows that only one integer comparison is needed to decide whether an event precedes another if the process in which the event occurs is known.

An example of the assignment of vector timestamps to events is given in Figure 2. For this example, the validity of the two precedence tests given above is easily verified. The vertical lines represent processes. Time increases from top to bottom. Events are depicted as dots. The arrows represent the communication relation Γ. A synchronous communication is represented by a horizontal arrow.

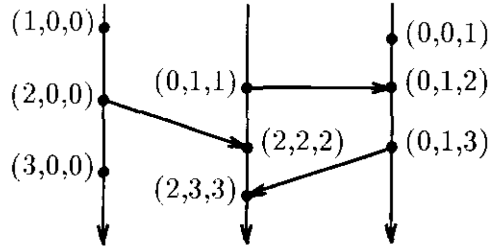

Figure 2: Timestamping events in a distributed computation.

An algorithmic calculation of vector timestamps is based on a system of counters, one for each process. Every time an event occurs in a process the local counter is incremented by one. The *global* time corresponding to any cut of the computation consists of the vector of all the counter values. Every process $i \in \mathcal{P}$ is assigned a vector $\mathcal{C}_i$ of size N which is its local clock. This local clock contains the local approximation to global time. Communication provides a way to update local clocks with information present in other processes. Every event is assigned a vector timestamp that is equal to the local clock value at the time of its occurrence. The precise rules for maintaining the local clocks in such a way that the timestamps satisfy the function T are as follows. The rules adapt the definitions of Fidge [10, 11, 12] and Mattern [16, 17] for synchronous communication. Small variations on the rules can be found in [5, 8, 18].

1. *Initialization*:
   Initially, all the local clocks are set to the zero vector.
2. *Ticking*:
   At the occurrence of an event in process $i \in \mathcal{P}$, component i of the local clock, $\mathcal{C}_i.i$, is incremented by one.
3. *Asynchronous communication*:
   At the occurrence of an asynchronous message receipt, first rule 2 is applied and then the receiving process $i \in \mathcal{P}$ assigns to its local clock the componentwise maximum of $\mathcal{C}_i$ and $T.e$, where $e \in E$ is the corresponding send event.
4. *Synchronous communication*:
   When some process $i \in \mathcal{P}$ sends a synchronous message to another process $j \in \mathcal{P}$, first process

i applies rule 2 and then assigns to its local clock the componentwise maximum of $\mathcal{C}_i$ and $\mathcal{C}_j$. Subsequently, process j applies rule 2 and assigns to its local clock the componentwise maximum of $\mathcal{C}_i$ and $\mathcal{C}_j$. Finally, *after* timestamping the two events, process i increments component j of its local clock, $\mathcal{C}_i.j$, by one.

In the remainder of this subsection, the notion of global time is formalized. It is shown that global time vectors have a structure that is isomorphic to the structure of cuts. This property is needed in the remaining sections when causal relations between abstract events are investigated. As mentioned, the global time at any point during a computation consists of the vector of all the local counter values. This is formalized in the following definition.

Definition 2.12. (Global time of a cut, [17]) The function $\mathcal{T} : C_{\preceq_l} \rightarrow \mathbb{N}^N$ defines the global time of a cut. For any cut C, component i, where $0 \leq i < N$, of the time vector is defined as $\mathcal{T}.C.i = |C \cap E_i|$. The set of all global time vectors of a computation, i.e., $\{\mathcal{T}.C \mid C \in C_{\preceq_l}\}$, is denoted by $T_{\preceq_l}$.

This definition has some interesting consequences.

Corollary 2.13. *For any $i \in \mathcal{P}$ and any event $e \in E$,*

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

Proof. It follows from Definitions 2.5 (Causal past), 2.6 (Causal past in a process), and 2.12 (Global time of a cut). □

Corollary 2.14. [17] *For any event $e \in E$, $\mathcal{T}.\mathbf{p}e = \mathcal{T}.e$.*

Proof. It follows from Definitions 2.5 (Causal past), 2.9 (Timestamp function), and 2.12 (Global time of a cut). □

Corollary 2.14 states that the timestamp of an event reflects its causal past. This is exactly what is shown in Figure 1.

Corollary 2.15. [17] *For any cuts $C_0, C_1 \in C_{\preceq_l}$, $C_0 \subseteq C_1 \Leftrightarrow \mathcal{T}.C_0 \leq \mathcal{T}.C_1$.*

Proof. It follows from Definitions 2.1 (Cut) and 2.12 (Global time of a cut). □

Corollary 2.15 implies that the function $\mathcal{T}$ is an isomorphism between the two complete lattices $(C_{\preceq_l}, \subseteq)$ and $(T_{\preceq_l}, \leq)$, as well as the following.

Theorem 2.16. (Structure of time vectors, [17]) *The set of time vectors, $T_{\preceq_l}$, with the ordering defined by $\leq$, forms a complete lattice. It is isomorphic to the lattice $(C_{\preceq_l}, \subseteq)$.*

Proof. It follows immediately from Theorem 2.2 (Structure of cuts) and Corollary 2.15. □

Theorem 2.17. (Structure of consistent time vectors) *The set of* consistent *time vectors $T_{\preceq} = \{\mathcal{T}.C \mid C \in C_{\preceq}\}$, with the ordering defined by $\leq$, forms a complete lattice. It is isomorphic to $(C_{\preceq}, \subseteq)$.*

Proof. It follows from Theorem 2.4 (Structure of consistent cuts) and Corollary 2.15. □

Note that the set of consistent time vectors represents exactly the set of global states in which the computation might be during an actual execution.

The above results are established by defining an isomorphism between the lattices of cuts and time vectors. The infimum and the supremum of sets 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 $\mathcal{T}.(\cup\ c:\ c \in C:\ c)$ respectively. 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 2.1 (Cut) and 2.12 (Global time) that $\mathcal{T}.(\cap\ c:\ c \in C:\ c) = (\mathrm{INF}\ c:\ c \in C:\ \mathcal{T}.c)$ and $\mathcal{T}.(\cup\ c:\ c \in C:\ c) = (\mathrm{SUP}\ c:\ c \in C:\ \mathcal{T}.c)$. In other words, the infimum and supremum of sets of time vectors are defined by INF and SUP respectively.

This concludes the introduction to vector time. The definitions and basic results given above provide a basis for the next sections.

3 Abstract events and causality

In a hierarchy of abstract descriptions of program behavior, 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 often represented by non-empty sets of primitive events. Only when necessary is an abstract event represented in terms of its elements in the level below. For this purpose, the term "constituents" is used exclusively.

An important part of an abstract description of behavior is a representation of causality among abstract events. The causality structure is defined by a precedence relation. For reasons of mathematical and practical convenience, it would be preferable if this relation had the same structural properties as the precedence relation on primitive events, i.e. if it were a partial order. There are two obvious possibilities to define precedence.

The first possibility is to specify that an abstract event A precedes an abstract event B if and only if every event in A precedes every event in B. This definition has one advantage: it guarantees anti-symmetry and transitivity. Taking the reflexive closure yields a precedence relation on abstract events that is a partial order. However, it also has two important disadvantages. First, 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 the same 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. Second, this definition of precedence is too restrictive to be useful. Too many abstract events may be unrelated, implying that the first level of abstraction might abstract away too much structure. Higher levels of abstraction might even contain only unrelated events. This defeats the purpose of behavioral abstraction that focuses on causal relations among events. Universal quantification seems to be too strong a requirement.

The second possibility is to specify that an abstract event A precedes an abstract event B if and only if there exists an event in A that precedes at least one event in B. Intuitively, this is also a meaningful definition, since at least part of A happens before B is completed. This definition has one important disadvantage. In general, the precedence relation on abstract events is no longer a partial order. The relation usually is neither anti-symmetric nor transitive. An advantage is that

concurrency conforms to its intuitive meaning. Two abstract events A and B are causally unrelated if and only if every event in A is causally unrelated to every event in B. An additional advantage is that abstract behavior maintains a lot of the structure of underlying levels of description. Although this choice does, in general, not yield a precedence relation which is a partial order, it seems to be the only one that is both meaningful and practically useful. Therefore, precedence among abstract events is formally defined as follows.

Definition 3.1. (Precedence between abstract events) Let A and B be two abstract events. Precedence is defined as $A \preceq B \Leftrightarrow (\exists\, a : a \in A : (\exists\, b : b \in B : a \preceq b))$

To represent causality concisely, it seems a logical choice to investigate the use of vector timestamps. The goal is to derive precedence tests similar to the ones for primitive events. Since the precedence relation on abstract events is not necessarily a partial order, two approaches are possible.

The first approach is to restrict the structure of abstract events so that the precedence relation is guaranteed to be a partial order, and then to try to timestamp these restricted abstract events in a meaningful way. This is the approach taken by Cheung [8], Kunz [14], and Summers [18]. Unfortunately, imposing structural requirements that guarantee transitivity and anti-symmetry either makes the resulting abstractions difficult to manage or severely limits their expressiveness. The second approach, pursued in this paper, is to allow more flexible abstract-event structures at the cost of increasing the difficulty of obtaining and comparing timestamp vectors. By presenting timestamps for two classes of abstract events, this paper points out the trade-offs in structural flexibility and timestamping effort.

Before giving any results, it is important to have criteria to evaluate timestamps and precedence tests. First, 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. Second, in a hierarchy of true abstractions, a representation 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 abstract event from the timestamps of its constituents. 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. For most of the definitions and results in the remaining sections, it is usually clear whether they satisfy the second criterion. Formal proofs are only given on a few occasions.

4 Timestamping arbitrary abstract events

This section discusses timestamps and precedence tests for arbitrary abstract events. Section 4.1 starts by introducing some new definitions and adapting some previous definitions to abstract events. In Section 4.2, a timestamp is derived that represents the causal past of an abstract event. However, precedence tests using this timestamp do not satisfy the criteria for such tests. Section 4.3 then introduces the notions of causal future and reversed vector time. In Section 4.4, these concepts are used to derive another timestamp for abstract events representing the causal future. Using the two timestamps for abstract events, precedence tests are derived that satisfy the criteria mentioned above.

4.1 Basic definitions and results

Definition 4.1. (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 $e \in E$, $\boldsymbol{l}e = \{i \in \mathcal{P} \mid e \in E_i\}$. 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 always is a singleton.

A property of primitive events that no longer holds for abstract events is atomicity. Consequently, abstract events have a duration. This is expressed by the following two functions.

Definition 4.2. (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 whether an abstract event A precedes another abstract event B, it is sufficient to consider the beginning of A and the end of B instead of A and B in their entirety.

Definition 4.3. (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 $A \subseteq E$, $\mathbf{p}A = (\cup\ a :\ a \in A :\ \mathbf{p}a)$. Note that the causal past of an abstract event is a consistent cut.

Definition 4.4. (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 $A \subseteq E$, $\mathbf{p}_i A = (\cup\ a :\ a \in A :\ \mathbf{p}_i a)$. Note that $\mathbf{p}_i A = \mathbf{p}A \cap E_i$ and that $\mathbf{p}_i$A is a cut, not necessarily consistent.

The next corollary is a result of the previous definitions. It shows that the function $\lceil . \rceil$ really models the end of an abstract event.

Corollary 4.5. *For an abstract event A, $\mathbf{p}\lceil A \rceil = \mathbf{p}A$.*

Corollary 4.5 states that the past of the end of an abstract event corresponds to the past of the completed event. Unfortunately, there is no such simple result relating causal past to the beginning of an abstract event.

The last two results of this subsection are expressions for the time of the causal past of an abstract event. The next corollary is a direct consequence of Definition 4.4 (Causal past of an abstract event in a process) and Definition 2.12 (Global time of a cut). It is a generalization of the result as derived for primitive events (Corollary 2.13).

Corollary 4.6. *For any process $i \in \mathcal{P}$ and abstract event A,*

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

The definition of the causal past of an abstract event can be used to derive the last result of this subsection.

Property 4.7. *For any abstract event A, $\mathcal{T}.\mathbf{p}A = (\mathrm{SUP}\ a :\ a \in A :\ \mathcal{T}.a)$.*

Proof.

$\mathcal{T}.\mathbf{p}A$
$=$ { Definition 4.3 (Causal past of an abstract event) }
$\mathcal{T}.(\cup\, a:\ a \in A:\ \mathbf{p}a)$
$=$ { Theorem 2.17 (Structure of consistent time vectors) }
$(\mathrm{SUP}\ a:\ a \in A:\ \mathcal{T}.\mathbf{p}a)$
$=$ { Corollary 2.14 }
$(\mathrm{SUP}\ a:\ a \in A:\ T.a)$ □

This result is useful; combining it with Corollary 4.5 yields that there is a simple expression in terms of timestamps that marks the end of an abstract event. Since Corollary 4.5 has no equivalent for the beginning of an abstract event, it is not possible to obtain a similar result for the beginning. Such a result may not be necessary but would at least be convenient to express causal relations between abstract events. One way to get a similar result for the beginning of an abstract event is to introduce the notion of causal future. As shown in the following, this is the key to deriving precedence tests that satisfy the two criteria stated in the previous section. First, however, precedence among abstract events is characterized in the current framework.

4.2 A timestamp for abstract events

In this subsection, a timestamp and some precedence tests for abstract events are derived. As mentioned, precedence tests are evaluated according to their efficiency and hierarchical applicability.

The efficiency of precedence tests in terms of the number of integer comparisons is compared to the efficiency of the straightforward precedence test that is a direct result of the definition of the precedence relation on abstract events. To determine whether an abstract event A precedes an abstract event B, one could simply check all the primitive events until a primitive event in A is found that precedes a primitive event in B. If the precedence test of Theorem 2.10 is used, this takes at most $|A| \cdot |B| \cdot N$ integer comparisons since it does not make use of the location set of primitive events. Obviously, this is not very efficient.

The second criterion for precedence tests is hierarchical applicability. Obviously, the abovementioned test depends on the primitive level of the computation. Therefore, it does not satisfy the second criterion.

One would expect that it is possible to do better. If abstract events are timestamped in a meaningful way, it should be possible to reduce the number of integer comparisons and perhaps make it independent of the primitive level of the computation. The time that marks the end of an abstract event stated in Property 4.7 seems to be a good candidate for a timestamp. The derivation below shows that this vector can indeed be used as a timestamp. Let a be an event in E and let B be an abstract event.

$a \in \mathbf{p}B$
$\Leftrightarrow$ { $\mathbf{p}B$ is a consistent cut; Corollary 2.7 }
$\mathbf{p}a \subseteq \mathbf{p}B$
$\Leftrightarrow$ { Theorem 2.17 (Structure of consistent time vectors) }
$\mathcal{T}.\mathbf{p}a \leq \mathcal{T}.\mathbf{p}B$
$\Leftrightarrow$ { Corollary 2.14; Property 4.7 }
$T.a \leq (\mathrm{SUP}\ b:\ b \in B:\ T.b)$

This derivation shows that a primitive event is an element of the causal past of an abstract event if and only if its timestamp is at most the time that marks the end of the abstract event. Since an abstract event A precedes an abstract event B if and only if at least one of its primitive events is an element of the causal past of B, it is useful to extend the timestamp function on primitive events to abstract events as follows.

Definition 4.8. (Timestamp of an abstract event) The function $T : 2^E \longrightarrow \mathbb{N}^N$ defines the timestamp of an abstract event as follows. For any $A \subseteq E$, $T.A = (\text{SUP } a : a \in A : T.a)$.

This timestamp is an efficient encoding of the end of an abstract event (see Figure 3). Furthermore, the associativity of the quantifier SUP implies that the timestamp of an abstract event is equal to the supremum of the timestamps of its constituents. Therefore, the introduction of this timestamp seems to be a step towards the fulfillment of the two criteria for precedence tests.

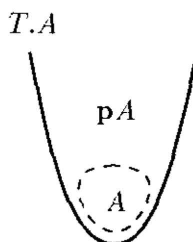

Figure 3: The meaning of the timestamp of an abstract event.

The following precedence test is a direct consequence of the derivation above and the definitions of the beginning of an abstract event (4.2) and the timestamp function (4.8).

Theorem 4.9. (Precedence test for abstract events) *For any abstract events A and B,* $A \preceq B \Leftrightarrow (\exists\, a : a \in \lfloor A \rfloor : T.a \leq T.B)$.

This test has two disadvantages. First, it is still not very efficient. In the worst case, the timestamp of each primitive event in $\lfloor A \rfloor$ must be compared with the timestamp of B, yielding $|\lfloor A \rfloor| \cdot N$ comparisons. Since the number of processes in the location set of A is an upper bound for the number of primitive events in the beginning, the maximum number of comparisons is $|lA| \cdot N$. Second, it still 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 of the computation to compute the beginning of the newly formed abstract event. Therefore, this test still does not satisfy either of the criteria for precedence tests, although it is considerably more efficient than the straightforward test mentioned earlier. An advantage of this test is that the only information needed to implement it is the set of timestamps of primitive events.

Two remarks about the test in Theorem 4.9 are in order. First, the test does not make use of the location set of an event. It is possible to reduce the maximum number of integer comparisons if this information is available. This is shown in the remainder of this subsection. Second, note that the test in Theorem 4.9 compares the beginning of abstract event A with the end of abstract event B. This has already been suggested as a possible improvement in determining precedence between abstract events. However, 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 in its timestamp. If the asymmetry can be resolved, this might lead to precedence tests that no longer depend on the primitive level of the computation. Once again, this suggests introducing the causal future of events.

The following derivation uses information about the location of events. Let a be a primitive event and B an abstract event. Let i be the process in $\mathbf{l}a$.

$$\begin{array}{ll} & a \in \mathbf{p}_i B \\ \Leftrightarrow & \{\ a \in E_i;\ \mathbf{p}_i B \text{ is a cut; Corollary 2.8 }\} \\ & \mathbf{p}_i a \subseteq \mathbf{p}_i B \\ \Leftrightarrow & \{\ \text{Theorem 2.16 (Structure of time) }\} \\ & \mathcal{T}.\mathbf{p}_i a \leq \mathcal{T}.\mathbf{p}_i B \\ \Leftrightarrow & \{\ \text{Corollary 2.13; Corollary 4.6 }\} \\ & \mathcal{T}.\mathbf{p}a.i \leq \mathcal{T}.\mathbf{p}B.i \\ \Leftrightarrow & \{\ \text{Corollary 2.14; Property 4.7; Definition 4.8 (Timestamp } T) \} \\ & T.a.i \leq T.B.i \end{array}$$

Theorem 4.10. (Precedence test for abstract events) *For any abstract events A and B,* $A \preceq B \Leftrightarrow (\exists\ i, a :\ i \in \mathcal{P}\ \wedge\ a \in \lfloor A \rfloor \cap E_i :\ T.a.i \leq T.B.i)$.

This test is more efficient than the previous one. In the worst case, the number of comparisons is $|\lfloor A \rfloor|$, which in turn has an upper bound $|\mathbf{l}A|$. However, it still has the disadvantage that it depends on the primitive level of the computation. There seems to be no obvious way to overcome this problem within the current framework. Therefore, the next step is to extend the framework in a useful and meaningful way. Until now, only the past of events has been considered. The next step is to formalize the notion of the causal future of events.

4.3 Causal future and reversed vector time

In this subsection, the causal future is introduced for primitive events. The goal is two-fold. The first goal is to derive new precedence tests for primitive events in terms of causal future. The second goal is to find an efficient way to represent the causal future, in the same way that timestamps represent the causal past.

The causal future is the dual of the causal past. Therefore, to define the causal future and explain its meaning in the entire framework, the duals of the relations $\preceq$ and $\preceq_l$ are needed. The successor relation $\succeq$ is defined as the dual of $\preceq$. That is, for any $e_0, e_1 \in E$, $e_0 \succeq e_1$ if and only if $e_1 \preceq e_0$. The local successor relation $\succeq_l$ is the dual of $\preceq_l$. Before the definition of causal future is given, some related concepts are defined that clarify its relation with previous definitions.

Definition 4.11. (Successor cut) A set $C \subseteq E$ is called a successor cut, or $\succeq$-cut, of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \succeq_l e_0 \Rightarrow e_1 \in C$. A $\succeq$-cut is said to be left-closed under $\succeq_l$ or right-closed under $\preceq_l$. The set of all $\succeq$-cuts is denoted by $C_{\succeq_l}$.

Definition 4.12. (Consistent successor cut) A set $C \subseteq E$ is called a *consistent* $\succeq$-cut of E if and only if for all events $e_0 \in C$ and $e_1 \in E$, $e_1 \succeq e_0 \Rightarrow e_1 \in C$. A consistent $\succeq$-cut is said to be left-closed under $\succeq$. The set of all consistent $\succeq$-cuts is denoted by $C_{\succeq}$.

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

Corollary 4.13. *Let C be a subset of E.*
$C \in C_{\preceq_l} \Leftrightarrow E \backslash C \in C_{\succeq_l}$,
$C \in C_{\preceq} \Leftrightarrow E \backslash C \in C_{\succeq}$.

Definition 4.14. (Causal future) The function $\mathbf{f}. : E \longrightarrow 2^E$ defines the causal future of an event as follows. For any $e \in E$, $\mathbf{f}e = \{e_0 \in E \mid e_0 \succeq e\}$. Note that $\mathbf{f}e$ is a consistent $\succeq$-cut.

Definition 4.15. (Causal future in a process) For any process $i \in \mathcal{P}$, the function $\mathbf{f}_i. : E \longrightarrow 2^E$ defines the causal future in process i of an event as follows. For any $e \in E$, $\mathbf{f}_i e = \{e_0 \in E_i \mid e_0 \succeq e\}$. Note that $\mathbf{f}_i e = \mathbf{f}e \cap E_i$ and that $\mathbf{f}_i e$ is a $\succeq$-cut.

Since computations are finite, it is appropriate to define the following.

Definition 4.16. (Reversed vector time of a successor cut) The function $\mathcal{T}^R : C_{\succeq_l} \rightarrow \mathbb{N}^N$ defines the *reversed vector time* of a $\succeq$-cut. For any $\succeq$-cut C, component i, where $0 \leq i < N$, of the reversed time vector is defined as $\mathcal{T}^R.C.i = |C \cap E_i|$.

This definition is the first step towards an efficient representation of the causal future. The following corollary is a direct result of this definition, the definition of the time of a cut (2.12), and Corollary 4.13. It states the relation between vector time and reversed vector time. The binary operator "$-$" on vectors is componentwise subtraction.

Corollary 4.17. *For any successor cut $C \in C_{\succeq_l}$, $\mathcal{T}.(E \backslash C) = \mathcal{T}.E - \mathcal{T}^R.C$. For any cut $C \in C_{\preceq_l}$, $\mathcal{T}^R.(E \backslash C) = \mathcal{T}^R.E - \mathcal{T}.C$.*

Note that $\mathcal{T}.E$ and $\mathcal{T}^R.E$ are both equal to the vector in $\mathbb{N}^N$ whose i^{th} component is equal to the number of events in process i, i.e. $|E_i|$. In the following, this vector is denoted by $\mathbf{E}$ to emphasize that it is a constant.

For the derivation of precedence tests that make use of the location set, the cut $\mathbf{p}_i e$, for some $e \in E$ and $i \in \mathcal{P}$, is of particular interest. For this cut, the corollary above yields the following equation: $\mathcal{T}^R.(E \backslash \mathbf{p}_i e) = \mathbf{E} - \mathcal{T}.\mathbf{p}_i e$. If e is an event in E_i, then the left-hand side of this equation can, for component i, be simplified as shown in the following corollary.

Corollary 4.18. *For any $i \in \mathcal{P}$ and $e \in E_i$, $\mathcal{T}^R.\mathbf{f}_i e.i - 1 = \mathbf{E}.i - \mathcal{T}.\mathbf{p}_i e.i$*

The following derivation shows the meaning of precedence in terms of causal past and causal future. Let e_0 and e_1 be events in E.

$e_0 \preceq e_1$
$\Leftrightarrow$ { The relation $\preceq$ is reflexive and transitive }
$(\exists\, e : e \in E : e_0 \preceq e \ \wedge\ e \preceq e_1)$
$\Leftrightarrow$ { Definitions 4.14 (Causal future) and 2.5 (Causal past) }
$(\exists\, e : e \in E : e \in \mathbf{f}e_0 \ \wedge\ e \in \mathbf{p}e_1)$
$\Leftrightarrow$ { Definition of set intersection }
$\mathbf{f}e_0 \cap \mathbf{p}e_1 \neq \emptyset$

The last expression in this derivation states that event e_0 precedes event e_1 if and only if the future of e_0 and the past of e_1 overlap. If this derivation is continued, a new precedence test can be derived.

$$
\begin{aligned}
& \mathbf{f}e_0 \cap \mathbf{p}e_1 \neq \emptyset \\
\Leftrightarrow \quad & \{ \text{ Set calculus; } \mathbf{f}e_0 \subseteq E \text{ and } \mathbf{p}e_1 \subseteq E \} \\
& E \backslash \mathbf{f}e_0 \not\supseteq \mathbf{p}e_1 \\
\Leftrightarrow \quad & \{ \text{ Corollary 4.13 and Theorem 2.17 (Structure of consistent time vectors) } \} \\
& \mathcal{T}.(E \backslash \mathbf{f}e_0) \not\geq \mathcal{T}.\mathbf{p}e_1 \\
\Leftrightarrow \quad & \{ \text{ Corollary 4.17 } \} \\
& \mathbf{E} - \mathcal{T}^R.\mathbf{f}e_0 \not\geq \mathcal{T}.\mathbf{p}e_1
\end{aligned}
$$

Note that there are two possibilities for continuing this derivation in the first step. It is easy to verify that the final result is the same. Derivations similar to the two above can be given if the location set of e_0 or e_1 is known.

Theorem 4.19. *Let i and j be processes in $\mathcal{P}$ and let e_0 and e_1 be events in E_i and E_j respectively.*

$$
\begin{aligned}
e_0 \preceq e_1 \quad & \Leftrightarrow \quad \mathbf{E} - \mathcal{T}^R.\mathbf{f}e_0 \not\geq \mathcal{T}.\mathbf{p}e_1 \\
& \Leftrightarrow \quad \mathbf{E}.i - \mathcal{T}^R.\mathbf{f}e_0.i < \mathcal{T}.\mathbf{p}e_1.i \\
& \Leftrightarrow \quad \mathbf{E}.j - \mathcal{T}^R.\mathbf{f}e_0.j < \mathcal{T}.\mathbf{p}e_1.j
\end{aligned}
$$

For natural numbers, the relations $\not\geq$ and $<$ are equivalent. In the last two expressions of this theorem, therefore, the relation $<$ appears. Furthermore, Corollaries 4.18 and 2.13 yield, for any process i in $\mathcal{P}$ and event e in E_i, the equation $\mathbf{E}.i - \mathcal{T}^R.\mathbf{f}e.i = \mathcal{T}.\mathbf{p}e.i - 1$. This equality and Corollary 2.14 applied to the second equivalence of the last theorem yield once again the precedence test of Theorem 2.11.

There is also another way to look at these precedence tests. The equivalences stated in this theorem are useful for determining the precedence between two events if there is an efficient representation of the reversed vector time of the causal future of events, similar to the representation of the vector time of the causal past of events in Corollary 2.14. So the question is whether there exists a function $T^R : E \longrightarrow \mathbb{N}^N$ that can be calculated without too much overhead, such that, for any $e \in E$, $\mathcal{T}^R.\mathbf{f}e = T^R.e$. The answer is yes.

Definition 4.20. (Timestamp function T^R) The function $T^R : E \longrightarrow \mathbb{N}^N$ defines a timestamp in reversed vector time for every event as follows. For any event $e \in E$ and process $i \in \mathcal{P}$, $T^R.e.i = |\mathbf{f}_i e|$.

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. The duality of the relations $\preceq$ and $\succeq$ immediately yields the desired result.

Corollary 4.21. *For any event $e \in E$, $\mathcal{T}^R.\mathbf{f}e = T^R.e$.*

Every result in Section 2.2 has a dual in the framework developed in this subsection. The following theorem summarizes all the precedence tests for primitive events derived thus far.

Theorem 4.22. (Precedence tests for primitive events) *Let i and j be processes in $\mathcal{P}$ and let e_0 and e_1 be events in E_i and E_j respectively.*

$$\begin{array}{rcl} e_0 \preceq e_1 & \Leftrightarrow & T.e_0 \ \leq \ T.e_1 \\ & \Leftrightarrow & T.e_0.i \ \leq \ T.e_1.i \\ & \Leftrightarrow & T^R.e_1 \ \leq \ T^R.e_0 \\ & \Leftrightarrow & T^R.e_1.j \ \leq \ T^R.e_0.j \\ & \Leftrightarrow & \mathbf{E} - T^R.e_0 \ \not\geq \ T.e_1 \\ & \Leftrightarrow & \mathbf{E}.i - T^R.e_0.i \ < \ T.e_1.i \\ & \Leftrightarrow & \mathbf{E}.j - T^R.e_0.j \ < \ T.e_1.j \end{array}$$

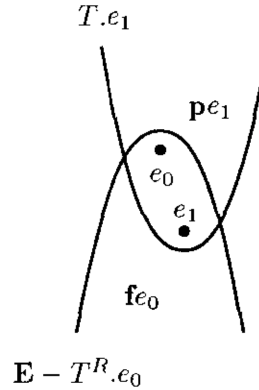

Figure 4: Precedence among primitive events in terms of vector time and reversed vector time.

The third and fourth tests are the duals of the first and second respectively. The last three tests are new. Figure 4 illustrates the meaning of these new tests. In terms of primitive events, nothing is gained by the three new tests. To make them useful, event information has to be traversed backwards while assigning every event a reversed timestamp. This is computationally expensive and requires extra storage. However, if these tests can be generalized to efficient precedence tests for abstract events that do not depend on primitive events, then the extra costs might be acceptable. Section 4.4 shows that such a generalization is indeed possible.

4.4 Another timestamp for abstract events

In this subsection, the new timestamp and precedence tests for primitive events are generalized to abstract events. The resulting tests are efficient and independent of the primitive level of the abstraction hierarchy. Like the tests for primitive events, the tests depend on reversed vector time. Before any precedence tests are derived, causal future is defined for abstract events.

Definition 4.23. (Causal future of an abstract event) The causal future of an abstract event is defined by a function $\mathbf{f}. : 2^E \longrightarrow 2^E$ as follows. For any $A \subseteq E$, $\mathbf{f}A = (\cup\ a :\ a \in A :\ \mathbf{f}a)$. Note that $\mathbf{f}A$ is a consistent $\succeq$-cut.

Definition 4.24. (Causal future of an abstract event in a process) For any $i \in \mathcal{P}$, the function $\mathbf{f}_i. : 2^E \longrightarrow 2^{E_i}$ defines the causal future in process i of an abstract event as follows. For any $A \subseteq E$, $\mathbf{f}_iA = (\cup\ a :\ a \in A :\ \mathbf{f}_ia)$. Note that $\mathbf{f}_iA = \mathbf{f}A \cap E_i$ and that $\mathbf{f}_iA$ is a $\succeq$-cut.

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

Corollary 4.25. *For an abstract event A, $\mathbf{f}\lfloor A \rfloor = \mathbf{f}A$.*

This corollary is the dual of Corollary 4.5. It states that the beginning of an abstract event and the event itself share the same causal future.

The next two results are expressions for the reversed time of the causal future of an abstract event.

Corollary 4.26. *For any process $i \in \mathcal{P}$ and abstract event A,*

$$\mathcal{T}^R.\mathbf{f}_i A.j = \begin{cases} \mathcal{T}^R.\mathbf{f}A.j, & \textit{for } j = i \\ 0, & \textit{otherwise} \end{cases}$$

Property 4.27. *For any abstract event A, $\mathcal{T}^R.\mathbf{f}A = (\mathrm{SUP}\ a :\ a \in A :\ \mathcal{T}^R.a)$.*

Property 4.27 gives a simple expression in terms of reversed vector time for the beginning of an abstract event. This is exactly the result we were looking for.

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.

$$
\begin{array}{ll}
& A \preceq B \\
\Leftrightarrow & \{ \text{ The relation } \preceq \text{ is reflexive and transitive; Definition 3.1 (Precedence) } \} \\
& (\exists\, e :\ e \in E :\ (\exists\, a :\ a \in A :\ a \preceq e) \ \wedge\ (\exists\, b :\ b \in B :\ e \preceq b)) \\
\Leftrightarrow & \{ \text{ Definitions 4.14 (Causal future) and 2.5 (Causal past) } \} \\
& (\exists\, e :\ e \in E :\ (\exists\, a :\ a \in A :\ e \in \mathbf{f}a) \ \wedge\ (\exists\, b :\ b \in B :\ e \in \mathbf{p}b)) \\
\Leftrightarrow & \{ \text{ Definition of set union } \} \\
& (\exists\, e :\ e \in E :\ e \in (\cup\, a :\ a \in A :\ \mathbf{f}a) \ \wedge\ e \in (\cup\, b :\ b \in B :\ \mathbf{p}b)) \\
\Leftrightarrow & \{ \text{ Definitions 4.23 (Causal future) and 4.3 (Causal past) } \} \\
& (\exists\, e :\ e \in E :\ e \in \mathbf{f}A \ \wedge\ e \in \mathbf{p}B) \\
\Leftrightarrow & \{ \text{ Definition of set intersection } \} \\
& \mathbf{f}A \cap \mathbf{p}B \neq \emptyset
\end{array}
$$

This result is similar to the result for primitive events. It states that A precedes B if and only if the future of A and the past of B overlap.

$$
\begin{array}{ll}
& \mathbf{f}A \cap \mathbf{p}B \neq \emptyset \\
\Leftrightarrow & \{ \text{ Set calculus; } \mathbf{f}A \subseteq E \text{ and } \mathbf{p}B \subseteq E \ \} \\
& E \backslash \mathbf{f}A \not\supseteq \mathbf{p}B \\
\Leftrightarrow & \{ \text{ Corollary 4.13 and Theorem 2.17 (Structure of consistent time vectors) } \} \\
& \mathcal{T}.(E \backslash \mathbf{f}A) \not\geq \mathcal{T}.\mathbf{p}B \\
\Leftrightarrow & \{ \text{ Corollary 4.17 } \} \\
& \mathbf{E} - \mathcal{T}^R.\mathbf{f}A \not\geq \mathcal{T}.\mathbf{p}B \\
\Leftrightarrow & \{ \text{ Property 4.27; Property 4.7; Definition 4.8 (Timestamp } T) \ \} \\
& \mathbf{E} - (\mathrm{SUP}\ a :\ a \in A :\ \mathcal{T}^R.a) \not\geq T.B
\end{array}
$$

This derivation suggests introducing a second timestamp for abstract events as follows.

Definition 4.28. (Reversed timestamp of an abstract event) The function $T^R : 2^E \longrightarrow \mathbb{N}^N$ defines the reversed timestamp of an abstract event as follows. For any $A \subseteq E$, $T^R.A = (\text{SUP } a : a \in A : T^R.a)$.

Like the other timestamp for abstract events, the reversed timestamp for an abstract event can be calculated from the timestamps of its constituents. It is an encoding of the beginning of the abstract event. Note that in an actual implementation, one probably wants to use $\mathbf{E} - T^R.A$ as the reversed timestamp for an abstract event A. That is, one wants to timestamp an abstract event with two times in vector time instead of one in vector time and one in reversed vector time. Figure 5 shows the meaning of the reversed timestamp of an abstract event. It extends Figure 3. An abstract event is determined by the two times that mark its beginning and end.

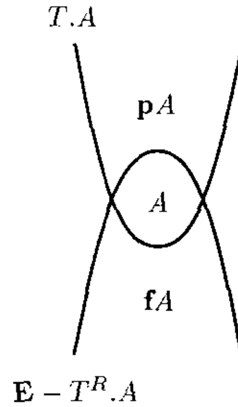

Figure 5: The meaning of the two timestamps of an abstract event.

The derivation above and the introduction of the reversed vector timestamp for abstract events yield a precedence test that is independent of the computation at the level of primitive events.

Theorem 4.29. *For any abstract events A and B, $A \preceq B \Leftrightarrow \mathbf{E} - T^R.A \not\geq T.B$.*

The number of integer comparisons for this test is at most N. Without further knowledge it is not possible to derive a more efficient test. Therefore, this is the first test that satisfies the two criteria for precedence tests for abstract events. Figure 6 illustrates this result. The solid lines depict $\mathbf{E} - T^R.A$ and $T.B$. The dashed lines depict $T.A$ and $\mathbf{E} - T^R.B$ which are not involved in deciding $A \preceq B$.

If the location set of an abstract event is known, derivations similar to the ones above yield the following result.

Theorem 4.30. *Let A and B be abstract events.*

$$\begin{aligned} A \preceq B &\Leftrightarrow (\exists\, i : i \in \boldsymbol{l}A : \mathbf{E}.i - T^R.A.i < T.B.i) \\ &\Leftrightarrow (\exists\, i : i \in \boldsymbol{l}B : \mathbf{E}.i - T^R.A.i < T.B.i) \end{aligned}$$

These tests are, in terms of integer comparisons, more efficient than the previous one. If the location set of abstract events is known, one only has to consider components of the time vectors within the location set of one of the abstract events. For the two tests, the maximum number of integer comparisons is $|\boldsymbol{l}A|$ and $|\boldsymbol{l}B|$ respectively. This is similar to the efficiency of the last test

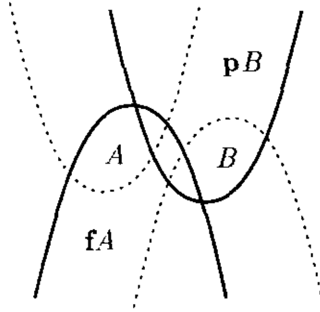

Figure 6: Precedence among abstract events in terms of vector time and reversed vector time.

derived in Section 4.2 (see Theorem 4.10). However, unlike this test, the tests in Theorem 4.30 are independent of the primitive level of the abstraction hierarchy.

Summarizing, at the cost of an extra timestamp for both primitive and abstract events, independence from the primitive abstraction level is gained. The question of which precedence test is most useful can only be answered in the context of a particular application. If many abstract events are formed in many different levels, then the tests derived in this subsection are probably more useful than the ones derived in Section 4.2. The next section even provides a third alternative based on the notion of *convex* abstract events.

5 Timestamping 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, timestamps and precedence tests for the subclass of *convex* abstract events are discussed.

Definition 5.1. (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 that is not a constituent of A but 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. Note that every non-convex abstract event implies a violation of the anti-symmetry requirement of a partial order.

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, complete precedence abstractions [18] and contractions [3, 8, 18]. Third, since there are no interfering events, convex abstract events are considerably easier to display than arbitrary abstract events. Finally, this section shows that determining causality among convex abstract events requires less timestamping effort than determining causality among arbitrary abstract events.

There does not seem to be another class of abstract events that combines all these properties. Therefore, the class of convex abstract events is an interesting class for further study.

In addition, convexity can be used to derive alternative timestamping schemes for arbitrary abstract events. The idea of using timestamping schemes for convex abstract events for arbitrary events is based on the following definition and resulting theorem.

Definition 5.2. (Convex closure) The convex closure of an abstract event A is defined by a function $\mathbf{c}. : 2^E \longrightarrow 2^E$ as the smallest convex set of events containing A.

Theorem 5.3. *For any abstract events A and B, $A \preceq B \Leftrightarrow \mathbf{c}A \preceq \mathbf{c}B$.*

Proof. Assume $A \preceq B$. Since A is a subset of $\mathbf{c}A$ and B a subset of $\mathbf{c}B$, the definition of precedence among abstract events (Definition 3.1) yields that $\mathbf{c}A \preceq \mathbf{c}B$.

Assume $\mathbf{c}A \preceq \mathbf{c}B$. Let $a \in \mathbf{c}A$ and $b \in \mathbf{c}B$ be two events such that $a \preceq b$. There are four possibilities: (1) $a \in A$ and $b \in B$; (2) $a \notin A$ and $b \in B$; (3) $a \in A$ and $b \notin B$; (4) $a \notin A$ and $b \notin B$. In case (1), it follows immediately that $A \preceq B$. In case (2), it follows from the definitions of convex abstract events (Definition 5.1) and the convex closure (Definition 5.2) that there is an event $e \in A$ such that $e \preceq a$. Since $a \preceq b$, e also precedes b. This yields $A \preceq B$. For cases (3) and (4), similar arguments can be given. □

Theorem 5.3 means that timestamp schemes derived for convex abstract events can be used for arbitrary abstract events as follows. For any abstract event, first calculate its convex closure and use the resulting convex event set to calculate timestamps according to some scheme for convex abstract events. Then, assign these timestamps to the possibly non convex abstract event. The remainder of this section shows that this is an interesting alternative to reversed timestamping.

The following property is the key to deriving an efficient precedence test for convex abstract events. It states that to determine the causal past of a convex abstract event A in a process i in its location set, it is sufficient to consider primitive events in the intersection of A and E_i. This is not necessarily true for an abstract event that is not convex: an event in E_i can succede all events in $\mathbf{p}_iA$ and precede some other event in $A \setminus E_i$.

Property 5.4. *Let A be a convex abstract event. For any $i \in lA$, $\mathbf{p}_iA = \mathbf{p}_i(A \cap E_i)$.*

Proof. First, it is proven that $\mathbf{p}_i(A \cap E_i) \subseteq \mathbf{p}_iA$. This follows immediately from $A \cap E_i \subseteq A$.

Second, it is proven that $\mathbf{p}_iA \subseteq \mathbf{p}_i(A \cap E_i)$. Assume that e is an event in $\mathbf{p}_iA \backslash \mathbf{p}_i(A \cap E_i)$. Note that $e \in \mathbf{p}_iA$ implies that $e \in E_i$. So $e \notin \mathbf{p}_i(A \cap E_i)$ implies that $e \notin A$. It follows from $i \in lA$, that $A \cap E_i$ is not empty. Given that $e \notin \mathbf{p}_i(A \cap E_i)$, it follows that there is an $a_i \in A \cap E_i$, such that $a_i \prec e$. Since $e \in \mathbf{p}_iA$, there also exists an event $a \in A$ such that $e \preceq a$. However, this contradicts the convexity of A. From this contradiction it follows that $\mathbf{p}_iA \subseteq \mathbf{p}_i(A \cap E_i)$.

Combining these two results yields $\mathbf{p}_iA = \mathbf{p}_i(A \cap E_i)$. □

The duality of $\preceq$ and $\succeq$ gives the following property.

Property 5.5. *Let A be a convex abstract event. For any $i \in lA$, $\mathbf{f}_iA = \mathbf{f}_i(A \cap E_i)$.*

Let A be a convex abstract event and let i be a process in lA. Property 5.5 can be used to derive an expression for $T^R.A.i$ in terms of vector time. This means that the first test in Theorem 4.30 can be expressed in terms of vector time, yielding a precedence test for the subclass of convex abstract events.

$T^R.A.i$
$= \quad \{$ Definition 4.28 (Reversed timestamp T^R); Property 4.27; Corollary 4.26 $\}$
$T^R.\mathbf{f}_i A.i$
$= \quad \{$ Property 5.5 $\}$
$T^R.\mathbf{f}_i(A \cap E_i).i$
$= \quad \{$ Definition 4.24 (Causal future) $\}$
$T^R.(\cup\, a: \ a \in A \cap E_i : \ \mathbf{f}_i a).i$
$= \quad \{$ (The dual of) Theorem 2.17 $\}$
$(\mathrm{SUP}\ a: \ a \in A \cap E_i : \ T^R.\mathbf{f}_i a).i$
$= \quad \{$ SUP is the componentwise maximum $\}$
$(\mathrm{MAX}\ a: \ a \in A \cap E_i : \ T^R.\mathbf{f}_i a.i)$
$= \quad \{$ Corollary 4.18 $\}$
$(\mathrm{MAX}\ a: \ a \in A \cap E_i : \ \mathbf{E}.i - T.\mathbf{p}_i a.i + 1)$
$= \quad \{$ Corollaries 2.13 and 2.14; Domain is not empty $\}$
$\mathbf{E}.i - (\mathrm{MIN}\ a: \ a \in A \cap E_i : \ T.a.i) + 1$

It follows that for any $i \in lA$, $\mathbf{E}.i - T^R.A.i = (\mathrm{MIN}\ a: \ a \in A \cap E_i : \ T.a.i) - 1$.

This derivation suggests replacing the reversed timestamp for convex abstract events. Two definitions of a new timestamp are given and shown equivalent. The first definition follows from the derivation above and is in terms of primitive timestamps. The second, recursive definition shows that the new timestamp can be calculated from the timestamps of the constituents of an abstract event.

Definition 5.6. (Second timestamp for convex abstract events) The function $T^{CR} : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows:

$$T^{CR}.e.i = \begin{cases} T.e.i - 1, & \text{for } e \in E_i \\ \mathbf{E}.i, & \text{otherwise} \end{cases}$$

for any $e \in E$.

$$T^{CR}.A.i = \begin{cases} (\mathrm{MIN}\ a: \ a \in A \cap E_i : \ T.a.i) - 1, & \text{for } i \in lA \\ \mathbf{E}.i, & \text{otherwise} \end{cases}$$

for any $A \subseteq E$.

For reasons of mathematical convenience that will become clear in a moment, any component i corresponding to a process outside the location set of an event is defined to be $\mathbf{E}.i$.

Corollary 5.7. *For any convex abstract event A and process $i \in lA$, $\mathbf{E}.i - T^R.A.i = T^{CR}.A.i$.*

This corollary can be used to adapt Theorem 4.30 which yields the following precedence test for convex abstract events.

Theorem 5.8. (Precedence test for convex abstract events) *For any convex abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, i: \ i \in lA : \ T^{CR}.A.i < T.B.i)$.*

Note that the components of the timestamp that correspond to processes outside the location set are not necessary to determine precedence between convex abstract events.

As mentioned, the precedence test in Theorem 5.8 can also be used for arbitrary abstract events provided that every abstract event is assigned the two timestamps defined by T and T^{CR}. The latter must be calculated using the convex closure of abstract events.

Theorem 5.9. (Precedence test for arbitrary abstract events) *For any abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, i:\ i \in \boldsymbol{l}(\mathbf{c}A):\ T^{CR}.\mathbf{c}A.i < T.B.i)$.*

In order to show that the new timestamp T^{CR} can be computed from the timestamps in the previous level of the abstraction hierarchy, the following recursively defined timestamp is shown to be equivalent to T^{CR}. The binary operator **co** is used to denote constituents of an abstract event.

Definition 5.10. (Recursive definition of T^{CR}) The function $T^{CRR} : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows. For any $e \in E$, $T^{CRR}.e = T^{CR}.e$. For any $A \subseteq E$, $T^{CRR}.A = (\text{INF } A_0 :\ A_0\, \mathbf{co}\, A :\ T^{CRR}.A_0)$.

Property 5.11. $T^{CRR} = T^{CR}$.

Proof. The proof is by induction on the level of abstraction.
Basis: For any primitive event $e \in E$, $T^{CRR}.e$ is equal to $T^{CR}.e$ by definition.
Induction Hypothesis: For any abstract event A in abstraction level at most n, $T^{CRR}.A = T^{CR}.A$.
Inductive Step: Assume A is an abstract event in abstraction level $n+1$. To avoid case analysis in the derivation below, it is assumed that n is at least one. For any process $i \in \mathcal{P}$,

$$
\begin{array}{ll}
& T^{CRR}.A.i \\
= & \quad \{ \text{ Definition 5.10 } \} \\
& (\text{MIN } A_0 :\ A_0\, \mathbf{co}\, A :\ T^{CRR}.A_0.i) \\
= & \quad \{ \text{ Induction Hypothesis } \} \\
& (\text{MIN } A_0 :\ A_0\, \mathbf{co}\, A :\ T^{CR}.A_0.i) \\
= & \quad \{ \text{ Definition 5.6; } n \geq 1 \ \} \\
& (\text{MIN } A_0 :\ A_0\, \mathbf{co}\, A \ \wedge\ i \in \boldsymbol{l}A_0 :\ (\text{MIN } a :\ a \in A_0 \cap E_i :\ T.a.i) - 1) \text{ min} \\
& (\text{MIN } A_0 :\ A_0\, \mathbf{co}\, A \ \wedge\ i \notin \boldsymbol{l}A_0 :\ \mathbf{E}.i) \\
= & \quad \{ \ i \in \boldsymbol{l}A \ \Leftrightarrow\ (\exists\, A_0 :\ A_0\, \mathbf{co}\, A :\ i \in \boldsymbol{l}A_0);\ \text{MIN is associative} \ \} \\
& \begin{cases} (\text{MIN } a :\ a \in A \cap E_i :\ T.a.i) - 1, & i \in \boldsymbol{l}A \\ \mathbf{E}.i, & i \notin \boldsymbol{l}A \end{cases} \\
= & \quad \{ \text{ Definition 5.6 } \} \\
& T^{CR}.A.i
\end{array}
$$

It is easy to verify that the same result is obtained for n equal to zero. □

The timestamp defined by T^{CR} (or T^{CRR}) marks the beginning of abstract events in *vector time*. It replaces the timestamp defined by T^R that marks the beginning in *reversed vector time*. The relation between T^R and T^{CR} is shown in Figure 7.

Two remarks are in order. First, Theorems 5.8 and 5.9 have duals in terms of reversed vector time. However, they does not appear to be of any practical use. Second, timestamp T^{CR} can also be used to obtain a precedence test for primitive events. However, this test is the same as in Theorem 2.11.

Compared to the precedence tests derived in the previous section, the precedence tests in Theorems 5.8 and 5.9 have one big advantage: they are independent of reversed vector time. However, it is still necessary to timestamp every abstract event twice. An interesting question is whether it is possible to determine precedence using only a single timestamp. The answer is yes provided that

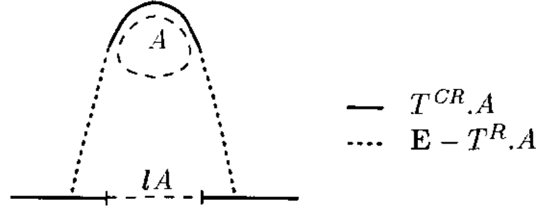

Figure 7: The relation between T^R and T^{CR}.

abstract events are convex and mutually *disjoint*. Depending on the application, this might be a reasonable assumption.

A useful single timestamp for convex abstract events seems to be a timestamp that integrates the times for the beginning and end of events.

Definition 5.12. (A single timestamp for convex abstract events) The function T^{C0} : $E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows. For any $\alpha \in E \cup 2^E$, $T^{C0}.\alpha = T.\alpha \inf T^{CR}.\alpha$.

For processes inside the location set of a convex abstract event, this timestamp represents the beginning of the event. For processes outside the location set, it represents the end. This is formalized in the following two corollaries (see also Figure 8). Note that the corollaries are true for arbitrary primitive or abstract events.

Corollary 5.13. *For any primitive or abstract event $\alpha \in E \cup 2^E$ and process $i \in l\alpha$, $T^{C0}.\alpha.i = T^{CR}.\alpha.i$.*

Corollary 5.14. *For any primitive or abstract event $\alpha \in E \cup 2^E$ and process $i \notin l\alpha$, $T^{C0}.\alpha.i = T.\alpha.i$.*

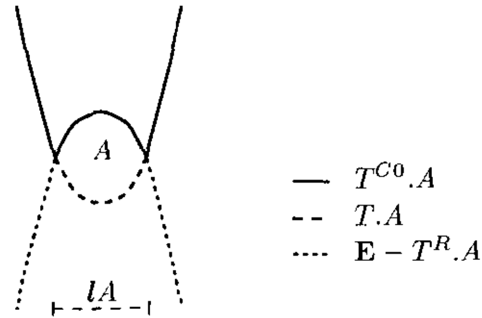

Figure 8: The timestamp T^{C0}.

Assuming that abstract events are convex and mutually disjoint, timestamp function T^{C0} characterizes causality.

Theorem 5.15. (Precedence test for disjoint, convex abstract events) *For any disjoint, convex abstract events A and B, $A \preceq B \Leftrightarrow (\exists\, i :\ i \in lA :\ T^{C0}.A.i < T^{C0}.B.i)$.*

Proof. For convex abstract events, Theorem 5.8 gives $A \preceq B \Leftrightarrow (\exists\, i : i \in \mathbf{l}A : T^{CR}.A.i < T.B.i)$. Corollary 5.13 states that for any $i \in \mathbf{l}A$, $T^{C0}.A.i = T^{CR}.A.i$. Furthermore, Corollary 5.14 yields that only for components i in both $\mathbf{l}A$ and $\mathbf{l}B$, $T^{C0}.B.i \neq T.B.i$. Since $T^{C0}.B.i \leq T.B.i$, it is sufficient to prove the following implication.

> Assume $A \preceq B$. If there is a process $i \in \mathcal{P}$ and a primitive event $a \in A \cap E_i$ such that $i \in \mathbf{l}B$ and $a \in \mathbf{p}_i B$, then $T^{C0}.A.i < T^{C0}.B.i$.

Informally, if the ordering between A and B is caused by a primitive event in process i that is an element of both location sets, then the corresponding component of their timestamps must reflect this.

Assume that $A \preceq B$ and that i and a exist as above. Because A and B are disjoint and convex, $\mathbf{f}_i a \supset \mathbf{f}_i B$. Definition 4.24 (Causal future of an abstract event in a process) gives $\mathbf{f}_i A \supset \mathbf{f}_i B$, which yields:

$\mathbf{f}_i A \supset \mathbf{f}_i B$
$\Leftrightarrow$ { (The dual of) Theorem 2.17 }
$T^R.\mathbf{f}_i A > T^R.\mathbf{f}_i B$
$\Leftrightarrow$ { Corollary 4.26 }
$T^R.\mathbf{f}A.i > T^R.\mathbf{f}B.i$
$\Leftrightarrow$ { Property 4.27; Definition 4.28 (Reversed timestamp T^R) }
$T^R.A.i > T^R.B.i$
$\Leftrightarrow$ { Algebra }
$\mathbf{E}.i - T^R.A.i < \mathbf{E}.i - T^R.B.i$
$\Leftrightarrow$ { Corollary 5.7 }
$T^{CR}.A.i < T^{CR}.B.i$
$\Leftrightarrow$ { $i \in \mathbf{l}A \cap \mathbf{l}B$; Corollary 5.13 }
$T^{C0}.A.i < T^{C0}.B.i$

□

Unfortunately, the timestamp in Definition 5.12 and the precedence test in Theorem 5.15 do not appear to be useful for arbitrary abstract events: in most cases, it cannot be guaranteed that the convex closures of all abstract events are mutually disjoint.

It remains to be shown that the new timestamp is useful in a hierarchy of abstract descriptions. First, it is shown that the timestamp can be expressed in terms of primitive timestamps. Second, we show that it can be defined recursively and, therefore, calculated from the timestamps of the constituents.

Definition 5.16. **(T^{C0} in terms of primitive timestamps)** The function $T^{C1} : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows:

$$T^{C1}.e.i = \begin{cases} T.e.i - 1, & \text{for } e \in E_i \\ T.e.i, & \text{otherwise} \end{cases}$$

for any $e \in E$.

$$T^{C1}.A.i = \begin{cases} (\text{MIN}\ a :\ a \in A \cap E_i :\ T.a.i) - 1, & \text{for } i \in \mathbf{l}A \\ (\text{MAX}\ a :\ a \in A :\ T.a.i), & \text{otherwise} \end{cases}$$

for any $A \subseteq E$.

Definition 5.17. (Recursive definition of T^{C0}) The function $T^{C2} : E \cup 2^E \longrightarrow \mathbb{N}^N$ defines a timestamp for primitive and abstract events as follows:

$T^{C2}.e = T^{C1}.e,$

for any $e \in E$.

$$T^{C2}.A.i = \begin{cases} (\text{MIN } A_0 : A_0 \,\mathbf{co}\, A \wedge i \in \boldsymbol{l}A_0 : T^{C2}.A_0.i), & \text{for } i \in \boldsymbol{l}A \\ (\text{MAX } A_0 : A_0 \,\mathbf{co}\, A : T^{C2}.A_0.i), & \text{otherwise} \end{cases}$$

for any $A \subseteq E$.

Property 5.18. $T^{C0} = T^{C1} = T^{C2}$.

Proof. First, we show that $T^{C0} = T^{C1}$. The equivalence is only shown for abstract events. It is left to the reader to verify the property for primitive events. For any abstract event A and process $i \in \mathcal{P}$,

$$\begin{array}{ll} & T^{C0}.A.i \\ = & \quad \{ \text{ Definition 5.12 } \} \\ & (T.A \text{ inf } T^{CR}.A).i \\ = & \quad \{ \text{ Definition 4.8 } \} \\ & (\text{MAX } a : a \in A : T.a.i) \text{ min } T^{CR}.A.i \end{array}$$

Two cases can be distinguished.

Assume $i \in \boldsymbol{l}A$.

$$\begin{array}{ll} & (\text{MAX } a : a \in A : T.a.i) \text{ min } T^{CR}.A.i \\ = & \quad \{ \text{ Definition 5.6 } \} \\ & (\text{MAX } a : a \in A : T.a.i) \text{ min} \\ & ((\text{MIN } a : a \in A \cap E_i : T.a.i) - 1) \\ = & \quad \{ \text{ Domains not empty } \} \\ & (\text{MIN } a : a \in A \cap E_i : T.a.i) - 1 \\ = & \quad \{ \text{ Definition 5.16 } \} \\ & T^{C1}.A.i \end{array}$$

Assume $i \notin \boldsymbol{l}A$.

$$\begin{array}{ll} & (\text{MAX } a : a \in A : T.a.i) \text{ min } T^{CR}.A.i \\ = & \quad \{ \text{ Definition 5.6 } \} \\ & (\text{MAX } a : a \in A : T.a.i) \text{ min } \mathbf{E}.i \\ = & \quad \{ \text{ Domain not empty } \} \\ & (\text{MAX } a : a \in A : T.a.i) \\ = & \quad \{ \text{ Definition 5.16 } \} \\ & T^{C1}.A.i \end{array}$$

Second, it is shown that $T^{C2} = T^{C1}$. The proof is by induction.

Basis: For any primitive event $e \in E$, $T^{C2}.e$ is equal to $T^{C1}.e$ by definition.

Induction Hypothesis: For any abstract event A in abstraction level at most n, $T^{C2}.A = T^{C1}.A$.

Inductive Step: Assume A is an abstract event in abstraction level $n+1$. To avoid case analysis in the derivations below, it is assumed that n is at least one. Distinguish two cases.

For any $i \in \boldsymbol{l}A$,

$$\begin{array}{ll} & T^{C2}.A.i \\ = & \quad \{ \text{ Definition 5.17 } \} \\ & (\text{MIN } A_0 : A_0 \,\mathbf{co}\, A \wedge i \in \boldsymbol{l}A_0 : T^{C2}.A_0.i) \\ = & \quad \{ \text{ Induction Hypothesis } \} \\ & (\text{MIN } A_0 : A_0 \,\mathbf{co}\, A \wedge i \in \boldsymbol{l}A_0 : T^{C1}.A_0.i) \\ = & \quad \{ \text{ Definition 5.16; } n \geq 1 \} \end{array}$$

For any $i \notin \boldsymbol{l}A$,

$$\begin{array}{ll} & T^{C2}.A.i \\ = & \quad \{ \text{ Definition 5.17 } \} \\ & (\text{MAX } A_0 : A_0 \,\mathbf{co}\, A : T^{C2}.A_0.i) \\ = & \quad \{ \text{ Induction Hypothesis } \} \\ & (\text{MAX } A_0 : A_0 \,\mathbf{co}\, A : T^{C1}.A_0.i) \\ = & \quad \{ (\forall A_0 : A_0 \,\mathbf{co}\, A : i \notin \boldsymbol{l}A_0); \\ & \quad\;\; \text{Definition 5.16; } n \geq 1 \} \end{array}$$

$$
\begin{array}{ll}
& (\text{MIN } A_0 : A_0 \,\mathbf{co}\, A \;\wedge\; i \in lA_0 : \\
& \quad (\text{MIN } a : a \in A_0 \cap E_i : T.a.i) - 1) \\
= & \quad \{ \text{ Domains not empty; associativity } \} \\
& (\text{MIN } a : a \in A \cap E_i : T.a.i) - 1 \\
= & \quad \{ \text{ Definition 5.16 } \} \\
& T^{C1}.A.i
\end{array}
\qquad
\begin{array}{ll}
& (\text{MAX } A_0 : A_0 \,\mathbf{co}\, A : \\
& \quad (\text{MAX } a : a \in A_0 : T.a.i)) \\
= & \quad \{ \text{ Domains not empty; associativity } \} \\
& (\text{MAX } a : a \in A : T.a.i) \\
= & \quad \{ \text{ Definition 5.16 } \} \\
& T^{C1}.A.i
\end{array}
$$

It is left to the reader to verify the case n equal to zero. □

6 Conclusions

In this paper, we have investigated causality among abstract events. In particular, causality was formalized by defining a precedence relation on abstract events, and characterizations of the precedence relation using vector time were given.

Several tests to determine precedence between abstract events were derived. Each test was evaluated against two criteria: efficiency and applicability in a hierarchical abstraction facility.

For arbitrary abstract events, two alternatives were given that satisfy the criteria. The first alternative is based on the notion of reversed vector time. Every abstract event is assigned two timestamps, one in ordinary vector time marking the end of the abstract event and one in reversed vector time marking the beginning. This leads to precedence tests that satisfy the two criteria. A disadvantage is that reversed vector time is computationally expensive.

The second alternative is based on the notion of convexity. The structure of abstract events was restricted to the class of convex abstract events. For this class, the reversed vector timestamp can be replaced by a timestamp in terms of ordinary vector time. Although the class of convex events is itself meaningful and practically useful, this new timestamp can also be used for arbitrary abstract events: the convex closure of abstract events is used to calculate the correct timestamp. An advantage of this alternative is that reversed timestamps are no longer needed. However, every abstract event is still timestamped twice. A disadvantage is that on the fly calculations of the convex closure might be too expensive for some applications.

As a final result, for mutually disjoint convex abstract events, a single timestamp is sufficient. This timestamp integrates the two timestamps marking the beginning and end of an abstract event. Unfortunately, this timestamp does not appear to be useful for arbitrary abstract events.

References

1. A.A. Basten. Event Abstraction in Modeling Distributed Computations. In K. Ecker, editor, *Proceedings of the 11th. Workshop on Parallel Processing*, Lessach, Austria, September 1993. To appear as Informatik-Bericht, Technische Universität Clausthal, Germany, 20 pp. in ms.

2. A.A. Basten. *Hierarchical Event-Based Behavioral Abstraction in Interactive Distributed Debugging: A Theoretical Approach.* Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, The Netherlands, August 1993.

3. E. Best and B. Randell. A Formal Model of Atomicity in Asynchronous Systems. *Acta Informatica*, 16:93–124, 1981.

4. K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. Technical Report 91-1192, Computer Science Department, Cornell University, Ithaca, New York, USA, February 1991.

5. J.P. Black, M.H. Coffin, D.J. Taylor, T. Kunz, and A.A. Basten. Linking Specification, Abstraction, and Debugging. CCNG Technical Report E–232, Computer Communications and Networks Group, University of Waterloo, Ontario, Canada, November 1993. Submitted 1993.11.04 to *IBM Systems Journal*, 26pp in ms.

6. B. Charron-Bost. Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 45–56. Springer Verlag, Berlin, Germany, 1989. Proceedings of the 3rd International Workshop WDAG '89, Nice, France, September 1989.

7. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and Asynchronous Communication in Distributed Computations. Technical Report LITP 91.55, Institut Blaise Pascal, Université Paris 7, Paris, France, September 1991.

8. W.-H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs.* Ph.D. Thesis, CCNG Technical Report T-189, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989.

9. 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, California, US, USA, May 1991. The proceedings appeared also as *ACM SIGPLAN Notices*, 26(12), December 1991.

10. C.J. Fidge. Timestamps in Message Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 55–66, Brisbane, Australia, February 1988.

11. C.J. Fidge. *Dynamic Analysis of Event Orderings in Message-Passing Systems.* PhD thesis, Australian National University, Department of Computer Science, Canberra, Australia, 1989.

12. C.J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

13. 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, USA, January 1988.

14. T. Kunz. Event Abstraction: Some Definitions and Theorems. Technical Report TI-1/93, Technische Hochschule Darmstadt, Fachbereich Informatik, Darmstadt, Germany, February 1993.

15. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

16. F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V., Amsterdam, North-Holland, The Netherlands, 1989. Proceedings of the International Workshop held in Gers, France, October, 1988.

17. 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 4th, 1991, Rennes, France.

18. J.A. Summers. *Precedence-Preserving Abstraction for Distributed Debugging.* Master's thesis, University of Waterloo, Dept. of Computer Science, Waterloo, Ontario, Canada, 1992.

19. D.J. Taylor. A Prototype Debugger for Hermes. In *Proceedings of the 1992 CAS Conference, Volume I*, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.

20. G. Winskel. An Introduction to Event Structures. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer Verlag, Berlin, Germany, 1989. Proceedings of a workshop held in Noordwijkerhout, the Netherlands, May/June 1988.

Computing Science Notes — **Department of Mathematics and Computing Science, Eindhoven University of Technology**

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for $F\omega$, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Bacten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.

93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.