Distributed Computing (1996) 2:173–191

Synchronous, asynchronous, and causally ordered communication*

Bernadette Charron–Bost[1], Friedemann Mattern[2], Gerard Tel[3]

[1] Laboratoire d'Informatique, Ecole Politechnique, F-91128 Palaiseau Cedex, France
[2] Department of Computer Science, Technical University of Darmstadt, Alexanderstrasse 10, D-64283 Darmstadt, Germany
[3] Department of Computer Science, University of Utrecht, P.O. Box 80.089m 3508 TB Utrecht, The Netherlands

Summary. This article studies characteristic properties of synchronous and asynchronous message communications in distributed systems. Based on the causality relation between events in computations with asynchronous communications, we characterize computations which are realizable with synchronous communications, which respect causal order, or where messages between two processes are always received in the order sent. It is shown that the corresponding computation classes form a strict hierarchy. Fruthermore, an axiomatic definition of distributed computations with synchronous communications is given, and it is shown that several informal characterizations of such computations are equivalent when they are formalized appropriately. As an application, we use our results to show that the distributed termination detection algorithm by Dijkstra et al. is correct under a weaker synchrony assumption than orginally stated.

Key words: Distributed computation – Message passing – Synchronous communication – Asynchronous communication – Causal order – Distributed termination detection – Distributed systems – Causality relation – Communication

Correspondence to: F. Mattern

* The work of Gerard Tel was supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM)

Distrib Comput (1996) 9: 173–191

DISTRIBUTED COMPUTING

Synchronous, asynchronous, and causally ordered communication*

Bernadette Charron-Bost[1], **Friedemann Mattern**[2], **Gerard Tel**[3]

[1] Laboratoire d'Informatique, Ecole Polytechnique, F-91128 Palaiseau Cedex, France
[2] Department of Computer Science, Technical University of Darmstadt, Alexanderstrasse 10, D-64283 Darmstadt, Germany
[3] Department of Computer Science, University of Utrecht, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Received: November 1992 / Accepted: July 1995

Summary. This article studies characteristic properties of synchronous and asynchronous message communications in distributed systems. Based on the causality relation between events in computations with asynchronous communications, we characterize computations which are realizable with synchronous communications, which respect causal order, or where messages between two processes are always received in the order sent. It is shown that the corresponding computation classes form a strict hierarchy. Furthermore, an axiomatic definition of distributed computations with synchronous communications is given, and it is shown that several informal characterizations of such computations are equivalent when they are formalized appropriately. As an application, we use our results to show that the distributed termination detection algorithm by Dijkstra et al. is correct under a weaker synchrony assumption than originally stated.

Key words: Distributed computation – Message passing – Synchronous communication – Asynchronous communication – Causal order – Distributed termination detection – Distributed system – Causality relation – Communication

1 Introduction

Messages, their transmission modes, and the semantics of communication play an important role in distributed systems since messages are the only means by which processes can exchange data and can synchronize their actions. *A priori*, communication in distributed systems is not reliable – messages can be lost because of communication failures, duplicated because of retransmissions, or their contents can be garbled or destroyed. Sophisticated techniques and protocols have been devised which cope with these problems and guarantee reliable message delivery to the application. However, even if one supposes that message communication is reliable (as will be done in this article), distributed systems may behave in different ways. For example, messages may be transmitted synchronously or asynchronously, and messages might be received in the order in which they were sent or received out of sequence.

In order to help the programmer to implement efficient and well-structured distributed programs and algorithms, distributed operating systems and programming languages provide various communication mechanisms. Examples are selective receive statements, which block the receiver until a suitable message is available on one of several ports or channels, or remote procedure call abstractions, which block the sender until a return message is available. Non-blocking send instructions, which do not force the sender to wait until the message is delivered at its destination, are also very common. Usually, non-blocking communication mechanisms are called "asynchronous", whereas blocking communication mechanisms are called "synchronous" [41].[1] We will call distributed computations that make exclusive use of asynchronous send instructions *A-computations*, and computations where communication is done solely in synchronous mode *S-computations*. The distinction between synchronous and asynchronous refers only to the communication between processes; the underlying system model we consider here is in all cases asynchronous in the sense that we do not assume the existence of synchronized clocks or fixed communication delays.

It is widely accepted that neither of the two communication modes "synchronous" and "asynchronous" is generally superior to the other. While asynchronous communication is less prone to deadlocks and often allows a higher degree of parallelism (since the sender can proceed while the message is still being delivered), its implementation requires complex buffer management and flow control

Correspondence to: F. Mattern

* The work of Gerard Tel was supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM)

[1] Sometimes, however, a difference between the two notions "blocking" and "synchronous" (or "non-blocking" and "asynchronous", resp.) is made in the literature. This is the case, for example, for the MPI message passing interface which is becoming a de facto standard for message-based communication [24]. Here, a *blocking* send primitive does not return until the message has been copied out of the sender's buffer, whereas a *non-blocking* send can return immediately [16]. Since we abstract from implementation aspects such as message buffering, we are not concerned with this issue here.

mechanisms. Furthermore, algorithms making use of asynchronous communications are often more difficult to develop and to verify than algorithms working in a synchronous environment. It is a well-known fact, however, that it is possible to simulate one mode with the other. In synchronous mode, explicit buffers or intervening buffer processes can be used to decouple the sender of a message from the receiver, thus simulating asynchronous mode.[2] In asynchronous mode, synchronous mode can be simulated by waiting for an explicit acknowledgement immediately after asynchronously sending a message.

The control of A-computations usually requires more sophisticated algorithms than the control of S-computations. This is due to their higher degree of parallelism and non-determinism, and due to messages that can be in transit in A-computations. Examples of control problems for distributed computations include achieving mutual exclusion, termination or deadlock detection, computing consistent snapshots, and collecting garbage objects. Two naturally arising questions of practical importance are:

1. Are control algorithms that are correct for A-computations in general also correct for the control of S-computations?
2. And, conversely, is it possible to adapt distributed control algorithms for S-computations to A-computations?

Informally, S-computations are often regarded as a special case of A-computations, namely computations where the communication channels always appear to be "empty". Thus, one would expect that a distributed algorithm designed for the asynchronous case remains indeed correct when executed on a synchronous system or applied to an underlying S-computation. Formally, an assertional proof of the algorithm (showing an invariant implying its correctness) remains valid if the proof rules for A-computations are replaced by proof rules for S-computations (see Schlichting and Schneider [38]). However, one should be aware of the fact that an algorithm working correctly in the asynchronous case might *deadlock* in the synchronous case. We will indeed show that not all A-computations are realizable under synchronous communication.

Clearly, the safety properties of a synchronous algorithm are usually not preserved when the algorithm is executed in an asynchronous environment (see Gribomont [23]). A good example is the well-known distributed termination detection algorithm of Dijkstra, Feijen, and van Gasteren [19]. This algorithm is safe if message communication of the underlying computation is synchronous,[3] but fails if it is asynchronous because it does not consider messages in transit. Hence, disregarding the deadlock problem, it seems that A-computations are more general than S-computations in the sense that a distributed algorithm designed for an asynchronous environment will work in a synchronous environment but not necessarily vice versa.

The preceding observations show that for the theory of distributed algorithms a precise characterization of synchronous and asynchronous communication modes and an analysis of their relations is of great interest. This article should contribute to the understanding of those fundamental concepts and relations. In particular, we will formally define the notion of a distributed computation (Sect. 3.1) and demonstrate that the hierarchy of S-computations and A-computations indicated by the preceding remarks can be refined in a sensible way. Besides FIFO and non-FIFO message passing disciplines, we will consider in particular so-called *causally ordered* computations (Sect. 3.2). These computations respect the causality relation between send events and receive events in a similar way than do S-computations but they can be implemented without blocking. It will be shown that they lie between S-computations and FIFO-communication based computations in the hierarchy of distributed computation classes (Sect. 3.3). We will give different characterizations of causally ordered computations and prove their equivalence (Sect. 3.4). However, it is equally interesting to characterize S-computations accordingly (Sect. 4), to formally define the semantics of synchronous communications (Sect. 5), and to analyze the conditions under which A-computations can be realized in synchronous mode.

Parts of this article consist of a structured synthesis of known (but sometimes unproven) results, put into a unifying formal framework. Only with such a framework it is possible to axiomatically characterize and formally compare various distributed computation and communication models. Even then, however, some proofs remain non-trivial and require a certain amount of technical subtleties. They show, on the other hand, that intuitive arguments (such as "message arrows in space-time diagrams of S-computations are vertical") have a formal justification.

Besides this, we also present several original results. Among other things we prove that a simpler and apparently stronger "message order" property is equivalent to the classical causal order property (Sect. 3.4), we give a new criterion (the "crown criterion", Sect. 4.3) to decide whether a distributed computation can be realized with synchronous communications, and we show that a well-known termination detection algorithm remains correct under weaker communication requirements than originally stated (Sect. 6). Before we do all this on a more formal basis, we shall first introduce in the next section some fundamental concepts and characterize S-computations and A-computations from an informal point of view.

2 Distributed computations: an informal view

The intention of this section is to provide an intuitive understanding of the main characteristics of synchronous and asynchronous message transmissions. The basic notions and main features discussed in this article are introduced in an informal manner, using diagrams and examples. Formal definitions and proofs as well as

[2] True simulation requires an unbounded amount of buffer storage, however.

[3] The authors use the term "instantaneous" instead of "synchronous".

more precise further characterizations are found in later sections.

2.1 Events and space-time diagrams

A distributed system consists of sequential processes communicating solely by messages. The behavior of each process is controlled by a local algorithm which determines the local sequence of actions and the reaction of the process to incoming messages. The concurrent (and coordinated) execution of all local algorithms forms a *distributed computation.*

In an abstract setting, a distributed computation is determined by the types and the relative order of atomic actions called *events* occurring at the processes. Usually, events are classified into three types according to Lamport [28]: *send* events, *receive* events, and *internal* events. A send event causes a message to be sent, and a receive event causes a message to be received and the local state to be updated by the content of the message. A send event s and a receive event r are said to *correspond* if the same message, sent by s, is received by r. Internal events cause only a change of the local process state – they are not of major concern here since we are mainly interested in the causal relation of events on different processes, which is caused by the exchange of messages.

Lamport [28] pointed out that distributed computations can be visualized using *space-time diagrams,* of which Fig. 1 shows an example.[4] The events of each process are depicted as dots located on a *process line.* On such a line, an event e is drawn to the left of an event e' if and only if e "happens before" e' on the corresponding process. Thus "time" runs from left to right on a process line. Messages are depicted as arrows connecting send events with their corresponding receive events. In an A-computation (where send events are decoupled from their corresponding receive events) it seems obvious that the execution of an event e can *causally affect* another event e' if and only if there is a path in the diagram from e to e'. This path must follow the direction of the arrows and run from left to right on process lines. The existence of such a path induces a causal relationship on the set of events. Since this causality relation is transitive and cycle-free (it is not possible that two events mutually depend on each other), it forms an *irreflexive partial order* which will be denoted as "$\prec$". Appropriately, $e \prec e'$ can be read "e can causally affect e'", "e' potentially depends on e", "e happens before e'", or "e' knows about e".

Message arrows need not be drawn in such a way that receive events are necessarily located to the right of their corresponding send events. Such a drawing, however, is always *possible* by introducing a global time frame and drawing a time axis in parallel to the process lines. The vertical projection of an event on the time axis then represents the time at which the event is executed. Since the sending of a message never occurs later than its receipt, message arrows do not go from right to left in such diagrams.

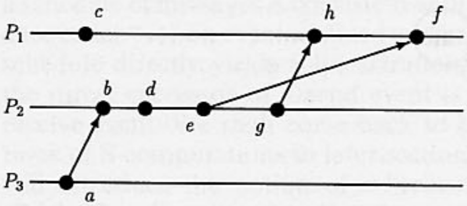

Fig. 1. A space-time diagram of a distributed computation

2.2 Synchronous and causally ordered computations

In general, it is not immediately clear from a space-time diagram whether it depicts a computation with synchronous communications (i.e., an S-computation) or a computation with asynchronous communications (i.e., an A-computation). But what, in fact, is the difference between "synchronous" and "asynchronous" here? Literally, "synchronous" signifies "at the same time" which would mean that in an S-computation a send event should always happen simultaneously with its corresponding receive event. But this is not possible in reality for physical reasons.[5] Therefore, in order to characterize S-computations we confine ourselves to "approximate" instantaneous message transmission in the sense that a computation which exhibits (or which could exhibit) a phenomenon that cannot be observed if message transmission were instantaneous is not "synchronous" and hence not called an S-computation. This rules out the computation of Fig. 1: if message transmission were instantaneous, the first message sent by P_2 (event e) must be received at P_1 (event f) *before* a later message sent by P_2 to P_1 (event g) is received (event h). As a matter of fact, non-FIFO-computations are therefore not S-computations.

Consider now a distributed computation formed by two main processes P_1 and P_2 and a control process P_3, connected as depicted in Fig. 2. Assume that P_1 counts the messages it sends to P_2 and P_2 counts the messages it receives from P_1. If message transmission were instantaneous, the two counters should show the same value at any instant in time. Therefore, if P_3 first asks P_1 for the current value of its send-counter S and then (after receipt of the reply) P_2 for the value of its receive-counter R, one would expect that (if message transmissions between P_1 and P_2 were instantaneous) all messages sent by P_1 when it replies to P_3 have been received at P_2 when P_2 receives P_3's request message. Hence, P_3 should always

[4] *"It is true that there are certain implicit dangers in using such graphical representations, because in every geometrical diagram time appears to be misleadingly spatialized. On the other hand, such diagrams, provided we do not forget their symbolic nature, have a definite advantage…"* This philosophical critique by Milič Čapek [13] on Minkowski's relativistic space-time concept and its simplified models also applies here; as Lamport observes [28], those models are indeed structurally very similar to space-time diagrams as we use them.

[5] We quote again Milič Čapek [12]: *"There is an upper limit to the transmission of any causal action: this is the speed of electromagnetic waves. This is, as Paul Langevin said, the speed limit of causality. Thus there are no instantaneous transmissions in nature … the effect is never contemporaneous with its cause."*

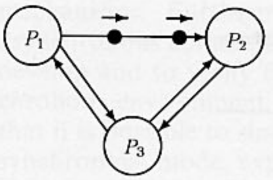

Fig. 2. A controller process P_3 and two main processes P_1 and P_2

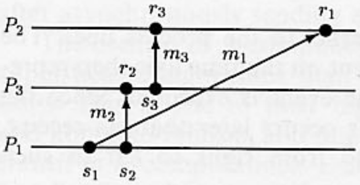

Fig. 3. A computation that is not causally ordered

observe $S \leqq R$ (note, however, that $S = R$ cannot be guaranteed).

If message transmission is not instantaneous, however, P_3 could observe $S > R$ in the above-mentioned scenario. Figure 3 explains this phenomenon. The request message m_3 sent after the receipt of the reply message m_2 (sent after message m_1) arrives at P_2 before m_1. Thus, indirect communication via P_3 was faster than direct communication between P_1 and P_2 – the computation violates the "triangle inequality", so to speak. A distributed computation which does not show the effect that a message is indirectly (or directly) bypassed by a chain of other messages, is called a *causally ordered* computation (or CO-computation for short). This notion will be defined formally in Sect. 3.2.

Obviously, causal order is a generalization of FIFO in the sense that a CO-computation is always a FIFO-computation (but not vice versa as Fig. 3 shows). In CO-computations, messages (i.e., their delivery sequences at the receiving processes) respect the (possibly indirect) causality relation among their send events. This is of importance to many practical applications where causal consistency (in the sense of consistent views or snapshots) must be guaranteed. Renesse [36] presents a collection of such applications and also discusses the use of a generalization of the causal order principle to broadcast communications.

It should be emphasized that in a non-CO-computation there exists a *single* message that is overtaken by another message or a chain of messages (see Fig. 3 where m_1 is overtaken by m_2/m_3). As the S-computation depicted in Fig. 4 shows, it is quite possible that a *chain* of messages between two processes P_1 and P_2 is "overtaken" by a message (or another chain of messages) even if message transmission is instantaneous. The reason is that the intervening process P_3 may buffer the messages for an arbitrary time and therefore essentially simulate asynchronous message transmission between P_1 and P_2.

Returning to the system of Fig. 2, $S > R$ should also be unobservable in an S-computation if P_3 first asks P_1, and then (without waiting for P_1's reply) P_2 to take a local snapshot of their counters. If later these snapshots are compared, the value of the snapshot of S should always be

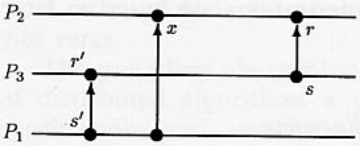

Fig. 4. A synchronous computation with an intervening buffer P_3

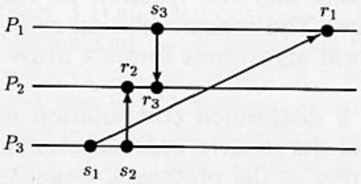

Fig. 5. A causally ordered computation

smaller than or equal to R's snapshot (because P_1 and P_2 receive P_3's request message instantaneously). The space-time diagram of Fig. 5 depicts a scenario which should therefore not happen in an S-computation: the snapshot of S would be 1 (when the "snapshot event" r_1 happens, the send event s_3 of the message from P_1 to P_2 did already happen locally) but R's snapshot would be 0 (the receive event r_3 of the message did not yet happen when the snapshot event r_2 of process P_2 happens). Notice that (if one abstracts from process names) Fig. 5 looks similar to Fig. 3, but because events s_3 and r_3 are swapped, there are no indirect message overtakings. Hence Fig. 5 depicts a CO-computation which is nevertheless able to produce an effect not observable with S-computations. The reason for the non-synchronous behavior is that in some sense (namely in "real time order") r_1 is "later" than r_2 although the order is reversed for the corresponding send events (i.e., s_1 happens earlier than s_2). This "message crossing" is disclosed by the message from P_1 to P_2 which is transmitted between the two receive events r_2, r_1.

Even without having yet precisely defined S-computations, our discussion indicates that the hierarchy *S-computations* $\subset$ *CO-computations* $\subset$ *FIFO-computations* $\subset$ *general A-computations* is strict. We will come back to this further down in Sect. 3.

2.3 Cycles and schedules

In order to characterize S-computations more precisely, we shall now look at a definition given by Bougé [10]: "*A system has synchronous communications if no message of a given type can be sent along a channel before the receiver is ready to receive (that is, in a state where the next action may be a reception of) a message of this type on the channel. For an external observer, the transmission then looks instantaneous and atomic. Sending and receiving a message correspond in fact to the same event.*"

If the computation of Fig. 3 would be executed on a system with synchronous communications as characterized by Bougé, s_1 could not take place, because P_2 is not ready to receive the message. Process P_2 must first execute r_3, but this event depends on s_3, which depends on r_2,

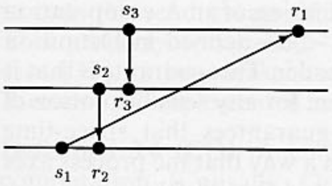

Fig. 6. Another causally ordered computation

which depends on s_2, which in turn depends on s_1. Hence, we have a *deadlock* situation – no message can be scheduled first because m_1 can only be scheduled after m_3, m_3 can only be scheduled after m_2, and m_2 can only be scheduled after m_1. The scheduling of entire message transmissions (rather than individual events) seems to be a natural consequence of the atomicity of a send and its corresponding receipt. A message schedule implies a schedule of events in which the corresponding communication events are executed consecutively, without being separated by other events. The two events thus simulate a *single* communication event,[6] distributed over sender and receiver. This notion will be formalized in Definition 3.6 ("RSC-computations").

The computation of Fig. 5 exhibits the same cyclic dependency on messages as the computation of Fig. 3 (despite the fact that events s_3 and r_3 are swapped) and is therefore also not realizable in synchronous mode. Figure 6 depicts yet another (rather typical) deadlock situation. Here, all processes are blocked in a send instruction since no process is ready to receive a message before it has sent its message. It should be noted, however, that the computations depicted in Fig. 3, 5, and 6 are not deadlocked when communications are asynchronous. For example, breaking the atomicity of m_1 in Fig. 3 yields $s_1, s_2, r_2, s_3, r_3, r_1$ as a possible global schedule of events for the computation.

Interestingly, for all three space-time diagrams just considered, it is possible to construct the same cyclic dependency $r_1 \prec s_2 \prec r_2 \prec s_3 \prec r_3 \prec s_1 \prec r_1$ on *events* by reversing the direction of appropriate message arrows (e.g., m_1 in Fig. 3) and swapping the associated send-receive events. It should be clear that if a cycle can be constructed in this way, there is no first message transmission which can be scheduled as a whole. Hence, space-time diagrams containing such cycles are invalid in the synchronous mode. Conversely, if such a cycle cannot be constructed, it should be possible to find a message schedule. We therefore come to the conjecture that an A-computation is realizable in synchronous mode (i.e., there exists an equivalent S-computation with the same events and messages) if and only if no such cycle can be formed. Notice that a schedule of messages is consistent with the causal precedence order "$\prec$" on events defined earlier. In fact, a message schedule directly yields a linearization of "$\prec$" such that the direct successor of a send event is its corresponding receive event. We shall come back to these characterizations of S-computations in later sections; in particular we will introduce the notion of a "crown" (Definition 4.1) which formalizes the idea of message scheduling cycles.

3 Distributed computations

In this section various classes of distributed computations are formally defined. Furthermore, it is shown that different requirements on the message transmission disciplines with respect to the preservation of the causality relation yield a hierarchy of distributed computation classes. As mentioned before, we consider reliable distributed systems only.

3.1 Computations with asynchronous communications

A distributed system consists of processes $P_1, \ldots, P_n$, communicating only by exchanging messages. We do not make any assumption on the relative speed of processes or on message transmission delays, hence we consider only so-called "asynchronous systems". A computation on such a system basically consists of a tuple of local computations, one for each process, and a causality relation on the set of events, which is induced by the exchange of message. Each process P_i runs a sequential program, whose execution is modeled by a finite sequence C_i of *events*. This sequence is called a *local computation* (of P_i). A computation C of a distributed system is composed of local computations $C_1, \ldots, C_n$ performed by $P_1, \ldots, P_n$, respectively. We write $a \sim b$ to denote that events a and b take place at the same process, i.e., a $\sim b$ if there is an i such that $a \in C_i$ and $b \in C_i$.

For a given computation C, let $\Gamma = \{(s, r) \in C_i \times C_j : s$ corresponds to $r\}$ denote the set of corresponding pairs of communication events. Since we model point-to-point communications, where only a single message is either sent or received in any communication event, we often also say that r corresponds to s if s corresponds to r. We assume that a process does not communicate with itself, i.e., corresponding send and receive events are located on *different* processes.[7] With point-to-point communications, each communication event has at most one corresponding communication event. Since every message which is received must have been sent, we will only consider n-tuples $C = (C_1, \ldots, C_n)$ such that, for every receive event r in C, there exists a send event s such that $(s, r) \in \Gamma$. If, in addition, for each send event there is a corresponding receive event (i.e., send and receive events are in a one-to-one correspondence), then there is no message in transit at the

[6] Baeten and Weijland [3], for example, define a synchronous communication as the result of the simultaneous execution of two corresponding actions, together forming a single communication action. This view is also implied by the semantics of CCS [32] and CSP [25] where the meaning of two matching communication actions (a and $\bar{a}$ in CCS, or $p!e$ and $q?x$ in CSP) is given by a single "atomic" action (the silent action τ in CCS and the assignment statement $x := e$ in CSP).

[7] This does not exclude application of our results to algorithms where processes send messages to themselves. Often, however, one would model such a message by a single internal event or a send event immediately followed by the corresponding receive event thus reflecting the usual implementation shortcut of self-sends.

end of the computation. Such computations are called *complete* computations.

We shall now precisely define the causality relation "$\prec$" of an n-tuple of local computations. As each P_i is a sequential process, the events of this process are totally ordered by their occurrence in the sequence C_i. This order, which we denote by $\prec_i$, implies a causal order on the events of process P_i in the sense that an earlier event may affect a later event on the same process. Clearly, $\prec_i$ is an irreflexive total order (on the events of a single process C_i). Causal dependencies between events at different processes are induced by message exchanges. If $(s, r) \in \Gamma$, then the receive event r causally depends on the corresponding send event s. All causal dependencies between the events of C are induced by transitivity from the relations $\prec_1, \ldots, \prec_n$ and Γ, as formalized in the following definition originally stated by Lamport [28] for his "happened before" relation.

Definition 3.1 (Causality relation). The causality relation $\prec$ in C is the smallest relation that satisfies the following three properties.

AS1 If $a \prec_i b$, then $a \prec b$.

AS2 If $(s, r) \in \Gamma$, then $s \prec r$.

AS3 If $a \prec b$ and $b \prec c$, then $a \prec c$.

The reflexive closure of $\prec$ will be denoted by $\preceq$, i.e., $a \preceq b \Leftrightarrow (a \prec b \vee a = b)$. Two distinct events are said to be *concurrent* if they are not ordered by $\prec$.

When a distributed computation is executed, an idealized global observer (who has an instantaneous global view of the computation) can perceive the events only in an order which is consistent with the causality relation. In other words, if an event a could affect another event b, then a must happen before b in "real time". This explains why the causality relation of a distributed computation must be cycle-free and motivates the following definition.

Definition 3.2 (A-computation). A computation with asynchronous communications (or an A-computation for short) of a distributed system consisting of processes $P_1, \ldots, P_n$ is an n-tuple $C = (C_1, \ldots, C_n)$ of local computations together with a set Γ of corresponding communication events, for which the causality relation $\prec$ is a partial order.

It should be observed that the set Γ is essential for a suitable definition of A-computations. Without it, an A-computation would just be a partially ordered set of events, together with a partitioning into totally ordered subsets C_i. In that case, however, Fig. 3 and a similar but different scenario where message m_1 is removed (and events s_1 and r_1 thus replaced by internal events) would represent the *same* computation. This also shows that space-time diagrams are more than just Hasse diagrams[8] of the causality relation – it is in fact not possible to apply the so-called transitive reduction scheme because this would possibly remove messages and thus change the computation.

The crucial point in the definition of an A-computation is that the transitive relation $\prec$, as defined in Definition 3.1, is required to be a partial order. This guarantees that it is cycle-free, a basic requirement for any sensible notion of (potential) causality. It also guarantees that space-time diagrams can be drawn in such a way that the process axes and all message arrows go from left to right.

The term "asynchronous" in the definition of A-computations is motivated by property AS2. This axiom is responsible for the basic asymmetry between send events and receive events – receive events depend on their corresponding send events, but not vice versa. Hence, a receive event is blocking – it cannot be executed before the corresponding send event is executed. This is not the case for a send event; send events can be executed without waiting for the receiver. We will see in Sect. 5.1 that it is in fact only axiom AS2 which has to be replaced by a symmetric variant to formalize computations with *synchronous* communications (i.e., S-computations) accordingly.

3.2 Order preserving properties and constraints

In practice, distributed computations are often "less asynchronous" than allowed by Definition 3.2. This results from imposing certain constraints on the sending or receipt of messages. For example, it might be the case that a message can only be received if all messages sent earlier along the same communication channel have already been received, or a message might only be sent if the previously sent message has been acknowledged.

Such constraints could arise at different levels: they could result from a particular implementation of a communication protocol (e.g., no buffering of messages), or they could be imposed by the semantics of a distributed programming language in order to simplify correctness proofs of distributed programs. Interestingly, it turns out that the most important and commonly used restrictions can be neatly stated as properties of the causality relation. This will be shown in the following, where we assume that all computations are complete (i.e., for each send event there is a unique corresponding receive event and vice versa – there are no messages in transit at the end of the computation).

In the design of distributed algorithms (e.g., the well-known snapshot algorithm by Chandy and Lamport [14]) it is often assumed that any two messages exchanged between the same two processes are received in the order in which they have been sent. This assumption, referred to as the FIFO (i.e., "first-in, first-out") assumption, is modeled by the following condition.

Definition 3.3 (FIFO). A computation C is called a FIFO-computation if for all (s, r) and $(s', r') \in \Gamma$

$$\left.\begin{array}{l} s \sim s' \\ r \sim r' \\ s \prec s' \end{array}\right\} \Rightarrow r \prec r'$$

The FIFO-property can be strengthened by dropping the $s \sim s'$ condition, thereby requiring that *all* messages sent to the same process are received in an order consistent with the causal order of the corresponding send events. The order in which the messages are received may be arbitrary

[8] For a definition of Hasse diagrams we refer to appropriate textbooks on order theory (e.g., [17]).

in case the send events of the messages are concurrent. This property was called "causal ordering" by Schiper et al. [37]; for broadcast communications it was defined by Birman and implemented in the Isis system [6, 7].

Definition 3.4 (CO). A computation C is called a CO-computation (Causally Ordered) if for all (s, r) and $(s', r') \in \Gamma$

$$\left.\begin{array}{l} r \sim r' \\ s \prec s' \end{array}\right\} \Rightarrow r \prec r'.$$

One would expect that the CO-condition trivially holds for executions where a receive event always happens simultaneously or "synchronously" (i.e., at the same instant) with its corresponding send event. This is indeed the case, computations which allow such (idealistic) executions are called *realizable with synchronous communication* (or RSC for short). Before we can formally define them, the notion of an execution must be formalized.

Since events are abstractions of atomic actions, an execution of a distributed computation can be modeled as a feasible scheduling of the events of the computation, which is a total order that extends the partially ordered set (or *poset*, for short) $(C, \prec)$. Note that this means that instead of executing two causally unrelated events truly concurrently, they can always be executed sequentially in either order. In a natural way, an execution with instantaneous message transmission is modeled by an irreflexive linear extension[9] where a receive event is the direct successor of its corresponding send event, as stated in the following definition.

Definition 3.5 (Non-separated linear extension). A linear extension $(C, <)$ of an A-computation is called non-separated if for each pair $(s, r) \in \Gamma$ the interval $\{x \in C : s < x < r\}$ is empty.

The property of computations where corresponding send and receive events may happen simultaneously in an execution is now formally defined in the following way.

Definition 3.6 (RSC). A computation C is called an RSC-computation (Realizable with Synchronous Communication) if there exists a non-separated linear extension of the poset $(C, \prec)$.

Note that in such a linear extension two corresponding communication events always appear either both later than another event, or both earlier than another event. With respect to $<$, corresponding communication events thus have a common past and a common future – they appear as if they occurred simultaneously.

3.3 The hierarchy of distributed computation classes

After having formally defined the most important computation classes, we now show that there is a strict hierarchy *RSC* $\subset$ *CO* $\subset$ *FIFO* $\subset$ *A-computations*, as already indicated in Sect. 2.2. Indeed, the following theorem is an immediate consequence of the various definitions given in the previous section.

Theorem 3.7 (Hierarchy). *C is RSC $\Rightarrow$ C is CO $\Rightarrow$ C is FIFO.*

Proof. First assume C is RSC. Consider $(s, r), (s', r') \in \Gamma$ for which $s \prec s'$. As C is RSC, there exists a non-separated linear extension $<$ of $\prec$. As $<$ extends, we have $s < r$ and $s < s'$. As $<$ is total and non-separating, $r < s'$ follows (note that if conversely $s' < r$ would hold, then (s, r) would be "separated" by s'). Because $s' \prec r'$, this implies $r < r'$, and as $<$ extends $\prec$ and events of one process are linearly ordered, $r \sim r'$ implies $r \prec r'$. Thus C is CO.

Next assume C is CO. From Definition 3.3 and Definition 3.4 it follows directly that C is FIFO. □

Theorem 3.8 (Strict hierarchy). *There exist A-computations that are not FIFO. There exist FIFO-computations that are not CO. There exist CO-computations that are not RSC.*

Proof. Examples of such computations are shown in Figs. 1, 3, and 5. It is left to the reader to verify that these computations have (or do not have) the indicated properties. □

3.4 Different characterizations of CO-computations

The only communication disciplines which are traditionally considered for the design of distributed algorithms are synchronous communications, FIFO, and asynchronous communications. Since the CO-computations fall between FIFO-computations and computations which are realizable with synchronous communications (i.e., RSC-computations), it is interesting to study the relevance of the CO-property. In Sect. 6, where we shall consider the problem of termination detection as an example of a distributed control problem, we will prove that the CO-property is in fact sufficient for the correctness of a classical termination detection algorithm originally designed for the class of S-computations (which, as will be shown further down, are in some sense equivalent to RSC-computations). This motivates further investigations of the CO-property, of which some alternative characterizations will be discussed in this section.

First, recall that the CO-property (Definition 3.4) was derived from the FIFO-property (Definition 3.3) just by dropping the $s \sim s'$ condition, thus "globalizing" the causality preserving property of FIFO communication channels. One could now try to strengthen the CO-property further by also removing the condition regarding the location of the receive events. In order to obtain a significant class of computations we allow two receive events on different processes to be concurrent if their corresponding send events are causally related. We require, however, that these receive events are not ordered conversely to their corresponding send events. The resulting type of computations will be referred to as "message ordered".

Definition 3.9 (MO). A computation C is called an MO-computation (Message Ordered) if for all (s, r) and $(s', r') \in \Gamma$

$$s \prec s' \Rightarrow \neg (r' \prec r).$$

[9] Recall that a linear extension of a partial order $(C, \prec)$ is a total order $(C, <)$ such that $a \prec b \Rightarrow a < b$.

Note that in Definitions 3.3 (FIFO) and 3.4 (CO) the term $r \prec r'$ on the right-hand side of the implication can be replaced by $\neg(r' \prec r)$ (because by $r \sim r'$ the receive events are linearly ordered) thus emphasizing the formal similarity of those definitions to the MO-definition.

Surprisingly, the class of computations defined by the MO-property is identical to the class of CO-computations, as shown by the following theorem.

Theorem 3.10 (MO=CO). *A computation is CO if and only if it is MO.*

Proof. First assume C is a MO-computation. Consider $(s,r), (s',r') \in \Gamma$ for which $s \prec s'$ and $r \sim r'$. As C is MO, $\neg(r' \prec r)$, but $r \sim r'$ implies $r \prec r' \vee r' \prec r$. Hence $r \prec r'$, which implies that C is CO.

Next we shall show that if there are two messages violating the order required for the MO-property, then such a pair can be found for which the receive events occur on the same process, showing in this manner that the computation is not CO. Assume C is a CO-computation, but C is not MO, i.e., there are two pairs (s, r) and (s', r') of corresponding events such that $s \prec s'$ and $r' \prec r$. It follows from the definition of $\prec$ that there exists a chain $a_0, \ldots, a_k$ of events, such that $r' = a_0$, $r = a_k$, and for each $j < k$, $a_j \prec_i a_{j+1}$ for some i, or $(a_j, a_{j+1}) \in \Gamma$.

1. If no $(a_j, a_{j+1}) \in \Gamma$ occurs in this chain, then $r' \sim r$. As C is a CO-computation, this together with $s \prec s'$ implies $r \prec r'$, which contradicts the assumption.

2. Otherwise, let $(a_i, a_{i+1}) = (\sigma, \rho)$ be the last pair of Γ in this chain. Thus $\rho \sim r$ and $\rho \preceq r$, and from $s \prec s'$, $(s', r') \in \Gamma$, and $r' \prec \sigma$ we obtain $s \prec \sigma$. As C is a CO-computation and $\rho \sim r$, this implies $r \prec \rho$, which contradicts $\rho \preceq r$.

It follows that each CO-computation is also MO. □

The MO-characterization shows directly that in CO-computations a single message (s, r) cannot be overtaken by a chain of other messages, as already stated in Sect. 2.2. The reason is that for any message (s', r') which belongs to such a hypothetical bypass chain, one has necessarily $s \prec \cdots \prec s' \prec r' \prec \cdots \prec r$ which, by transitivity, contradicts the MO-property. Conversely, if the MO-property does not hold, then there exist two messages (s, r) and (s', r') with $s \prec s'$ and $r' \prec r$, and hence $s \prec s' \prec r' \prec r$. This means that (s', r') is part of a chain which overtakes message (s, r). Informally, one could thus say that CO-computations are characterized by the fact that indirect communications always take longer than direct communications. The following corollary summarizes these findings.

Corollary 3.11 (Triangle inequality). *A computation is CO if and only if no message is bypassed by a chain of other messages.*

As an application of this characterization consider a distributed program written in a synchronous message passing language. If the sender should be decoupled from the receiver, one might add to each process an input buffer and an output buffer both realized by a FIFO message queue. A send event now corresponds to the insertion of a message into the output buffer, and a receive event to its removal from the input buffer. Clearly, because of the FIFO property of the buffers and because message transmissions from the sender's output buffer to the receiver's input buffer are synchronous, no (indirect) message overtaking can take place. Hence, although from the application level perspective the computation is no longer guaranteed to be RSC, it still has the CO-property [31].

The definitions given above for FIFO, CO, and MO (Definitions 3.3, 3.4 and 3.9) have a similar structure since they all concern the relative order of receive events corresponding to ordered send events. The definition of RSC (Definition 3.6) has a quite different appearance, as it is related to the existence of linear extensions of $\prec$. We shall next prove a characterization (Proposition 3.13) of CO-computations with a similar flavor, namely based on the partial order $(C, \prec)$ as a whole. An application of this characterization will be a different demonstration that all RSC-computations are CO. We shall also give an interesting graphical interpretation of this characterization for space-time diagrams (Observation 3.14). Subsequently (Corollary 3.15) we shall derive that CO-computations are precisely those computations where the "causal future" of a send event is disjoint from the "causal past" of the corresponding receive event.

The following definition and proposition shows that CO is equivalent to the *empty interval* (EI) property, which states that for no pair $(s, r) \in \Gamma$ there are events "between" s and r.

Definition 3.12 (EI). A computation C is called an EI-computation (Empty Interval) if for each pair $(s, r) \in \Gamma$ the open interval $\langle s, r \rangle = \{x \in C : s \prec x \prec r\}$ is empty.

Proposition 3.13 (EI=CO). *A distributed computation C is CO if and only if it is EI.*

Proof. It is first shown that if C is not CO, then it is not EI. Assume that (s, r) and (s', r') are two pairs in Γ such that $s \prec s'$, $r \sim r'$, and $r' \prec r$. Since $s' \prec r'$, we have $s \prec s' \prec r' \prec r$, hence $\langle s, r \rangle$ is not empty. It follows that EI implies CO.

It is next shown that if C is not EI, then it is not CO. Let (s, r) be a pair of Γ such that $\langle s, r \rangle$ contains an event x. As $\neg(s \sim r)$ and $\sim$ is transitive, $\neg(s \sim x)$ or $\neg(x \sim r)$ holds. In either case there exists a chain $a_0, \ldots, x, \ldots, a_k$ with $s = a_0$, $r = a_k$, $a_j \prec a_{j+1}$ for all j, and at least one pair (a_j, a_{j+1}) is a pair (σ, ρ) of corresponding send and receive events such that $\sigma \neq s$ and $\rho \neq r$. Thus $s \prec \sigma$, while $\rho \prec r$, which shows that C is not MO; by Theorem 3.10 the computation is not CO. It follows that CO implies EI. □

As an example, Fig. 3 depicts a computation which is *not* EI: all four events s_2, r_2, s_3, r_3 are contained in the interval $\langle s_1, r_1 \rangle$. For the computations depicted in Fig. 5 and Fig. 6, however, all intervals formed by corresponding send-receive events are empty. This seems to be obvious for those pairs where the events are connected by vertical message arrows (since causality does not go from right to left, there is no space left for an event between the send event and the receive event), but one also readily checks the condition for $\langle s_1, r_1 \rangle$.

The difference between EI and RSC consists in the underlying order for which the intervals $\langle s, r \rangle$ must be empty. For EI the intervals of $\prec$ itself are empty, while for RSC a linear extension must be found with empty intervals. Clearly, if there exist events s, r, and x such that $s \prec x \prec r$, then for each linear extension $<$ of $\prec$ one has $s < x < r$. Consequently, if there exists a linear extension with empty intervals, then the corresponding intervals of $\prec$ are empty, too. This shows again that every RSC-computation is EI and hence CO, as already stated in Theorem 3.7.

The EI-characterization of CO-computations has a nice graphical interpretation. Note that for an empty interval $\langle s, r\rangle$ of a given computation it is always possible to find a linear extension $<$ of $\prec$ for which the corresponding interval $\{x \in C : s < x < r\}$ is empty. Intuitively, empty $\langle s, r\rangle$-intervals in a linear extension correspond to vertical message arrows in space-time diagrams, because one may move events s and r arbitrarily close together and in fact assign the same real-time instant to them. Conversely, a vertical message arrow from a send event s to a receive event r in a space-time diagram where no message arrow goes "backwards" implies that the corresponding $\langle s, r\rangle$ interval is empty with respect to $<$ and $\prec$. This yields the following characterization which is a consequence of Proposition 3.13.

Observation 3.14 (Vertical message arrow criterion). *A distributed computation C is CO if and only if for each message m there exists a space-time diagram for C such that m can be drawn as a vertical message arrow* (*and no other message arrows do go from right to left*).

An immediate consequence of this criterion is that for each single message of a CO-computation one can assume that this message is transmitted instantaneously.[10] Clearly, such an instantaneous message cannot be bypassed by another message or by a chain of messages, thus showing again the triangle inequality (Corollary 3.11).

An example of a computation for which the criterion of Observation 3.14 does *not* apply is again Fig. 3 – the arrow for message m_1 cannot be made vertical without forcing m_2 or m_3 to move backwards in time. Note also that it is in general not possible to find a *single* linear extension where *all* corresponding intervals are empty. An example is the computation depicted in Fig. 5 – although (for different space-time diagrams) each single message can be drawn vertically, no diagram can be drawn with all messages made vertical simultaneously, a property which is characteristic for RSC-computations as we shall see further down (Observation 4.5).

The following corollary characterizes CO-computations by a so-called *weak common past* or *weak common future* property. It is merely a different formulation of Proposition 3.13 which, however, looks rather similar to Definition 3.9 (MO), in particular if one observes that the implication in Definition 3.9 can be stated equivalently as $r' \prec r \Rightarrow \neg(s \prec s')$.

Corollary 3.15 (Weak common past/future). *A distributed computation C is CO if and only if for each pair* $(s, r) \in \Gamma$ *and for each* $x \in C$

$$x \prec r \Rightarrow \neg(s \prec x) \qquad (WCP)$$

or, equivalently, if and only if

$$s \prec x \Rightarrow \neg(x \prec r). \qquad (WCF)$$

It should be noted that the stronger properties $x \prec r \Rightarrow x \prec s$ and $s \prec x \Rightarrow r \prec x$ do usually not hold, not even for RSC-computations. (An example is the RSC-computation depicted in Fig. 4.) We will see in Sect. 5, however, that an important role in the axiomatization of distributed computations with synchronous communications (i.e., S-computations) is in fact played by such "strong" common past and common future properties, where the whole "causal future" (or "causal past", resp.) of a communication event is included in the "causal future" ("causal past", resp.) of its corresponding event. Obviously, the converse implications $x \prec s \Rightarrow x \prec r$ and $r \prec x \Rightarrow s \prec x$ of the "strong" common past and common future properties are trivially true for every computation because of AS2 ($s \prec r$) and AS3 (transitivity).

4 Characterizations of RSC-computations

Like CO-computations, RSC-computations can be characterized in various ways, and in this section several equivalent characterizations are presented and compared. Insight in properties satisfied by RSC-computations may help to find distributed control algorithms for this class which are simpler and more efficient than algorithms that cannot rely on these properties. Furthermore, efficiently determining whether a computation is RSC is sometimes of practical importance, for example when replaying a distributed computation for debugging purposes. Unfortunately, testing whether a computation is RSC by checking all linear extensions of the underlying partially ordered set according to Definition 3.6 is very expensive and thus not practical.

Therefore, two different characterizations of RSC-computations are given in this section. The first characterization is based on occurrences of a substructure called "crown", and the main result of this section (Theorem 4.4) is that a computation is RSC if and only if it does not contain a crown. We will show that efficiently checking the existence of a crown is possible by considering a simple binary relation on pairs of corresponding communication events (i.e., messages) and checking whether the graph of this relation is cycle-free. The second characterization, based on message scheduling, follows from the first one and has a simple graphical interpretation (Observation 4.5) – a computation is RSC if and only if it has a space-time diagram where all message arrows are vertical.

4.1 Crowns in distributed computations

Crowns capture the idea of cyclic dependencies of messages. We shall motivate their definition with two simple observations.

First, in a computation which is not CO, hence not RSC, there always exist two pairs (s, r) and (s', r') of corresponding send and receive events such that $s \prec s'$ and $r' \prec r$.

[10] This does not necessarily apply to any other message in that specific view, however.

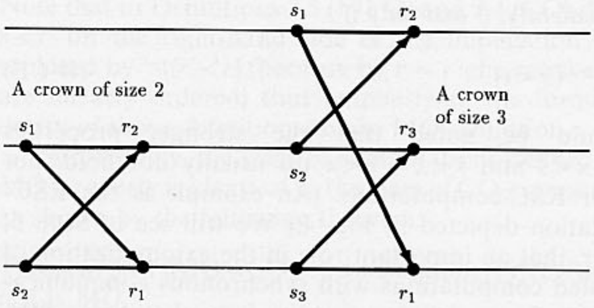

Fig. 7. Crowns of size 2 and 3

As $s' \prec r'$, by transitivity of $\prec$ we get the symmetric property

$$s \prec r' \quad \text{and} \quad s' \prec r.$$

This, however, is the typical "deadlock situation" considered in Sect. 2.3 which disallows the schedule of whole messages.

Second, consider Figs. 5 and 6 which show CO-computations that are not RSC, and Fig. 3 which shows a non-CO-computation. For all these three examples of non-RSC-computations we have

$$s_1 \prec r_2, \quad s_2 \prec r_3, \quad s_3 \prec r_1.$$

If here we "identify" corresponding send and receive events s_i, r_i, which is the intuition behind the idea of scheduling whole messages or considering single communication events in the synchronous case, we get again a cyclic dependency.

Generalizations of these two cases suggest the following definition.

Definition 4.1 (Crown). Let C be a computation. A crown (of size k) in C is a sequence $\langle (s_i, r_i), i \in \{1, \ldots, k\} \rangle$ of pairs of corresponding send and receive events such that

$$s_1 \prec r_2,\ s_2 \prec r_3, \ldots, s_{k-1} \prec r_k,\ s_k \prec r_1.$$

Examples of "canonical" crowns with a characteristic wrapped-around zigzag shape responsible for their name are given in Fig. 7. Figures 3 and 5 show, however, that not all crowns have this typical geometrical shape and that the send events (or the receive events) do not necessarily lie on different processes. Besides the crowns of size 3 indicated above, both computations also have crowns of size 2. (For Fig. 3, $s_1 \prec r_2$ and $s_2 \prec r_1$; and for Fig. 5, $s_1 \prec r_3$ and $s_3 \prec r_1$.)

The first observation at the beginning of this subsection shows that every non-CO-computation necessarily contains a crown of size 2. Notice that the computation of Fig. 6 does not contain a crown of size 2 and is therefore CO. Computations which contain a crown of size 2, however, can be CO as Fig. 5 proves. As indicated above, one can easily verify that every computation that contains a crown is non-RSC. The main result of this section is that, conversely, any computation with no crown is RSC, as stated in the Theorem 4.4 further down. Hence, the existence of a crown fully characterizes non-RSC-computations. The proof of this *crown criterion* relies on several auxiliary results which we shall prove next.

4.2 The reduction lemma

In this section we show that the internal events of a computation can be ignored in the crown criterion (Theorem 4.4). Let C be a computation and define a computation $\tilde{C}$, called the *reduced computation* of C, by removing all internal events of C (i.e., $\tilde{C}$ is the subset of send and receive events). Corresponding pairs in $\tilde{C}$ are the same as in C. Clearly, the causality relation on $\tilde{C}$ is the reduction on $\tilde{C}$ of the causality relation defined on the computation C, and $\tilde{C}$ is a computation according to Definition 3.2.

Lemma 4.2 (Reduction lemma). *The reduced computation $\tilde{C}$ of C is RSC if and only if C is RSC.*

Proof. A linear extension L of C defines a linear extension $\tilde{L}$ of $\tilde{C}$ by removing all internal events from L. If L is a non-separated linear extension of C, then $\tilde{L}$ is a non-separated linear extension of $\tilde{C}$. Thus, if C is RSC, then $\tilde{C}$ is RSC.

Conversely, suppose that $\tilde{C}$ is RSC; let $\tilde{L} = (s_1, r_1, \ldots, s_p, r_p)$ be a non-separated linear extension of $(\tilde{C}, \prec)$. To extend $\tilde{L}$ to a non-separated linear extension of C, we first partition the internal events in C by combining successive internal events of a process which lie between two communication events. Inductively define the sets S_m and R_m for $m \in \{1, \ldots, p\}$ and E_i for $i \in \{1, \ldots, n\}$ by

$$\begin{aligned} S_m = \{x \in C \backslash \tilde{C} : & (x \sim s_m) \text{ and } (x \prec s_m) \\ & \text{and } (x \notin \cup_{1 \leqslant k \leqslant m-1} (S_k \cup R_k))\} \\ R_m = \{x \in C \backslash \tilde{C} : & (x \sim r_m) \text{ and } (x \prec r_m) \\ & \text{and } (x \notin \cup_{1 \leqslant k \leqslant m-1} (S_k \cup R_k))\} \end{aligned}$$

and the set of final local events

$$E_i = C_i \backslash \tilde{C} \backslash (S_1 \cup \cdots \cup S_p) \cup (R_1 \cup \cdots \cup R_p).$$

Informally, the set S_m (R_m, resp.) is formed by the internal events located on the same process as s_m (r_m, resp.), just before s_m (r_m, resp.) and which have not yet been considered in the preceding sets $S_1, \ldots, S_{m-1}$ and $R_1, \ldots, R_{m-1}$. Internal events that occur in C_i after the last send or receive event performed by P_i are combined into E_i.

Each of these sets is totally ordered by the relation $\prec$. The sequence L is obtained from $\tilde{L}$ by insertion of the internal events as follows. Before s_m (for each $m \in \{1, \ldots, p\}$), insert first S_m and then R_m, and after r_p, add $E_1, \ldots, E_n$. We claim that L is a non-separated extension of $(C, \prec)$.

We shall first show that L is an extension of $(C, \prec)$. To this end we must show that the events of each process appear in L in the correct order, and that corresponding send and receive pairs appear in the correct order. Consider events a and b satisfying $a \prec_i b$. We may assume that there is no event between a and b in P_i. The following four cases are exhaustive:

1. If a and b are both a communication event, then a and b appear in L in the same order as in $\tilde{L}$, thus a appears before b.
2. If a and b are both an internal event, they are in the same set (S_m, R_m or E_i) and appear in L in the correct order.

3. If a is an internal event and b is a communication event, then a is in the set S_m or R_m corresponding to b and appears in L before b by the construction of L.

4. If a is a communication event and b is an internal event, then b belongs to S_m, R_m or E_i. In the latter case ($b \in E_i$), b appears after a in L by construction of L. If $b \in S_m$ or $b \in R_m$ then b is in the set corresponding to a communication event c for which $a \prec b \prec c$. Now b appears in L before c, and (as $a \prec c$), a appears before c. By construction of L, the only possible communication event between b and c (in L) is the event corresponding to c. But a and c are not a corresponding pair, because $a \sim b \sim c$. Thus a appears before b in L.

For a pair (s, r) of corresponding send and receive events, it follows from the construction of L that s appears before r in L (because s appears before r in $\tilde{L}$). Consequently, L is a linear extension of $(C, \prec)$.

Second, we must show that L is non-separated. If (s, r) is a pair of corresponding send and receive events, then $s, r \in \tilde{C}$, and the pair is not separated in $\tilde{L}$. As the insertion of the internal events does not separate corresponding pairs, the pair is not separated in L. □

4.3 The crown criterion

If a computation is RSC, then there exists a linear extension of $\prec$ such that corresponding communication events are consecutive (cf. Definition 3.6). This means that it should be possible to schedule whole messages at once, as discussed in Sect. 2.3, and consequently there must be no cyclic dependency on messages. In this section, we show that a cyclic dependency relation on messages is precisely captured by the existence of crowns whose existence can easily be checked with a simple algorithm.

For a given computation C, consider the decomposition of its reduced computation $\tilde{C}$ into ordered pairs $[s, r]$ of corresponding send and receive events (with $s \prec r$) and a relation $\lhd$ on these pairs (i.e., messages) induced by the partial order $\prec$ on $\tilde{C}$. That is, $[s, r] \lhd [s', r']$ if and only if $s \prec s'$ or $s \prec r'$ or $r \prec s'$ or $r \prec r'$. Since $s \prec r$ and $s' \prec r'$, this disjunction is equivalent to $s \prec r'$. Let $G_\lhd$ denote the associated directed graph whose vertices are the pairs $[s, r]$ and which contains an arc from $[s, r]$ to $[s', r']$ if and only if $[s, r] \lhd [s', r']$. Then the relation $\lhd$ on the set of ordered pairs $[s, r]$ is a partial order if and only if $G_\lhd$ has no directed cycle.

Lemma 4.3. *$G_\lhd$ has a directed cycle if and only if the computation C has a crown.*

Proof. $G_\lhd$ has a directed cycle $[s_1, r_1], \ldots, [s_k, r_k], [s_1, r_1]$ if and only if C has k pairs of corresponding send and receive events such that for all i (modulo k), $s_i \prec r_{i+1}$, which is a crown. □

We can now state and prove the main result of Sect. 4 which yields a simple characterization of RSC-computations:

Theorem 4.4 (Crown criterion). *A computation C is RSC if and only if C contains no crown.*

Proof. Let C be a computation. Form the reduced computation $\tilde{C}$ by removing all internal events of C. By Lemma 4.2, C is RSC if and only if $\tilde{C}$ is RSC. By definition, $\tilde{C}$ is RSC if and only if $(\tilde{C}, \prec)$ admits a non-separated linear extension, that is, a linear extension which is a concatenation of pairs $[s_i, r_i]$ of corresponding send and receive events.

Such a linear extension exists if and only if the set of ordered pairs of corresponding send and receive events together with the relation $\lhd$ is a partial order. But this is equivalent to the condition that the associated graph $G_\lhd$ has no directed cycle. By Lemma 4.3 $G_\lhd$ has a directed cycle if and only if C contains a crown. It follows that C is RSC if and only if C contains no crown. □

Since there exist linear time algorithms which check whether a given graph contains a crown (see, e.g., Bouchitte and Habib [9]), Theorem 4.4 provides an efficient RSC-criterion.

4.4 Message scheduling

Since $\lhd$ is a relation on corresponding send and receive events, it captures the idea of *scheduling whole messages* at once. Informally, such a schedule is possible if during the execution of the computation there is always a pair of corresponding communication events which do not depend on "later" communication events that must be executed first.

One advantage of the characterization of RSC by the $\lhd$-relation is that it is easy to check whether $\lhd$ is a partial order. More precisely, given a computation C it is easy to check whether C is RSC (i.e., whether it is crown-free) by only considering the relation $\lhd$ on the pairs of corresponding send and receive events. The idea is to try to construct a linearization by applying a topological sorting scheme. If by successively removing minimal elements the scheme will eventually terminate with the empty set, then there exists a (non-separated) linear extension. Otherwise there is a cycle which cannot be broken. Given a space-time diagram of a finite distributed computation there is an obvious algorithm which implements that scheme:

1. Remove all internal events.
2. While this is possible, take a pair $[s,r]$ that is minimal with respect to $\lhd$. That is, take a leftmost event on one of the process axes which has a corresponding event on another process axis that is also a leftmost event, and remove both events.
3. The computation is RSC if and only if after step 2 all events have been removed.

Examples are given in Fig. 1 (which is not RSC because $[g, h]$ cannot be removed before $[e, f]$ and vice versa), Fig. 4 (which is RSC), and the structurally similar Figs. 3, 5, and 6 which are not RSC as already discussed in Sect. 2.3.

The message-scheduling characterization of RSC-computations is of interest because it indicates again that in the synchronous case a pair of corresponding send-receive events can be regarded as a single combined communication event: whereas in general A-computations the events must not form a cycle with respect to $\prec$, in RSC-computations the combined communication events must not form

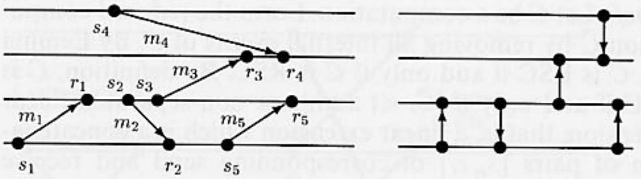

Fig. 8. An RSC-computation with vertical message arrows

a cycle with respect to $\lhd$. For a given space-time diagram there is a simple way to determine whether $\lhd$ is a partial order. If (and only if) it is possible to form a cycle by moving along message arrows in either direction, but always from left to right along process lines, then $\lhd$ contains a cycle and the computation is not RSC. This criterion is again an immediate consequence of the previously mentioned definitions and results.

The above-mentioned algorithm for checking the message scheduling criterion by successively removing pairs of corresponding events gives rise to another, graphical characterization of RSC-computations. According to Definition 3.6, an RSC-computation C has a non-separated linear extension and consequently there exists a mapping $t: C \rightarrow \mathbb{R}$ such that[11]

1. $\forall (s, r) \in \Gamma : t(s) = t(r)$ and
2. $\forall (a, b) \in C^2 \backslash \Gamma : a \prec b \Rightarrow t(a) < t(b)$.

If $t(m)$ denotes the value $t(s) = t(r)$ for message $m = [s, r]$, then the above conditions imply that $t(m) < t(m')$ if $m \lhd m'$. A possible interpretation of $t(s) = t(r)$ is that corresponding send and receive events "happen at the same time" (i.e., message transmission is instantaneous).

This result allows us to draw space-time diagrams of RSC-computations in such a way that all message arrows are vertical. In such a diagram, an event e is drawn to the left of another event e' if e' causally depends on e and if $(e, e') \notin \Gamma$; Fig. 8 shows an example. By following the "causality paths" in the space-time diagram from send events to receive events one can easily check that $m_1 \lhd m_2 \lhd m_3 \lhd m_4$ and $m_3 \lhd m_5$. Messages m_4 and m_5 are not related by $\lhd$, i.e., $\neg (m_4 \lhd m_5)$ and $\neg (m_5 \lhd m_4)$.

Obviously, a computation for which it is possible to draw a space-time diagram with only vertical message arrows admits a non-separated linear extension and is therefore realizable under synchronous communication. Hence, from an informal point of view a computation C is RSC if and only if in a space-time diagram of C all message arrows can be made vertical by moving events along the process lines without changing their relative order on such a line (i.e., using only "elastic deformations"):

Observation 4.5 (All vertical message arrows criterion). *A distributed computation is RSC if and only if its space-time diagram can be drawn in such a way that all message arrows are vertical.*

Comparing this to Observation 3.14 shows again that all RSC-computations are CO (but not necessarily conversely). In the next section we shall see that the drawing of space-time diagrams with vertical message arrows only is in fact justified for all computations with *synchronous* communications. This explains why in proofs of distributed algorithms with synchronous communications one may make use of the fact that there are no messages in transit – an assumption which is usually wrong in the asynchronous case where in invariant based proofs one must in general take into account states where a message has been sent but not yet received [23].

5 Computations with synchronous communications

In our model, a distributed computation is essentially characterized by its space-time diagram (i.e., the sequences of events C_i and the set of pairs of send-receive events Γ). Such space-time diagrams, however, must be feasible in the sense that the causality relation does not contain cycles. For A-computations, the causality relation is defined by the three axioms AS1–AS3 (Definition 3.1), which reflect the intuitive understanding of asynchronous message transmissions. These rules, however, do not capture the additional causal dependencies between events induced by *synchronous* communications. For example, it seems to be common understanding that in S-computations (i.e., computations with synchronous communications) the actions performed by a process after the *sending* of a message m causally depend on the *receipt* of m. Hence, somewhat different axioms must be found to suitably define S-computations.

5.1 The axioms of synchronous communications

In the following, we shall try to axiomatize the causality relation between events in S-computations. To this end, the semantics of synchronous message passing must be made precise. A common understanding of synchronous communication is that the sending of a message is a blocking event, i.e., a message cannot be sent if its receiver is not ready to receive it. As is the case with asynchronous communication, receiving a message is also blocking. In other words, if in synchronous mode a process P wants to send a message to Q, and process Q wants to receive from process P, both P and Q can achieve nothing until the message has been exchanged. Thus an event that causally depends on the sending of a message also depends on the corresponding receipt (and vice versa). If $\ll$ denotes the causality relation in an S-computation C, this "strong common future" condition can be stated more formally as follows:

$$\forall (s, r) \in \Gamma,\ \forall a \in C : s \ll a \Leftrightarrow r \ll a.$$

Similarly, if a receive event or a send event depends on some other event a, the corresponding communication event cannot be executed before a – it also depends on a. This is captured by the "strong common past" property:

$$\forall (s, r) \in \Gamma,\ \forall a \in C : a \ll s \Leftrightarrow a \ll r.$$

[11] Here $\mathbb{R}$ is used as the standard model of time. However, a similar mapping into $\mathbb{N}$ can equivalently be required.

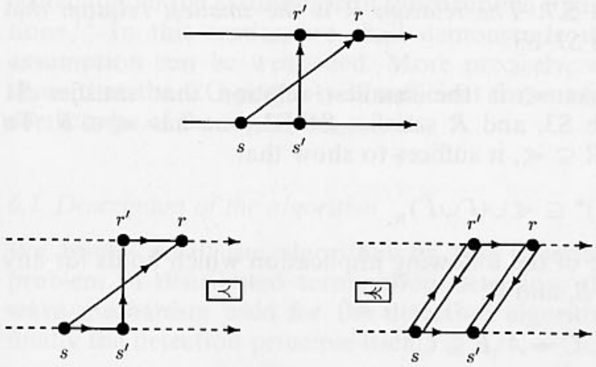

Fig. 9. A space-time diagram and the corresponding relations $\prec$ and $\ll$

These two conditions replace property AS2 ($\forall (s,r) \in \Gamma : s \prec r$) of asynchronous communications (see Definition 3.1). As it is the case for asynchronous communication (cf. AS1 and AS3), causality should, in order to be meaningful, respect the occurrence order of events within a process and be transitive. This leads to the following axiomatization, originally given by Fidge [21, 22].

Definition 5.1 (Synchronous causality relation). The synchronous causality relation $\ll$ in C is the smallest relation that satisfies the following properties.

S1 If $a \prec_i b$, then $a \ll b$.

S2 If $(s,r) \in \Gamma$, then $\forall a \in C : a \ll s \Leftrightarrow a \ll r$ and $s \ll a \Leftrightarrow r \ll a$.

S3 If $a \ll b$ and $b \ll c$, then $a \ll c$.

Observe that in contrast to asynchronous communications, corresponding send and receive events are in general not causally related by $\ll$. On the other hand, however, S2 is symmetric with respect to send and receive events. As an example, Fig. 9 exhibits a computation together with the graphs of the two relations $\prec$ and $\ll$; the arcs induced by the transitivity axioms AS3 or S3 are not shown. One observes that the graph for $\ll$ contains cycles, whereas the graph for $\prec$ does not.

S-computations are defined in a way similar to A-computations (Definition 3.2); in particular, the causality relation must not form cycles. This yields the following definition.

Definition 5.2 (S-computations). A computation with synchronous communications (or an S-computation for short) of a distributed system consisting of processes $P_1, \ldots, P_n$ is an n-tuple $C = (C_1, \ldots, C_n)$ of local computations together with a set Γ of corresponding communication events, for which the synchronous causality relation $\ll$ is a partial order.

As the example of Fig. 9 shows, a space-time diagram depicts the sequences of local events C_i and the "message set" Γ, but it does not explicitly show the causality order $\prec$ or $\ll$. Of course, space-time diagrams do only represent valid computations if the corresponding causality relations are free of cycles. There are, in fact, space-time diagrams which represent A-computations but not S-computations (cf. Fig. 5, the reader may easily verify that $x_1 \ll x_2$, $x_2 \ll x_3$, and $x_3 \ll x_1$ holds, where x_i stands for s_i and r_i). Of particular interest are space-time diagrams which represent both, A-computations and S-computations. This case will be considered next.

5.2 S-computations and RSC-computations

It is quite natural to compare the class of RSC-computations (based on asynchronous communications) to the class of S-computations. In fact, the main justification for Definition 3.6 (RSC) and Definition 5.2 (S-computations) is given by the following "equivalence" theorem.

Theorem 5.3 (S-computations ≡ RSC). *Let C be an A-computation. The following statements are equivalent:*

1. *C is RSC;*

2. *C is an S-computation.*

Note that in case 1 the causality relation is $\prec$, whereas in case 2 it is $\ll$.

Proof. It has to be shown that $\ll$ is a partial order if and only if C is RSC. This will be done in the remainder of this section with the help of some technical lemmas.

Let C be an A-computation and let $\prec$ denote its (asynchronous) causality relation. Because in S-computations corresponding send and receive events are symmetric with respect to causality, we complement Γ by the converse relation

$$\bar{\Gamma} = \{(r,s) : (s,r) \in \Gamma\}$$

and we consider the smallest relation (denoted by $\bar{\prec}$) that satisfies AS1 and AS3 (with $\prec$ replaced by $\bar{\prec}$), and

$\overline{\textbf{AS2}}$ $\forall (s,r) \in \Gamma : r \bar{\prec} s$.

Informally, $a \bar{\prec} b$ holds if in a space-time diagram b can be reached from a via a path that runs from left to right on process lines but follows "message arrows" in the reversed direction. Next define

$$R' = (\prec \cup \bar{\prec})^*$$

$$R = (\prec \cup \bar{\prec})^* \setminus (\Gamma \cup \bar{\Gamma}).$$

Informally, $aR'b$ holds if in a space-time diagram b can be reached from a via a path that runs from left to right on process lines but may follow message arrows in either direction. R is similar to R' but excludes corresponding send and receive events and thus avoids cycles of length two between corresponding communication events. We shall use the following technical lemma several times.

Lemma 5.4. *Let a and b be two events of a computation C. If aRb, then $a \prec b$ or there exists a sequence $\langle (s_i, r_i), i \in \{1, \ldots, k\} \rangle$ of pairs in Γ such that*

$$a \preceq \cdots \preceq r_1 \bar{\prec} s_1 \prec \cdots \prec r_i \bar{\prec} s_i \prec \cdots$$
$$\prec r_{i+1} \bar{\prec} s_{i+1} \prec \cdots \prec r_k \bar{\prec} s_k \preceq \cdots \preceq b.$$

Proof. This lemma is a straightforward consequence of the definition of R and $\bar{\prec}$ and of the fact that the relations $\prec$ and $\bar{\prec}$ both satisfy AS1. □

For the proof of Theorem 5.3 it will first be shown that $1 \Rightarrow 2$. To this end we shall demonstrate that for RSC-computations the relation R coincides with $\ll$ (i.e., that it is the smallest (Lemma 5.7) relation satisfying S1–S3 (Lemma 5.6)), and that it is cycle-free (Lemma 5.5). Assume that C is an RSC-computation.

Lemma 5.5. *The relation R as defined for C is irreflexive.*

Proof. Suppose that R is not irreflexive, i.e., there exists an event a in C such that aRa. It follows from Lemma 5.4 that there exists a chain

$$a \preceq \cdots \preceq r_1 \bar{\prec} s_1 \prec \cdots \prec r_i \bar{\prec} s_i \prec \cdots$$
$$\prec r_{i+1} \bar{\prec} s_{i+1} \prec \cdots \prec r_k \bar{\prec} s_k \preceq \cdots \preceq a.$$

It follows that $s_1 \prec r_2, \ldots, s_i \prec r_{i+1}, \ldots, s_k \prec r_1$, hence the sequence $\langle (s_i, r_i), i \in \{1, \ldots, k\} \rangle$ is a crown. By Theorem 4.4 this is a contradiction to the assumption that the computation is RSC. Consequently, R is irreflexive. □

Lemma 5.6. *The relation R satisfies S1–S3.*

Proof:
Property S1. If $a \prec_i b$ then $a \prec b$ and $(a, b) \notin \Gamma \cup \bar{\Gamma}$, hence aRb.

Property S2. For the first part of S2 ("strong common past") let $(s, r) \in \Gamma$, and assume that aRs. It follows that $aR's$. By the inclusion $\Gamma \subseteq R'$ and the transitivity of R' we get $aR'r$. By the definition of $\bar{\Gamma}$, (a, r) does not belong to $\bar{\Gamma}$. If (a, r) belongs to Γ then $a = s$, which contradicts Lemma 5.5. Hence $aR'r$ and $(a, r) \notin \Gamma \cup \bar{\Gamma}$, and consequently we have aRr. Thus $aRs \Rightarrow aRr$. The converse implication $aRr \Rightarrow aRs$ is proved similarly, and the second part of S2 ("strong common future") is proved in the same way as the first part.

Property S3. Let a, b, c be three events in C such that aRb and bRc. As $R \subseteq R'$ and since R' is transitive we have $aR'c$. To prove that aRc, it remains to show that $(a, c) \notin \Gamma \cup \bar{\Gamma}$ which will be done by contradiction, using that C is RSC. Assume that $(a, c) \in \Gamma \cup \bar{\Gamma}$. First consider the case $a \prec b$ and $b \prec c$. Then $a \prec c$, so $(a, c) \in \Gamma$, and Proposition 3.13 implies that C is not CO, contradicting the assumption that C is RSC. Hence, Lemma 5.4 applies to the pair a, b or b, c and it follows that there exists a chain

$$a \preceq \cdots \preceq r_1 \bar{\prec} s_1 \prec \cdots \prec r_i \bar{\prec} s_i \prec \cdots$$
$$\prec r_{i+1} \bar{\prec} s_{i+1} \cdots r_k \bar{\prec} s_k \preceq \cdots \preceq c.$$

1. If $(a, c) \in \Gamma$ then by denoting $(a, c) = (s_0, r_0)$ we get that the sequence $\langle (s_i, r_i), i \in \{0, \ldots, k\} \rangle$ is a crown.
2. If $(a, c) \in \bar{\Gamma}$ then by denoting $(a, c) = (r_0, s_0)$ we have $s_k \prec c = s_0 \prec r_0$ and $s_0 \prec r_0 = a \prec r_1$. Hence the sequence $\langle (s_i, r_i), i \in \{0, \ldots, k\} \rangle$ is a crown.

In both cases C contains a crown, contradicting the assumption that C is RSC (Theorem 4.4). Hence $(a, c) \notin (\Gamma \cup \bar{\Gamma})$ and thereby aRc. □

Lemma 5.7. *The relation R is the smallest relation that satisfies S1–S3.*

Proof. As $\ll$ is the smallest relation that satisfies S1 through S3, and R satisfies S1–S3, one has $\ll \subseteq R$. To prove $R \subseteq \ll$, it suffices to show that

$$(\prec \cup \bar{\prec})^* \subseteq \ll \cup (\Gamma \cup \bar{\Gamma}),$$

because of the following implication which holds for any sets A, B, and C:

$A \subseteq B \cup C \Rightarrow A \backslash B \subseteq C$.

Clearly we have $\prec \subseteq \ll \cup (\Gamma \cup \bar{\Gamma})$ and $\bar{\prec} \subseteq \ll \cup (\Gamma \cup \bar{\Gamma})$, so it suffices to show that for two events a and b in C such that

$$a(\prec \cup \bar{\prec})^* b \quad \text{and} \quad \neg (a \prec b) \quad \text{and} \quad \neg (a \bar{\prec} b),$$

the relation $a \ll b$ holds. For that, consider a shortest chain

$$a \preceq \cdots \preceq r_1 \bar{\prec} s_1 \prec \cdots \prec r_i \bar{\prec} s_i \prec \cdots$$
$$\prec r_{i+1} \bar{\prec} s_{i+1} \prec \cdots \prec r_k \bar{\prec} s_k \preceq \cdots \preceq b$$

from a to b. In a shortest chain $a \neq s_1$ and $b \neq r_k$ and $\forall i \in \{1, \ldots, k-1\}: r_i \neq r_{i+1}$. By the first part of S2 and the inclusion $\prec \subseteq \ll \cup (\Gamma \cup \bar{\Gamma})$, these last relations show that $r_1 \ll \cdots \ll r_k$. The following three cases are exhaustive:

1. If $b \neq s_k$, we have $r_k \ll b$, because of the second part of S2 and since $b \neq r_k$. In the same way we have $a \ll r_1$ or $a = r_1$. In these two cases we conclude by S3 that $a \ll b$.
2. If $b = s_k$ and $k \geqq 2$, then we have $r_{k-1} \ll b$ and therefore we get $a \ll b$.
3. If $k = 1$ and $b = s_1$, it follows from $\neg (a \bar{\prec} b)$ that a and r_1 are two distinct events. As $a \neq s_1$, we deduce by the second part of S2 that $a \ll r_1$ and hence $a \ll b$. □

As R coincides with $\ll$ and R is irreflexive, it can be concluded that $\ll$ is cycle-free, and hence C is an S-computation.

Conversely, it will now be shown that $2 \Rightarrow 1$ in Theorem 5.3. Let C be an S-computation, and assume that C is not RSC. By Theorem 4.4, C (with the asynchronous causality relation $\prec$) contains a crown that we denote $\langle (s_i, r_i), i \in \{1, \ldots, k\} \rangle$. Thus we have

$$s_1 \prec r_2 \bar{\prec} s_2 \prec r_3 \bar{\prec} \cdots \bar{\prec} s_{i-1} \prec r_i \bar{\prec} s_i$$
$$\prec r_{i+1} \bar{\prec} \cdots \bar{\prec} r_k \bar{\prec} s_k \prec r_1 \bar{\prec} s_1$$

By setting $a = s_1$ and $b = s_1$ in the chain used in the proof of Lemma 5.7, we get $s_1 \ll s_1$ which implies that $\ll$ in C is not cycle-free. This contradicts the fact that C (with $\ll$) is an S-computation.

This concludes the proof of Theorem 5.3. □

6 A termination detection algorithm revisited

The termination detection algorithm by Dijkstra, Feijen, and Van Gasteren [19] is an example of a well-known distributed control algorithm which was proved to be correct under the assumption of synchronous communications.[12] In this section we shall demonstrate that this assumption can be weakened. More precisely, we shall show that the CO-property is sufficient for ensuring the correctness of the algorithm.

6.1 Description of the algorithm

We briefly recall the algorithm by first describing the problem of distributed termination detection, then the wave mechanism used for the detection algorithm, and finally the detection principle itself.

The Problem. Let $P_0, P_1, \ldots, P_{n-1}$ be n processes, each of which can be either *active* or *passive*. Some processes might be active initially, at the start of the computation. An active process can become passive by itself, but a passive process can only become active when it receives a so-called *basic message*. Such messages can only be sent by active processes, from which it follows that if all processes are passive and no basic messages are in transit, the computation is *terminated*. The problem is to detect this termination state by a superimposed distributed control algorithm.

The wave mechanism. The control algorithm is based on a succession of *waves*, where each wave consists of a *visit event* in each process. A wave is implemented by the propagation of a *token* along a logical ring, to which end it is assumed that process P_{j+1} can send *control messages* to process P_j (all indices are modulo n). Basic messages are not restricted to pass on the ring; a process may send a basic message to any other process. A wave is initialized by P_0 by sending the token to process P_{n-1}.

The visit event happens at P_j when P_j sends the token to its neighbor P_{j-1}. (The first sending of the token by P_0 is left out of consideration.) The i^{th} sending of the token by P_j, i.e., the visit of P_j by the i^{th} wave, is denoted by $S_j^{(i)}$. Thus the i^{th} wave is formally defined by the set of events

$$W^{(i)} = \{S_0^{(i)}, \ldots, S_{n-1}^{(i)}\}.$$

The i^{th} receipt of the token by P_j is denoted by $R_j^{(i)}$. A process may send the token only if it holds the token. This condition is formally expressed by $R_j^{(i)} \prec_j S_j^{(i)}$.

As the event $S_j^{(i)}$ corresponds to the event $R_{j-1}^{(i)}$ for $j > 0$, we obtain that the visits of one wave are totally ordered by causality:

$$j > k \Leftrightarrow S_j^{(i)} \prec S_k^{(i)}. \tag{A}$$

As $S_0^{(i)}$ corresponds to $R_{n-1}^{(i+1)}$, the visits of subsequent waves are ordered by causality:

$$\forall i, j, k : S_j^{(i)} \prec S_k^{(i+1)}. \tag{B}$$

In (A) and (B), i ranges over wave numbers and j and k over process indices.

The Detection Principle. A basic message $m = [s, r]$ from P_j to P_k is said to *cross* wave $W^{(i)}$ if $s \prec_j S_j^{(i)}$ and $S_k^{(i)} \prec_k r$. The detection principle relies on the following observation.

Lemma 6.1 (Termination after a wave). *If for each j, P_j is passive when $S_j^{(i)}$ occurs and there is no basic message that crosses wave $W^{(i)}$, no process is ever active after its visit by wave $W^{(i)}$.*

Proof. Call an event e of process P_j a *post-i* event if $S_j^{(i)} \prec_j e$, and assume there is a post-i event of the underlying basic computation. Choose among those events an event that is minimal with respect to $\prec$. The minimality and the fact that a process is passive when it is visited by the wave imply that this event is a receive event. Again from its minimality *among post-i events* it follows that the corresponding send event is not a post-i event, hence the corresponding message crosses wave $W^{(i)}$. □

It can easily be ensured that for each j, P_j is passive when $S_j^{(i)}$ occurs, namely by requiring that a process sends the token only when it is passive. A consequence of the lemma is then that termination can be concluded if there is a wave which is not crossed by a message. To detect whether a message could have crossed the current wave, the algorithm uses a simple coloring scheme.

At the initialization of each wave, the token implementing the wave is white. Process P_j blackens the token in wave $W^{(i)}$ if and only if P_j sent a (basic) message to a process P_k with $k > j$ between the events $S_j^{(i-1)}$ and $S_j^{(i)}$. If any process blackens the token during wave $W^{(i)}$, then the token remains black until the end of $W^{(i)}$. Process P_0 declares termination (at the end of wave $W^{(i)}$) when it receives a white token and did not send a basic message between $S_0^{(i-1)}$ and $S_0^{(i)}$. If the token returns black or P_0 sent a basic message between $S_0^{(i-1)}$ and $S_0^{(i)}$, a new wave is initiated.

6.2 Correctness proof with the CO-assumption

Dijkstra et al. [19] prove the correctness of this termination detection algorithm for S-computations by showing its safety and liveness. The *safety* proof is based on invariants and relies on a representation of a send event and its corresponding receive event as a single atomic action ("the activation of a machine by an active machine"). In contrast to this, our proof for the CO-case does not rely on assertions that remain invariant in each global state, but considers the execution as a whole. However, invariant-based proofs are also possible as was recently shown by Stoller and Schneider [43].

The idea of our proof is that under the CO-assumption a basic message cannot cross two (or more) waves – a message that crosses a single wave, however, is detected by the coloring scheme. Informally, the single wave crossing property follows directly from the vertical message arrow criterion for CO-computations: Consider a message m that crosses a given wave. According to Observation 3.14 it is possible to draw a space-time diagram of the computation in such a way that m is symbolized by a vertical arrow. Since at any instance in time there exists at most one wave (graphically represented by a "diagonal" chain of

[12] As remarked before, the authors use the term "instantaneous" instead of "synchronous".

messages), such a vertical arrow cannot cross more than one wave. The following theorem makes use of this argument in a more formal way.

Theorem 6.2 (Safety). *If the combined computation of the basic and the control computations satisfies the CO-property, the distributed termination detection algorithm is safe.*

Proof. Suppose P_0 detects termination at the end of $W^{(i)}$ and suppose that there is a message $m=[s,r]$ sent by P_j and received by P_k that crosses $W^{(i)}$. As m crosses $W^{(i)}$, we have $s \prec S_j^{(i)}$ and $S_k^{(i)} \prec r$. Since the token remains white during $W^{(i)}$ and P_0 did not send a message between $S_0^{(i-1)}$ and $S_0^{(i)}$, either m is sent before $S_j^{(i-1)}$, or m is sent between $S_j^{(i-1)}$ and $S_j^{(i)}$ and $j>k$. In the first case, we have $s \prec S_j^{(i-1)}$. Using property (B) we obtain $s \prec S_j^{(i-1)} \prec S_k^{(i)} \prec r$, which violates the EI-property and hence contradicts the CO-assumption. In the second case, using property (A) we obtain $s \prec S_j^{(i)} \prec S_k^{(i)} \prec r$, which again violates the EI-property and hence contradicts the CO-assumption.

Thus, by Lemma 6.1 the algorithm is safe if the computation satisfies the CO-property. □

The *liveness* of the algorithm is demonstrated by considering what happens after termination of the basic computation; no basic messages are sent, so no process is blackened after termination. This implies that if a wave is started after termination, at the end of that wave all processes are white; during the next wave the token remains white, and termination is detected at the end of that wave. This argument, already given by Dijkstra et al., still applies when CO-computations are considered instead of the synchronous model.

Topor presents a similar termination detection algorithm for synchronous communications (based on a control tree [46]), and claims that the algorithm is also correct "provided the delay between sending and receiving a message is sufficiently small". Katz [26] more precisely indicates the meaning of "sufficiently small" by requiring that "a basic message is sensed by the receiving process before the wave can traverse the tree". As shown above, this aspect (called *semi-synchrony* by Katz) is indeed implied precisely by the CO-property.

An immediate practical consequence of Theorem 6.2 is that termination detection for CO-computations is not more difficult than termination detection for S-computations, despite the increased level of non-determinism and parallelism. However, it is easy to construct a scenario showing that the algorithm is no longer safe when the computations satisfy only the FIFO-property, but not the CO-property: it suffices to consider a basic message, sent via an edge not belonging to the control ring, that crosses two or more waves. Thus the CO-property is the weakest assumption of our hierarchy that implies the safety of this algorithm.

To safely detect the termination of non-CO-computations, more sophisticated termination detection algorithms must be used, which deal with in-transit messages explicitly. Methods to ensure the absence of in-transit basic messages include the use of send/receive counters [18, 30], acknowledgements [33], timers [45], or special marker-messages in case of FIFO-communication [30, 45].

7 Further remarks and conclusions

In this section, we briefly discuss some implementation aspects, mention some other communication schemes, and summarize our main results in a graphical scheme.

7.1 Implementation

We now sketch how the various synchronism assumptions considered in this article may be implemented on a reliable distributed system with asynchronous communication. We briefly describe possible communication protocols that can be used by the processes in order to guarantee that the resulting computations satisfy the FIFO, CO, or RSC-assumption. Such a protocol constitutes a layer between the processes and the message passing system. A process "sends" a message by handing it to the sender protocol, and receives a message by "accepting" it from the receiver protocol. We only sketch the main ideas, for the details the reader is referred to the literature.

FIFO. The FIFO-property only imposes an order on the receive events of messages exchanged between the same two processes, and consequently can be implemented on each communication line separately. A protocol which uses *sequence numbers* is well known: The sender appends a sequence number to each message, and the receiver accepts only the "next" message from its input buffer.

Causal order. A protocol for enforcement of causal order is more involved than the above-mentioned FIFO-protocol because it must also deal with the causal relation between send events at different processes. In contrast to the RSC-property, however, causal order message delivery can be realized without blocking the sender, thus reducing the possibility of introducing deadlocks.

A possible implementation, relying on sets of *vector timestamps*, was described by Schiper, Eggli, and Sandoz [37]. A slight variant which uses *integer matrices* of size $n \times n$ (i.e., vectors of vectors of length n, where n is the number of processes) was later given by Raynal, Schiper, and Toueg [35]. In this protocol, each process P_i has such a matrix M_i which is initialized to the null matrix. When process P_i sends a message to P_j, it increments $M_i[i,j]$ and attaches M_i to the message. Whenever a process P_j receives a message together with its attached matrix M, M_j is updated to the component-wise maximum of M and M_j. By that, the matrix of a message encodes the knowledge of all send events (i.e., other messages) it causally depends on: if $M[i,j]=k$, then it causally depends on the first, the second, …, the k^{th} sending of a message from P_i to P_j (with possible exceptions of reflexive dependencies).

A message that arrives "out of causal order" at some process P_j depends on a send event whose message has not yet been received by P_j. This can easily be detected by P_j by using a scheme that is a straightforward generalization of the FIFO-protocol mentioned above. The message which arrives "too early" is then simply delayed by the protocol and only delivered when its receipt does not violate the CO-property. (The related update of M_j is likewise deferred.) More formally, let M denote the matrix attached to a message sent by process P_i to P_j. The message is not delivered to P_j until[13]

1. $\forall k \neq i : M[k,j] \leqq M_j[k,j]$ and
2. $M[i,j] = M_j[i,j] + 1$.

Clearly, (2) guarantees FIFO-order between P_i and P_j. Delivery condition (1) essentially requires that (with respect to all other potential senders) local vector time at P_j represented by the vector $M_j[*,j]$ be greater than the vector timestamp $M[*,j]$ of a message that is accepted by P_j.

Note that if only messages directed towards a *single* location must be causally ordered (e.g., to realize a causally consistent monitor or causal memory [1]) or if causally ordered broadcasts are considered [8, 37], then the matrix can be reduced to a *vector* by simply "ignoring" its second dimension.

Interestingly, causal order message delivery can also be implemented without vectors or matrices by using input-output message buffers that communicate by a handshake protocol [31]. It should also be noted that if global (or "real") time is available (realized, for example, by sufficiently well synchronized physical clocks) and if lower and upper bounds on message transmission times are known, then other means to guarantee causal order are feasible. One such possibility consists in delaying all actions in such a way that no direct or indirect message overtaking is possible [27].

Synchronous communications. For point-to-point communications with unconditional send-receive statements, the RSC-property can be obtained by sending an *acknowledgement* back to the sender for each message received and block the sender until the corresponding acknowledgement arrives (see, e.g., Martin [29] or Seitz [40]). From the viewpoint of the sender, the sending of a message is instantaneous since all its activities are frozen after the start of the send operation, and it is only unfrozen when it is known that the message has been received. Since the sender is in a blocked state when the receiver sends the acknowledgement, there is indeed an instant in time where both processes are engaged in the communication operation simultaneously. Conceptually, one could imagine that the message is sent instantaneously at that moment. The enumeration of all events of the computation in the order of their occurrence (with a send enumerated before its corresponding receive) now gives a non-separated linear extension of the causal order, hence by Definition 3.6 the computation is RSC. In a less formal way this also follows from Observation 4.5 since for computations with instantaneous messages it is possible to draw a space-time diagram with only vertical message arrows.

7.2 Conclusions

In this article we have studied the structural aspects of distributed computations, depending on the degree of

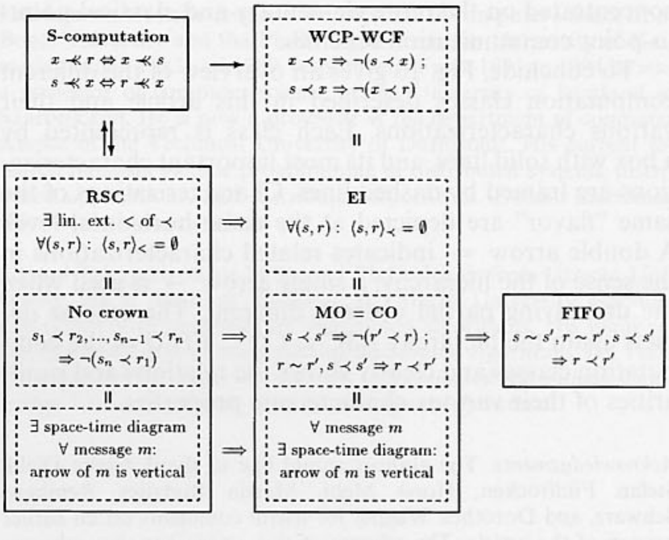
S-computation
$x \prec\!\!\!\prec r \Leftrightarrow x \prec\!\!\!\prec s$
$s \prec\!\!\!\prec x \Leftrightarrow r \prec\!\!\!\prec x$
WCP-WCF
$x \prec r \Rightarrow \neg(s \prec x)$;
$s \prec x \Rightarrow \neg(x \prec r)$
RSC
$\exists$ lin. ext. $<$ of $\prec$
$\forall(s,r) : \langle s,r \rangle_< = \emptyset$
EI
$\forall(s,r) : \langle s,r \rangle_\prec = \emptyset$
No crown
$s_1 \prec r_2, \ldots, s_{n-1} \prec r_n$
$\Rightarrow \neg(s_n \prec r_1)$
MO = CO
$s \prec s' \Rightarrow \neg(r' \prec r)$;
$r \sim r', s \prec s' \Rightarrow r \prec r'$
FIFO
$s \sim s', r \sim r', s \prec s'$
$\Rightarrow r \prec r'$
$\exists$ space-time diagram
$\forall$ message m:
arrow of m is vertical
$\forall$ message m
$\exists$ space-time diagram:
arrow of m is vertical

Fig. 10. The different computation classes and their characterizations

synchrony implied by the communication mechanism.[14] While usually only fully asynchronous, FIFO, and synchronous communications are considered in the design of distributed algorithms, our study reveals that the CO-assumption is also useful.

In practice, it is often possible to mix different communication mechanisms in a single distributed program. For example, message passing libraries such as MPI [24] support communication primitives with various synchronization and message selection properties. The exact semantics of their interplay, however, is usually not formally defined and therefore often remains unclear (for a critique, see Cypher and Leu [16]). Also, many distributed operating systems (see, e.g., Tanenbaum [44]) allow the use of blocking or non-blocking send operations, and some distributed programming languages (see, e.g., Bal [4] and Shatz [41]) provide synchronous as well as asynchronous communication features. Using axioms AS2 or S2 when appropriate, it should be possible to formally define the causality relation for such general computations.

More communication and synchronization schemes for distributed systems have been proposed in the literature. Examples include: *flush channels* as a weakening of the FIFO-protocol [2]; channels with *bounded buffering* capacity allowing a variable degree of synchronization freedom from strictly synchronous to totally asynchronous communication by adjusting the so-called synchronization slack between two communicating processes [29]; *multiway rendezvous* as a generalization of synchronous message exchange where an arbitrary number of concurrent processes participate in the execution of a single event at the same time [15, 20]; *remote procedure call* abstractions [5, 11, 34]; and various *broadcast* and *multicast* schemes, ranging from weakly synchronized and causally ordered variants to so-called atomic broadcasts [8, 39]. Analyzing and discussing the semantics of those mechanisms, however, is out of scope of this article, which

[13] This is a slight optimization compared to the solution by Raynal et al. where an extra vector is needed in every process.

[14] A study with a similar goal [42] was published while our article was under revision.

concentrated on the more elementary and classical point-to-point communication schemes.

To conclude, Fig. 10 gives an overview of the different computation classes described in this article and their various characterizations. Each class is represented by a box with solid lines, and its most important characterizations are framed by dashed lines. Characterizations of the same "flavor" are depicted at the same horizontal level. A double arrow ⇒ indicates related characterizations in the sense of the hierarchy; a single arrow → is used when the underlying partial order is different. The scheme depicts again the hierarchy $RSC \subset CO \subset FIFO$ of the computation classes and clearly shows the relations and similarities of their various characteristic properties.

Acknowledgements. The authors would like to thank Claire Diehl, Stefan Fünfrocken, Horst Mehl, Martin Rudalics, Reinhard Schwarz, and Dorothea Wagner for useful comments on an earlier version of the article. The referees of this article are also acknowledged for their remarks, which helped us to improve the presentation of the article.

References

1. Ahamad M, Burns JE, Hutto PW, Neiger G: Causal memory. In: P Spirakis, S Toueg (eds) Proc 5th International Workshop on Distributed Algorithms. Springer, Berlin Heidelberg New York, LNCS 579: 9–30 (1991)
2. Ahuja M: Flush primitives for asynchronous distributed systems. Inf Proc Lett 34: 5–12 (1990)
3. Baeten JCM, Weijland WP: Process algebra. Vol 18 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990
4. Bal HE: Programming distributed systems. Prentice Hall, 1990
5. Birell AD, Nelson BJ: Implementing remote procedure calls. ACM Trans Comput Syst 3: 39–59 (1984)
6. Birman K, van Renesse R (eds): Reliable distributed computing with the Isis toolkit. IEEE Computer Society Press, 1994
7. Birman K, Joseph TA: Reliable communication in the presence of failures. ACM Trans Comput Syst 5: 47–76 (1987)
8. Birman K, Schiper A, Stephenson P: Lightweight causal and atomic group multicast. ACM Trans Comput Syst 9: 272–314 (1991)
9. Bouchitte V, Habib M: The calculation of invariants for ordered sets. In: I Rival (ed) Algorithms and order. Kluwer 1989, pp 231–279
10. Bougé L: Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP. Theor Comput Sci 49: 145–169 (1987)
11. Brinch Hansen P: Distributed processes: a distributed programming concept. Commun ACM 21: 934–941 (1978)
12. Čapek M: Towards a widening of the notion of causality. Diogenes 28: 63–90 (1959)
13. Čapek M: Time-space rather than space-time. Diogenes 123: 30–49 (1983)
14. Chandy KM, Lamport L: Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3: 63–75 (1985)
15. Charlesworth A: The multiway rendezvous. ACM Trans Program Lang Syst 9: 350–366 (1987)
16. Cypher R, Leu E: The semantics of blocking and non-blocking send and receive primitives. Proc 8th International Parallel Processing Symposium, pp 729–735, 1994
17. Davey BA, Priestley HA: Introduction to lattices and order. Cambridge University Press, 1990
18. Dijkstra EW: Shmuel Safra's version of termination detection. Tech Rep EWD 998, Department of Computer Science, The University of Texas at Austin, 1987
19. Dijkstra EW, Feijen WHJ, van Gasteren, AJM: Derivation of a termination detection algorithm for distributed computations. Inf Proc Lett 16: 217–219 (1983)
20. Evangelist M, Francez N, Katz S: Multiparty interactions for interprocess communication and synchronization. IEEE Trans Softw Eng SE-15: 1417–1426 (1989)
21. Fidge C: Dynamic analysis of event orderings in message passing systems. PhD thesis, Department of Computer Science, The Australian National University, 1989
22. Fidge C: Logical time in distributed computing systems. Computer 24(8): 28–33 (1991)
23. Gribomont EP: From synchronous to asynchronous communication. In: C Rattray (ed) Specification and verification of concurrent systems. Springer, Berlin Heidelberg New York 1990, pp 368–383
24. Hempel R: The MPI standard for message passing. In: W Gentzsch, U Harms (eds) Heigh-performance computing and networking. Springer, Berlin Heidelberg New York, LNCS 797: 247–252 (1994)
25. Hoare CAR: Communicating sequential processes. Commun ACM 21: 666–677 (1978)
26. Katz S: A superimposition control construct for distributed systems. ACM Trans Program Lang Syst 15: 337–356 (1993)
27. Kopetz H: Sparse time versus dense time in distributed real-time systems. Proc 12th Int Conf Distr Computing Sys, pp 460–467, 1992
28. Lamport L: Time, clocks, and the orderings of events in a distributed system. Commun ACM 21: 558–565 (1978)
29. Martin AJ: An axiomatic definition of synchronization primitives. Acta Inf 16: 219–235 (1981)
30. Mattern F: Algorithms for distributed termination detection. Distrib Comput 2: 161–175 (1987)
31. Mattern F, Fünfrocken S: A non-blocking lightweight implementation of causal order message delivery. In: KP Birman, F Mattern, A Schiper (eds) Theory and practice in distributed systems. Springer, Berlin Heidelberg New York, LNCS 938: 197–213 (1995)
32. Milner AJRG: A calculus of communicating systems. Springer, Berlin Heidelberg New York, LNCS 92 (1980)
33. Naimi M: Global stability detection in the asynchronous distributed computations. Proc IEEE Workshop on Future Trends of Distributed Computing Systems in the 90's, Hong Kong, pp 87–92, 1988
34. Nelson BJ: Remote procedure call. Tech Rep CS-81-119, Carnegie-Melon University, 1981
35. Raynal M, Schiper A, Toueg S: The causal ordering abstraction and a simple way to implement it. Inf Proc Lett 39: 343–350 (1991)
36. Renesse R van: Causal controversy at Le Mont St. Michel. Oper Syst Rev 27(2): 44–53 (1993)
37. Schiper A, Eggli J, Sandoz A: A new algorithm to implement causal ordering. In: JC Bermond, M Raynal (eds) Distributed algorithms. Springer, Berlin Heidelberg New York, LNCS 392: 219–223 (1989)
38. Schlichting RD, Schneider FB: Using message passing for distributed programming: proof rules and disciplines. ACM Trans Program Lang Syst 6: 402–431 (1984)
39. Schmuck F: The use of efficient broadcast protocols in asynchronous distributed systems. Tech Rep 88-928, Cornell University, 1988
40. Seitz C: Multicomputers. In: CAR Hoare (ed) Developments in concurrency and communication. Addison-Wesley, 1990, pp 131–201 |
41. Shatz SM: Communication mechanisms for programming distributed systems. Computer 17: 21–28 (1984)
42. Soneoka T, Ibaraki T: Logically instantaneous Message passing in asynchronous distributed systems. IEEE Trans Comput 43: 513–527 (1994) m
43. Stoller SD, Schneider FB: Verifying programs that use causally-ordered message-passing. Science of Computer Programming 24: 105–128 (1995)

44. Tanenbaum A: Distributed operating systems. Prentice-Hall, 1995
45. Tel G: Topics in distributed algorithms. Vol 1 of Cambridge International Series on Parallel Computing. Cambridge University Press, 1991
46. Topor RW: Termination detection for distributed computations. Inf Proc Lett 18: 33–36 (1984)

Bernadette Charron-Bost received her Ph.D. degree in computer science (1989) from Paris 7 University, France. She is a researcher at the CNRS (French National Center of Scientific Research), in the Computer Science Department of the Ecole Polytechnique. Her research interests include program verification and semantics, distributed programming, and distributed algorithms. E-mail: charron@lix.polytechnique.fr

Friedemann Mattern received the Diploma in computer science from Bonn University and the Ph.D. degree from the University of Kaiserslautern in 1983 and 1989, respectively. From 1991 to 1994 he was a professor of computer science at the University of Saarland in Saarbrücken. He is now a professor at the department of computer science of the Technical University of Darmstadt. His current research interests include programming of distributed systems, distributed applications, and distributed algorithms. E-mail: mattern@informatik.th-darmstadt.de

Gerard Tel received his M.Sc. and Ph.D. degree from Utrecht University. He is Assistant Professor at Utrecht University, teaching algorithms with emphasis on distributed algorithms. His main research interest is in message passing distributed algorithms. Dr. Tel is the author of the book *Introduction to Distributed Algorithms*, Cambridge University Press, 1994. E-mail: gerard@cs.ruu.nl