ELSEVIER

Theoretical Computer Science 196 (1998) 45–70

Theoretical Computer Science

A framework for viewing atomic events in distributed computations[1]

Ajay D. Kshemkalyani

Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, P.O. Box 210030, Cincinnati, OH 45221-0030, USA

Abstract

Events in a distributed computation have been implicitly modeled in the literature in the isolated contexts of various applications. This paper presents a unifying framework for expressing and analyzing events at various levels of atomicity in a distributed computation. In the framework, events at any level of atomicity are defined and composed in terms of events at a finer level of atomicity using hierarchical views of the distributed computation. We identify and prove three properties that are satisfied by each level of atomicity. Results based on these properties that hold for any one level of atomicity apply to all levels of atomicity. The properties also show that the global states at the various levels of atomicity correspond to embedded lattices of global states, thereby providing different abstract views of the same computation.

Keywords: Atomicity; Causality; Concurrency; Distributed computation; Communication

1. Introduction

In the literature on distributed system executions (also known as computations), much attention has been focused on modeling events in order to reason better about the system executions. Thus far, events have been implicitly modeled in the isolated contexts of various applications. The events modeled have various levels of atomicity, and there is no prior treatment of the various levels of atomicity in a unifying framework. A formal treatment of grouping events in a distributed execution into higher-level nonatomic events is crucial in modeling distributed activities to provide different abstract views [21, 30]. Event abstraction also provides simplicity to the programmer and system designer in reasoning at the appropriate level of atomicity by reducing the amount of information to be handled. Lamport also argued that it is useful to

E-mail: ajayk@ececs.uc.edu.

[1] This is a revised and expanded version of the paper by the same title that appears in the Proc. Euro-Par'96, Lecture Notes in Computer Science, vol. 1123, Springer, Berlin, 1996, pp. 496–505.

0304-3975/98/$19.00
PII S0304-3975(97)00195-3

assume that primitive elements between which concurrency is modeled are nonatomic for studying basic questions about nonatomicity [30]. This paper provides a unifying framework for expressing and analyzing events at various levels of atomicity in distributed system executions: events at a particular level of atomicity are defined and hierarchically composed in terms of events at a finer level of atomicity. This work also helps to model concurrency in the execution more clearly than before by examining events in views at different levels of atomicity, and their corresponding ordering relations.

We define system executions for various levels of atomicity by first defining a system execution dealing with the most elementary events, suitably identified. We then hierarchically compose system executions at coarser levels of atomicity by using the system executions at finer levels of atomicity.

Specifically, we choose to identify the most elementary events as certain "basic" communication actions [14, 15] at both processes and communication channels [3, 10, 31] in a distributed system execution. This level of atomicity is useful for designing complex communication constructs [2, 4, 13, 43] and for comparing their flexibility with that of the primitive communication events at this lowest level of atomicity as a benchmark. Another use of this level of atomicity is in designing systems that are asynchronous at the application layer but are synchronous at the transport layer or between output and input buffers of processes. A specific example of this design is that of a lightweight nonblocking implementation of causal message ordering [13, 28, 34, 37]. Events at this level of atomicity can then be composed together to define events at a coarser level in the atomicity hierarchy. At this second level in the hierarchy, the events are abstract send and receive events executed at the processes. Modeling events at this level of atomicity has implicitly been done by many applications such as distributed snapshots [8], modeling distributed computations [18, 29, 33], concurrency measures [11, 12, 19], transfer of knowledge [9], leader election and mutual exclusion [41]. The next coarser level in the atomicity hierarchy has events that are reactive in nature, i.e., each event denotes activity at a process in response to messages received from other processes. Modeling events at this level of the hierarchy has been used for distributed debugging [16, 35] and distributed termination detection [32, 45]. Another level of the hierarchy has events such that each event is affected by and affects the rest of the computation only by the process states at the start and end of the event, respectively. In this view of the computation, the global state of the system before and after any event is a transitless global state, i.e., there are no messages in transit between any pair of processes in this state. Applications such as checkpointing and recovery [44] atomic transactions [6], and fault-tolerant computations [20, 36] use this level of atomicity, also considered in [1]. Transitless states also occur when strong stable properties [38] like deadlock [27] and termination [32] become true in the system. We also show the use of transitless states in resetting vector clocks [18, 33] which are widely used in distributed computations. We consider the above four levels of atomicity in a unifying framework using hierarchical composition and lattices. Although we consider four important levels of atomicity for distributed system executions in this paper, the methodology used is

useful to model other levels of atomicity for distributed system executions, as well as views of parallel system executions.

Besides proposing a unifying framework for expressing and analyzing events at various levels of atomicity and presenting the applications of each level of atomicity, we also prove that each level of atomicity formulated using the hierarchical framework satisfies three properties. (**Property P1**) The events at any level of atomicity partition the events at the finer level of atomicity in terms of which this level is defined. (See Definition 5 and Theorems 3, 6, and 9 for the four levels of atomicity considered.) (**Property P2**) The events at any level of atomicity ordered by the corresponding ordering relation form a strict partially ordered set, henceforth, also referred to simply as a poset. (See Definition 5 and Theorems 4, 7, and 10 for the four levels of atomicity considered.) P1 implies that all the events at any level of atomicity are included implicitly in more abstract events at coarser levels of atomicity. Any result based on these properties that applies to any one level of atomicity applies to all levels of atomicity.

At the finest level of atomicity, the set of all observable states of the system forms a lattice. We show that each level of atomicity also satisfies the following property. (**Property P3**) At any level of atomicity, the set of all observable states forms a sublattice of the lattice of all observable states at the finer level of atomicity in terms of which this coarser level of atomicity is defined. Thus, corresponding to the hierarchical composition of system executions, we also express the hierarchical sets of observable global states in terms of embedded lattices. (See Theorems 2, 5, 8, and 11 for the four levels of atomicity considered.) We also show that the event ordering relation at each level of atomicity captures a notion of causality that is meaningful for that level of atomicity.

Section 2 presents the system model and the hierarchical framework used to model the events at various levels of atomicity. Section 3 presents the events at four levels of atomicity by a hierarchical composition of the events at a finer level of atomicity. For each level of atomicity, the section also discusses the applications. Section 4 discusses the uses of the presented methodology and concludes.

2. System model

A distributed system contains a set of processes running on processors connected by communication channels. Without loss of generality, we assume a single process runs on each processor. Any two processes can communicate by point-to-point message passing over a channel that connects the two processes. Every message can be uniquely identified at a given level of atomicity. A channel is a passive entity in the sense that it receives messages from a sender process and delivers them to the receiver process; we assume it cannot generate, consume, or alter messages, but it can permute the order of delivery of messages. The transfer of messages between a pair of processes takes finite time on a global time scale; the transfer of messages between a sender process and a channel, as well as between a channel and a receiver process is instantaneous.

Depending on the level of atomicity being modeled, both processes and channels are modeled, or only processes are modeled in the system. Processes and channels will also be referred to as *nodes* in the system. In the literature, the channel has been modeled as a node in various ways such as a shared memory accessible to the sender and receiver processes [3], as a process [10], or as a bag of messages [5]. The instantaneous communication between a sender process and a channel, as well as between a channel and a receiver process can also be modeled by an I/O automaton [31]. A channel that transfers messages from process node i to process node j is denoted c_{ij}.

Let E be the set of the most elementary events in a system execution, i.e., a computation. We assign a semantic meaning to E later. Events of E are partitioned into local computations at a process or channel, assuming that each event of E occurs at a single process or channel only. The local computation at process i or at channel i is denoted E_i. Each local computation is a linearly ordered set. An event e in E_i is denoted e_i. The initial event in E_i is $\perp_i$. For finite computations, the final event in each E_i is $\top_i$.

Each process or channel node is associated with a state. A state transition at a node occurs when an event occurs at the node. The *system state* or *global state* is the collection of the local states of the nodes. A global state transition occurs each time a local state transition occurs.

The local action of sending (receiving) a message is a *send* (*receive*) event. For simplicity, the model deals with send events that unicast messages point-to-point, although it can be modified to allow a send event to multicast a message point-to-multipoint. The set of events that occur at any one node in a run of a computation can be decomposed into the sets $\mathcal{RC}$, $\mathcal{SD}$, and $\mathcal{IN}$, which are the sets of events of receiving a message from another node, sending a message to another node, and internal events, respectively. Individual events in the three sets are denoted by RC, SD, and IN, respectively. The sets $\mathcal{RC}$, $\mathcal{SD}$, and $\mathcal{IN}$ will be defined at multiple levels of atomicity which will be differentiated by appropriate subscripts.

When channel nodes are explicitly modeled in a view of a computation, their send and receive events are as follows. An event at which c_{ij} receives a message from process i is a RC event on c_{ij}. An event at which c_{ij} sends a message to process j is a SD event on c_{ij}.

Events in a computation are ordered by the *causality relation* $\prec$ on E [29]. The causality relation is the smallest transitive relation satisfying the following two conditions: (i) If e_i occurs immediately before e_i' at node i, then $e_i \prec e_i'$. (ii) If e_i is the sending of a message and e_j is the receipt of the same message, then $e_i \prec e_j$. $(E, \prec)$ is a strict partial order.

A *cut* C of $(E, \prec)$ is a subset of E such that if $e_i \in C$ then $\forall e_i'$: $e_i' \prec e_i$, we have $e_i' \in C$. Thus, a cut is left-closed when it is projected on individual nodes. Cuts that preserve the causality relation in the computation are termed consistent cuts and are of interest because they are prefixes of the computation and denote feasible computations. Formally, a *consistent cut* of $(E, \prec)$ is a left-closed subset of E, i.e., CC is a consistent cut of $(E, \prec)$ iff $CC \subseteq E \wedge (e \in CC \wedge e' \prec e \Rightarrow e' \in CC)$. The system state

after the execution of a consistent cut is termed a *consistent global state*. The set of (correctly) observable system states is the set of consistent global states.

We use the relation symbol "$\sqsubset$" to denote the subset "$\subset$" relation between cuts of a computation. The following is a known result (see [17, 33]).

Theorem 1. (1) $\mathscr{C}$, *the set of all cuts of a poset* $(E, \prec)$, *forms a lattice* $(\mathscr{C}, \sqsubset)$ *with the operations* $\cup$ *and* $\cap$.
(2) $\mathscr{CC}$, *the set of all consistent cuts of a poset* $(E, \prec)$, *forms a sublattice* $(\mathscr{CC}, \sqsubset)$ *of* $(\mathscr{C}, \sqsubset)$.

The height of the lattice $(\mathscr{CC}, \sqsubset)$ is the number of events in E. Henceforth, we deal only with lattices of consistent cuts as they capture only all the feasible computations. The lattice of consistent global states is isomorphic to the lattice of consistent cuts.

2.1. Unifying framework

We present a hierarchical framework to define events at various levels of atomicity in terms of elementary actions by adapting the formalism of hierarchical views of a system execution introduced by Lamport [30]. The choice of actions treated as elementary is based on the need to model sufficiently fine-grained actions for the known applications. Therefore, this choice is arbitrary to some extent.

The set of events in the system execution at an arbitrary level of atomicity x, as well as the ordering relation among the events at that level of atomicity is represented as a tuple $\langle \mathscr{A}_x, \prec_x \rangle$. $\mathscr{A}_x$ and $\prec_x$ are different for each level of atomicity x. The term "atom" will be used interchangeably with "event"; individual events (or atoms) and the set of events (or atoms) are denoted A_x and $\mathscr{A}_x$, respectively, to emphasize their atomic nature. The subscript will be dropped when the context is clear.

Consider $(\mathscr{A}_\gamma, \prec_\gamma)$ and $(\mathscr{A}_\beta, \prec_\beta)$, where $\mathscr{A}_\gamma$ and $\mathscr{A}_\beta$ are sets and $\prec_\gamma$ and $\prec_\beta$ are relations on the elements of $\mathscr{A}_\gamma$ and $\mathscr{A}_\beta$, respectively. Let mapping μ_β be a one-many surjective mapping that maps each element A_β of $\mathscr{A}_\beta$ to a non-empty subset of $\mathscr{A}_\gamma$. If μ_β^{-1} is a function then $\mathscr{A}_\beta$ defines a partition on $\mathscr{A}_\gamma$ – thus each element A_γ of $\mathscr{A}_\gamma$ is contained in exactly one element A_β of $\mathscr{A}_\beta$, and an element A_β may contain multiple elements from $\mathscr{A}_\gamma$. Each element A_β in $\mathscr{A}_\beta$ is a set that is a higher-level grouping of the events in $\mathscr{A}_\gamma$ that is of interest to some application. μ_β is specified so as to define meaningful events at an appropriate level of atomicity $(\mathscr{A}_\beta, \prec_\beta)$ in terms of the events specified at the level of finer atomicity in $(\mathscr{A}_\gamma, \prec_\gamma)$. Moreover, μ_β is specified so as to define a meaningful ordering relation $\prec_\beta$ that captures some notion of causality appropriate for level of atomicity β. Specifically, $\prec_\beta(A_\beta, A'_\beta)$ is a function of $\prec_\gamma$ over $\mu_\beta(A_\beta) \times \mu_\beta(A'_\beta)$.

There are two cases to consider when we define a system execution $S_\beta = \langle \mathscr{A}_\beta, \prec_\beta \rangle$. (i) For system executions S_β at recursively higher levels of atomicity, we specify a mapping μ_β, which maps S_β to a system execution S_γ at a finer level of atomicity. $\mathscr{A}_\beta$ contains events at a coarser level of atomicity than $\mathscr{A}_\gamma$. (ii) If S_β is at the level of

atomicity of the most elementary actions that we choose, μ_β maps S_β to S_β and we provide a semantic model for S_β.

Definition 1. A system execution S_β is a tuple $\langle \mathscr{A}_\beta, \prec_\beta \rangle$ where $\mathscr{A}_\beta$ is a set and $\prec_\beta$ is an ordering relation on $\mathscr{A}_\beta$. S_β is specified in terms of a mapping $\mu_\beta : S_\beta \rightarrow S_\alpha$, where S_α is a system execution at a finer level of atomicity such that:

1. μ_β is a one-many surjective mapping that maps each element in $\mathscr{A}_\beta$ to a subset of $\mathscr{A}_\alpha$, and
2. μ_β defines $\prec_\beta$ in terms of $\prec_\alpha$, i.e., $\prec_\beta(A_\beta, A'_\beta)$ is a function of $\prec_\alpha$ over $\mu_\beta(A_\beta) \times \mu_\beta(A'_\beta)$.

If S_β is at the finest level of atomicity, $S_\alpha = S_\beta$ and we give a semantic model for S_β.

At the finest level of atomicity, we will use the semantic model of E and the causality relation on E, i.e., $\langle E, \prec \rangle$, for the system execution.

We define a *computation graph* for a computation $S_\beta = \langle \mathscr{A}_\beta, \prec_\beta \rangle$, where $\mu_\beta : S_\beta \rightarrow S_\alpha$, as follows. There is a vertex in the graph for every event A_β. There is a directed edge from vertex v to vertex v' if v denotes event A_β, v' denotes event A'_β, and $\prec_\beta(A_\beta, A'_\beta)$ is a direct (not induced by transitivity) dependency of the ordering relation $\prec_\beta$. Assuming $\prec_\beta$ captures a meaningful notion of causality and $(\mathscr{A}_\beta, \prec_\beta)$ is a partial order, the computation graph for S_β is a directed acyclic graph. Henceforth, a reference to "an edge at an event" will mean "an edge in the computation graph at a vertex in the computation graph representing an event". Note that a vertex in the computation graph represents an event in the computation, whereas a node is either a process or a channel at which the event occurs. Wherever possible, we differentiate between two types of edges in the graph. An edge that orders two events at the same system node is termed a *local edge*. An edge that orders two events at different system nodes is termed a *message edge* and represents a message.

The definitions of consistent cut and consistent global state apply to each system execution S_β. A consistent cut CC_β in S_β is a left-closed subset of $\mathscr{A}_\beta$. $\mathscr{CC}_\beta$ is the set of all consistent cuts CC_β. A consistent global state in S_β is the system state after the execution of a consistent cut in S_β. Once a global state is observed after the execution of a consistent cut, any subsequent global state observed is the state after the execution of a superset of the consistent cut executed earlier. Two computations at a given level of atomicity are equivalent if their sets of events are the same and the ordering relation between the events is the same, even though the absolute order of concurrent events, i.e., events that cannot be ordered by the ordering relation, may be different in global time. Given any computation S_β, the sequence of global states observed by a global observer in this or in any equivalent computation can be traced by a maximal chain in $(\mathscr{CC}_\beta, \sqsubset)$.

Given two consistent cuts CC_β and CC'_β such that $CC_\beta \subset CC'_\beta$, the system state after CC_β can be changed to the system state after CC'_β by the execution of events in $CC'_\beta \backslash CC_\beta$ along a maximal chain from CC_β to CC'_β. If CC_β and CC'_β are incomparable and the currently observed state is after the execution of CC_β, the observed system

state can be changed from the state after CC_β to the state after $CC_\beta \cup CC'_\beta$ by the execution of events in $CC'_\beta \backslash CC_\beta$ along a maximal chain from CC_β to $CC_\beta \cup CC'_\beta$.

3. Modeling events in a distributed computation

In Sections 3.1, 3.2, 3.3, and 3.4, we define system executions at four levels of atomicity S_{dist} (view of elementary events in a distributed system execution), S_{SR} (view of send and receive events), S_{react} (view of reactive events), and S_{TL} (view of events between transitless system states), respectively, in a hierarchical manner, starting with the finest level S_{dist} to which we assign the semantic model of $\langle E, \prec \rangle$. Note that the choice of the level of atomicity that we use for the finest level of atomicity is arbitrary to some extent; it is adequate for the four levels of atomicity we consider in this paper. However, one could, for example, choose the finer level of atomicity at which electronic signals are transmitted between the components of a processor as the finest level of atomicity to model events and activity at the firmware or assembly code execution level.

We define a coarser level of atomicity $S_\beta = \langle \mathscr{A}_\beta, \prec_\beta \rangle$ in terms of a finer level of atomicity S_α. For each level of atomicity, we show the following properties using the hierarchical framework: (P1) Atoms of $\mathscr{A}_\alpha$ are partitioned into atoms in S_β, i.e., each atom A_α of $\mathscr{A}_\alpha$ is contained in exactly one atom A_β of $\mathscr{A}_\beta$, and an atom A_β may contain multiple A_α atoms. (P2) The atoms in $\mathscr{A}_\beta$ form a poset when ordered by $\prec_\beta$. From P2 and Theorem 1, it follows that $\mathscr{C}\mathscr{C}_\beta$, the set of all observable global states (consistent cuts) in S_β, forms a lattice $(\mathscr{C}\mathscr{C}_\beta, \sqsubset)$. (P3) For each level of atomicity S_β, the lattice $(\mathscr{C}\mathscr{C}_\beta, \sqsubset)$ is a sublattice of $(\mathscr{C}\mathscr{C}_\alpha, \sqsubset)$. We then identify the applications of each level of atomicity. Section 4 will discuss the significance and uses of our result that all levels of atomicity possess the properties P1, P2 and P3.

3.1. Primitive send and receive events

To view the system execution at the finest level of atomicity S_{dist}, we consider primitive send and receive events that are expressed by explicitly modeling channels that connect any two processes, and the input and output buffers of the two processes. Though there are many communication constructs to send and receive messages, e.g., [2, 4, 43], they are not necessarily atomic. It is shown in [14, 15] that all such constructs can be expressed as some combination of one of the following primitive events at process nodes.[2]

1. POST-SEND, abbreviated PS, is a send event that initiates a message send to the destination process, and can complete even before the message is copied out of the sender's buffer. The set of all PS events is $\mathscr{PS}$.

[2] These events are executed by specifying options such as the buffer size, message-id, and the size of data to be received. Details of such options are not described here.

2. WAIT-FOR-BUFFER-RELEASE, abbreviated WB, waits for the message to be copied out of the sender's buffer. Thus, it is a receive event at which it receives an acknowledgement from the channel that the message has been received by the channel. The set of all WB events is $\mathcal{WB}$.
3. WAIT-FOR-SEND-TO-BE-MATCHED, abbreviated WSM, is a receive event that waits for an acknowledgement from the channel that the destination process has received the message. The set of all WSM events is $\mathcal{WSM}$.
4. POST-RECEIVE, abbreviated PR, is a send event that requests the channel to deliver to the process any incoming message that matches the parameters and the sender-id specified. This event can complete before the message to be received is stored in the receive buffer specified. The set of all PR events is $\mathcal{PR}$.
5. WAIT-FOR-RECEIVE-TO-BE-MATCHED, abbreviated WRM, is a receive event that completes only after the incoming message has been placed in the specified receive buffer. The set of all WRM events is $\mathcal{WRM}$.

Blocking and nonblocking as well as synchronous and asynchronous send operations, and blocking and nonblocking synchronous receive operations[3] [2, 4, 13, 43] can be executed using the primitive events PS, WB, WSM, PR, and WRM at process nodes. Specifically, a blocking receive is executed by a PR event immediately followed by a WRM event, a blocking synchronous send is executed by a PS event immediately followed by a WSM event, a blocking asynchronous send is executed by a PS event immediately followed by a WB event. Nonblocking sends and receives are implemented by a two-phase operation, wherein the first phase consists of a PS or PR event, respectively, that also associates a *wait-object* with the operation. The second phase which is initiated any time afterwards consists of a *WAIT-FOR-COMPLETION* event [4] that waits for the list of wait-objects specified at the event to be posted. The wait-objects specified could be a collection of those associated with communication operations and those associated with noncommunication operations. Although the *WAIT-FOR-COMPLETION* event makes an operating system call, one of the following receive events is implicitly executed at the process node when the wait-object associated with the PS or PR event is posted. If the send operation initiated by the PS event is a nonblocking asynchronous (synchronous) send, the wait-object associated with the PS event is a *wait-for-WB object* (respectively, a *wait-for-WSM object*) and a WB (respectively, a WSM) event occurs when the wait-object gets posted. If the receive operation initiated by the PR event is nonblocking, the wait-object associated with the PR event is a *wait-for-WRM object* and a WRM event occurs when the wait-object gets posted. Therefore, for blocking and nonblocking calls, the send initiated by a PS event completes with a WB or WSM event, and the receive initiated by a PR event completes with a WRM event. If nonblocking send and receive operations exist in the computation, their component events in S_{dis} may be interleaved.

[3] Receive operations are always synchronous. Asynchronous receive operations do not make sense.

Based on the above observations, the complement, (abbreviated *comp*), of the *PS*, *WB*, *WSM*, *PR*, and *WRM* events at a process node in S_{dist} is defined to specify the relation between events at a process node that complement other events at the same process node.

Definition 2. *comp*(e) is defined as follows [14, 15]:

1. If e is a *PS* event, then *comp*(e) is the corresponding *WB* or *WSM* event, and vice versa.
2. If e is a *PR* event, then *comp*(e) is the corresponding *WRM* event, and vice versa.

The events in $\mathcal{PS}$, $\mathcal{WB}$, $\mathcal{WSM}$, $\mathcal{PR}$, and $\mathcal{WRM}$ occur at process nodes. In order to model activity at the channel nodes, as has been done in [3, 5, 10], we also need to model and identify events at channel nodes, by viewing each channel as an active node. For each *PS* and *PR* event (which are send events) at a process node, there exists a corresponding receive event at the channel node. For each *WB*, *WSM* and *WRM* event (which are receive events) at a process node, there exists a corresponding send event at the channel node. The following definition captures this involution relation.

Definition 3. If e is a *SD* or *RC* event, then *match*(e) is, respectively, the *RC* or *SD* event corresponding to the message that was sent or received at e.

match(e) exists and is unique. (The definition of *match*(e) can be extended to multicasts where the same message is sent to multiple receivers on multiple channels.) Internal events at the channel node are events such as those at which a pair of messages in the channel are reordered.

Fig. 1 is a *timing diagram* [29] that illustrates the events *PS*, *WB*, *WSM*, *PR*, and *WRM*, as well as Definition 3, by showing the message transfer from process i to process j on channel c_{ij}. In this diagram, time flows horizontally from left to right. (A timing diagram is simply a computation graph in which all local edges are in the same dimension representing the flow of time, whereas the other dimension represents system nodes.) The message send initiated by the *PS* event could complete by either the *WB* event or the *WSM* event; although both *WB* and *WSM* are shown in the figure, in practice one of them would be used.

$\mathcal{A}_{dist}$, the set of elementary events in S_{dist}, can now be defined using disjoint sets.

Definition 4. $\mathcal{A}_{dist} = \mathcal{PS} \cup \mathcal{WB} \cup \mathcal{WSM} \cup \mathcal{PR} \cup \mathcal{WRM} \cup \{match(PS):\ PS \in \mathcal{PS}\} \cup \{match(WB):\ WB \in \mathcal{WB}\} \cup \{match(WSM):\ WSM \in \mathcal{WSM}\} \cup \{match(PR):\ PR \in \mathcal{PR}\} \cup \{match(WRM):\ WRM \in \mathcal{WRM}\} \cup \mathcal{IE}$.

The following decomposition of $\mathcal{A}_{dist}$ shows how the set is partitioned orthogonally to the above into send events, receive events, and internal events:

- $\mathcal{SL}_{dist} = \mathcal{PS} \cup \mathcal{PR} \cup \{match(WB):\ WB \in \mathcal{WB}\} \cup \{match(WSM):\ WSM \in \mathcal{WSM}\} \cup \{match(WRM):\ WRM \in \mathcal{WRM}\}$.

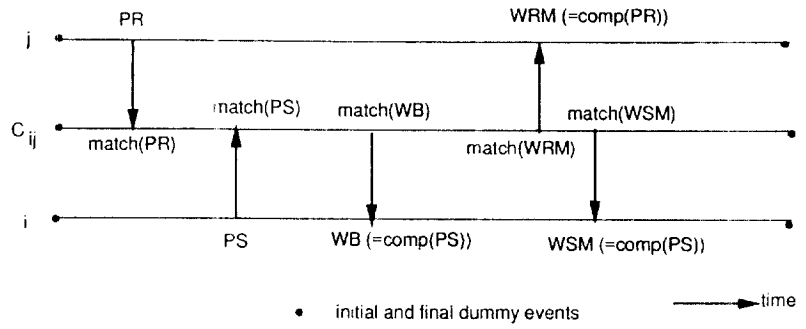

Fig. 1. Message communication events at the finest level of atomicity.

- $\mathscr{R}\mathscr{C}_{dist} = \mathscr{W}\mathscr{B} \cup \mathscr{W}\mathscr{S}\mathscr{M} \cup \mathscr{W}\mathscr{R}\mathscr{M} \cup \{match(PS): PS \in \mathscr{P}\mathscr{S}\} \cup \{match(PR): PR \in \mathscr{P}\mathscr{R}\}$.
- $\mathscr{A}_{dist} = \mathscr{A}$.

We can now define S_{dist}, the system execution at the finest level of atomicity, in terms of the semantic model of $(E, \prec)$.

Definition 5. System execution $S_{dist} = \langle \mathscr{A}_{dist}, \prec_{dist} \rangle$, where $\mu_{dist}(S_{dist} - S_{dist})$ is a 1–1 identity mapping. The semantic model of S_{dist} is $(E, \prec)$, where $\mathscr{A}_{dist}$ is E and $\prec_{dist}$ is the causality relation $\prec$ on $\mathscr{A}_{dist}$.

From Definition 5, it follows that S_{dist} satisfies – (Property P1) Atoms of $\mathscr{A}_{dist}$ partition events (atoms) in E, and (Property P2) $(\mathscr{A}_{dist}, \prec_{dist})$ is a poset.

Theorem 2 (Property P3). *$(\mathscr{C}\mathscr{C}_{dist}, \sqsubset)$, the lattice of consistent global states in S_{dist}, is embedded in the lattice of consistent global states $(\mathscr{C}\mathscr{C}, \sqsubset)$.*

Proof. From Definition 5, lattice $(\mathscr{C}\mathscr{C}_{dist}, \sqsubset)$ is isomorphic to $(\mathscr{C}\mathscr{C}, \sqsubset)$, the lattice of consistent global states in $(E, \prec)$. □

Any send or receive event e at a process node in S_{dist} identifies the set $\{e, comp(e), match(e), match(comp(e))\}$ that forms a single send or receive event at a higher level of abstraction. This is how events in S_{dist} are grouped together at a coarser level of granularity S_{SR}.

3.1.1. Applications

Complex communication constructs for specific communication styles, such as *remote procedure calls* (RPC) [2], *conversations* or *dialogs* [4], *messaging and queuing* constructs used in *sockets* applications, and message-passing for parallel systems [43], can be designed using *PS*, *WB*, *WSM*, *PR*, and *WRM* events of S_{dist}. Blocking and nonblocking, as well as synchronous and asynchronous send and receive operations

can be executed using specific combinations of the primitive events for diverse applications such as real-time industrial remote process control and monitoring, airlines reservations, and night-time reconciliation in banking. The primitive events of S_{dist} can provide a yardstick for evaluating the flexibility of network programming style permitted by complex communication constructs.

The S_{dist} view can be used to design nonblocking asynchronous programs that use blocking synchronous communication at the transport layer. The nonblocking asynchronous program can be written with greater concurrency while retaining the simplicity of reasoning and ensuing low implementation cost, offered by the synchronous blocking style. Consider the enforcement of *causal message ordering* which ensures that if the messages sent at two send events in the S_{SR} view of the computation have a common destination and the send events are ordered by causality, then the messages are delivered in the same order [13, 28, 37]. If the application sends messages without using synchronous blocking sends, then messages must piggyback information about messages sent earlier to enforce causal ordering; otherwise, there is no such overhead. A lightweight nonblocking asynchronous implementation for causal message ordering with no such overhead is designed using the S_{dist} view as follows [34]. Each process has a FIFO output buffer and a FIFO input buffer. The application process executes asynchronous nonblocking sends to its output buffer. The output buffer process executes PS and then WSM to the channel to execute a blocking synchronous send to the receiver's input buffer process before executing another PS and WSM pair for the subsequent send operation. The synchronous communication between the sender's output buffer and the receiver's input buffer is performed by a transport layer acknowledgement. The input buffer process of the receiver process forwards the received messages to the receiver process. The application uses asynchronous nonblocking sends to send messages without any overhead; under the covers, its output buffer does blocking synchronous sends by exploiting transport layer acknowledgements.

3.2. Send and receive constructs

Complex message send and receive events that atomically execute high-level communication constructs, e.g., constructs for various flavors of RPC surveyed in [2], the Message-Passing Interface MPI calls [43], or the CPI-C Common Programming Interface for Communications constructs [4], provide a higher level of abstraction than the primitive send and receive events of S_{dist}. A system execution at this level of atomicity, denoted S_{SR}, will be defined in terms of system execution S_{dist}. Only process nodes are considered in the S_{SR} view. Global state and snapshot definition and computation [7, 8], concurrency measures for a system execution [11, 12, 19], clock systems for distributed computations [18, 29, 33], transfer of knowledge [9], checkpointing and recovery [42, 44], leader election, and mutual exclusion algorithms [41] deal with send and receive events in the S_{SR} view of the system execution.

From Section 3.1, observe that any send or receive event e at a process node in S_{dist} identifies the set $\{e, comp(e), match(e), match(comp(e))\}$ that forms a send or

receive event at a higher level of abstraction – this set forms an atomic event in S_{SR}. Lemma 1 states that such an event in S_{SR} contains one, but not both, of a PS event and a WRM event from S_{dist}.

Lemma 1. *For any event $e \in (\mathcal{PS}_{dist} \cup \mathcal{WB}_{dist})$ at a process node, either a PS event belongs to the set $\{e, comp(e), match(e), match(comp(e))\}$ exclusive-or a WRM event belongs to the set $\{e, comp(e), match(e), match(comp(e))\}$.*

Proof. Follows from Definitions 2 and 3. □

For a send event $\{PS, comp(PS), match(PS), \text{match}(comp(PS))\}$ in the S_{SR} view, observe that the data being sent is beyond the control of the sender process once the PS event is executed. Thus, the "sending" occurs at the PS event. For a receive event $\{PR, comp(PR), match(PR), match(comp(PR))\}$ in the S_{SR} view, observe that at the receiver process, data is actually received at and can be used only after the $comp(PR) = WRM$ event is executed. Thus, the "receiving" occurs at the WRM event. The definition (part of Definition 6) of the *key_member* function on a send and receive event in the S_{SR} view formalizes the notion that in the S_{dist} view, the PS and WRM events, respectively, are their *key_member* events.

Definition 6. System execution $S_{SR} = \langle \mathcal{A}_{SR}, \prec_{SR} \rangle$ is defined by a mapping $\mu_{SR} : S_{SR} \rightarrow S_{dist}$ as follows:

1. $\mathcal{A}_{SR} = \mathcal{A}_{dist} \cup \{\{e, match(e), comp(e), match(comp(e))\} : e \in (\mathcal{PS} \cup \mathcal{WB})\}$.
2. For any $A_{SR} \in \mathcal{A}_{SR}$, define $key\_member(A_{SR})$ as follows:
   - $key\_member(A_{SR}) \stackrel{\text{def}}{=}$ a PS event in A_{SR}, if a PS event belongs to A_{SR},
   - $key\_member(A_{SR}) \stackrel{\text{def}}{=}$ a WRM event in A_{SR}, if a WRM event belongs to A_{SR},
   - $key\_member(A_{SR}) \stackrel{\text{def}}{=}$ a IN event in IN_{SR}, if a IN_{dist} event belongs to A_{SR}.

   Then, $A_{SR} \prec_{SR} A'_{SR}$ iff $key\_member(A_{SR}) \prec_{dist} key\_member(A'_{SR})$.

Definition 6 first defines the events in $\mathcal{A}_{SR}$, then defines the mapping *key_member* on events in $\mathcal{A}_{SR}$. $key\_member(A_{SR})$ identifies a PS, WRM, or IN event from $\mathcal{A}_{dist}$ that belongs to A_{SR}, and is useful to define $\prec_{SR}$ as well as to prove Theorem 4. Observe that Definition 6 can be modified for a system model that allows multicast sends, in which case there are multiple events of the types $match(PS)$, $comp(PS)$, and $match(comp(PS))$, one for each channel corresponding to each destination of the multicast at PS. We now show that S_{SR} satisfies properties P1, P2, and P3 in Theorems 3, 4, and 5, respectively.

Lemma 2. *Each atom in S_{dist} belongs to some atom in $\mathcal{A}_{SR}$.*

Proof. By Definition 6, for each PS event in $\mathcal{A}_{dist}$, the set $\{PS, match(PS), comp(PS), match(comp(PS))\} \in \mathcal{A}_{SR}$. Thus, for each PS event in $\mathcal{A}_{dist}$, its complementing event WB or WSM, as well as the events $match(PS)$ and either $match(WB)$

or $match(WSM)$ belong to such a set which is an event in $\mathcal{A}_{SR}$. There do not exist any other WB, WSM, $match(WB)$, or $match(WSM)$ events without the corresponding PS event.

By Definition 6, for each WRM event in $\mathcal{A}_{dist}$, the set $\{WRM, match(WRM), comp(WRM), match(comp(WRM))\} \in \mathcal{A}_{SR}$. Thus, for each WRM event in $\mathcal{A}_{dist}$, its complementing event PR, as well as the events $match(PR)$ and $match(WRM)$ belong to such a set which is an event in $\mathcal{A}_{SR}$. There do not exist any other PR, $match(PR)$, or $match(WRM)$ events without the corresponding WRM event.

From Definition 6, each internal event in $\mathcal{A}_{dist}$ is an event in $\mathcal{A}_{SR}$.

The lemma follows. □

The following lemma states that each event A_{SR} in $\mathcal{A}_{SR}$ has a uniquely defined $key\_member(A_{SR})$ which is a PS, WRM, or IN event of $\mathcal{A}_{dist}$.

Lemma 3. $\forall A_{SR} \in \mathcal{A}_{SR}$, $(key\_member(A_{SR}) \in \mathcal{PS})$ *exclusive-or* $(key\_member(A_{SR}) \in \mathcal{WRM})$ *exclusive-or* $(key\_member(A_{SR}) \in \mathcal{IN}_{dist})$.

Proof. Follows from Definition 6 and Lemma 1. □

Lemma 4. *The intersection of any two distinct atoms in S_{SR} is the empty set.*

Proof. Define the following sets which are disjoint by Lemma 3.

- $\mathcal{SS}_{SR} = \{A_{SR} \in \mathcal{A}_{SR}: key\_member(A_{SR}) \in \mathcal{PS}\}$,
- $\mathcal{RG}_{SR} = \{A_{SR} \in \mathcal{A}_{SR}: key\_member(A_{SR}) \in \mathcal{WRM}\}$,
- $\mathcal{IN}_{SR} = \{A_{SR} \in \mathcal{A}_{SR}: key\_member(A_{SR}) \in \mathcal{IN}_{dist}\}$.

If $X \in \mathcal{SS}_{SR}$, $Y \in \mathcal{RG}_{SR}$, then $X \cap Y = \emptyset$ because (i) $key\_member(X) \in \mathcal{PS}$ which is different from $key\_member(Y) \in \mathcal{WRM}$, and (ii) for any PS and WRM events, $\{PS, comp(PS), match(PS), match(comp(PS))\} \cap \{WRM, comp(WRM), match(WRM), match(comp(WRM))\} = \emptyset$ (see Definitions 2 and 3).

If $X \in \mathcal{SS}_{SR} \cup \mathcal{RG}_{SR}$, and $Y \in \mathcal{IN}_{SR}$, then $X \cap Y = \emptyset$ because (i) Y is an event $IN_{dist} \in \mathcal{IN}_{dist}$, and we have that (ii) event IN_{dist} cannot belong to X.

If both X and Y, which are distinct, belong to one of $\mathcal{SS}_{SR}$ or $\mathcal{RG}_{SR}$, then $X \cap Y = \emptyset$ in both the following cases: (i) If X and Y belong to $\mathcal{SS}_{SR}$, let $key\_member(X) = PS^1$, $key\_member(Y) = PS^2$. Then $\{PS^1, comp(PS^1), match(PS^1), match(comp(PS^1))\} \cap \{PS^2, comp(PS^2), match(PS^2), match(comp(PS^2))\} = \emptyset$. (see Definitions 2 and 3). (ii) If X and Y belong to $\mathcal{RG}_{SR}$, let $key\_member(X) = WRM^1$, $key\_member(Y) = WRM^2$. Then $\{WRM^1, comp(WRM^1), match(WRM^1), match(comp(WRM^1))\} \cap \{WRM^2, comp(WRM^2), match(WRM^2), match(comp(WRM^2))\} = \emptyset$. (see Definitions 2 and 3).

If both X and Y belong to $\mathcal{IN}_{SR}$, then $X \cap Y = \emptyset$ because X and Y are distinct elements in $\mathcal{IN}_{dist}$.

The lemma follows. □

The proof of Lemma 4 also shows that $\mathcal{A}_{SR}$ can be partitioned into $\mathcal{SS}_{SR}$, $\mathcal{RG}_{SR}$, and $\mathcal{IN}_{SR}$.

Theorem 3 (Property P1). *The atoms of $\mathcal{A}_{dist}$ are partitioned into atoms in S_{SR}.*

Proof. Follows from Lemmas 2 and 4. □

Theorem 4 (Property P2). *The atoms in $\mathcal{A}_{SR}$ ordered by $\prec_{SR}$ form poset $(\mathcal{A}_{SR}, \prec_{SR})$.*

Proof. Project the partial order $(E, \prec) = (\mathcal{A}_{dist}, \prec_{dist})$ to the events in PS, WRM, and WB_{dist} to get the partial order $(E', \prec)$. From Definition 6 and Lemmas 3 and 4, it follows that there is an one–one correspondence between the elements in E' and the events in $\{key\_member(A_{SR}): A_{SR} \in \mathcal{A}_{SR}\}$; also the relation $\prec_{SR}$ on $\mathcal{A}_{SR}$ is identical to the relation $\prec$ on E'. It follows that $(\mathcal{A}_{SR}, \prec_{SR})$ is isomorphic to the poset $(E', \prec)$ and is therefore a poset.

From Theorems 1 and 4, it follows that $(\mathcal{C}\mathcal{C}_{SR}, \sqsubset)$ forms a lattice, where $\mathcal{C}\mathcal{C}_{SR}$ is the set of all consistent cuts of poset $(\mathcal{A}_{SR}, \prec_{SR})$. We now show that $(\mathcal{C}\mathcal{C}_{SR}, \sqsubset)$ is a sublattice of $(\mathcal{C}\mathcal{C}_{dist}, \sqsubset)$.

The send and receive events in the S_{SR} view of a computation are linearly ordered at a process node; however, in the S_{dist} view of the process node, the component events of which these S_{SR} events are comprised may be interleaved if nonblocking sends and receives are permitted. Consider the following sequence of S_{SR} events at a process node, where the superscript of an event indicates its sequence number at that node:

$$\ldots SD^1, SD^2, RC^3, RC^4, \ldots$$

For any S_{SR} event in the above sequence, its component events in the S_{dist} view are assigned the same superscript. In the S_{dist} view of the events at the above process node, the component events of the above sequence could be interleaved as follows:

$$\ldots, PR^4, PS^1, PR^3, PS^2, WRM^3, WRM^4, WB^2, WSM^1, \ldots$$

In the S_{SR} view, the ordering of send and receive events is identified by the ordering of their *key_member* events in $\mathcal{A}_{dist}$ and the finer details of the S_{dist} view are lost in the abstraction of the S_{SR} view. Thus, even if $A_{SR} \prec_{SR} A'_{SR}$, it may be that $\exists A_{dist} \in A_{SR} \exists A'_{dist} \in A'_{SR} : A'_{dist} \prec_{dist} A_{dist}$. This implies that a left-closed subset of $\mathcal{A}_{SR}$ in the S_{SR} view may not be a left-closed subset of $\mathcal{A}_{dist}$ in the S_{dist} view. To overcome this drawback, we define procedure *Swap* to modify the S_{dist} view by reordering events within each node partition. When procedure *Swap* is invoked, S_{dist} is passed by value.

Swap(S_{dist}):

1. For each PS event in S_{dist} do
   (a) Repeatedly swap the event $comp(PS)$, if any, with each preceding event until the PS event is the immediate predecessor of the $comp(PS)$ event.
   (b) Repeatedly swap the event $match(comp(PS))$, if any, with each preceding event until the $match(PS)$ event is the immediate predecessor of the $match(comp(PS))$ event.
2. For each WRM event in S_{dist} do

(a) Repeatedly swap the event $comp(WRM)$, if any, with each succeeding event until the WRM event is the immediate successor of the $comp(WRM)$ event.
(b) Repeatedly swap the event $match(comp(WRM))$, if any, with each succeeding event until the $match(WRM)$ event is the immediate successor of the $match(comp(WRM))$ event.

Let S'_{dist} denote the transformed S_{dist}. The transformation of S_{dist} into S'_{dist} achieves the following.

- It preserves the S_{SR} view of the computation by the following reasoning. The set $\mathscr{A}_{SR}$ is unchanged. Also, the partial order on $\mathscr{A}_{SR}$ is preserved because (a) it is never the case that two events in $\mathscr{A}_{dist}$ that are $key\_members$ of two events in $\mathscr{A}_{SR}$ are swapped, and (b) the partial order of events in S_{SR} is defined by the partial order of their corresponding key_member events in the S_{dist} view.
- A left-closed subset of $\mathscr{A}_{SR}$ in the S_{SR} view is also a left-closed subset of $\mathscr{A}_{dist}$ in the S'_{dist} view because of Definition 6 and the fact that for each A_{SR}, $key\_member(A_{SR})$ and $comp(key\_member(A_{SR}))$ are adjacent events in S'_{dist}.

We use $(\mathscr{C}\mathscr{C}'_{dist}, \sqsubset)$, the lattice of consistent cuts in S'_{dist}, to show that $(\mathscr{C}\mathscr{C}_{SR}, \sqsubset)$ is a sublattice of $(\mathscr{C}\mathscr{C}_{dist}, \sqsubset)$.

Theorem 5 (Property P3). *$(\mathscr{C}\mathscr{C}_{SR}, \sqsubset)$, the lattice of consistent global states in S_{SR}, is embedded in the lattice of consistent global states $(\mathscr{C}\mathscr{C}_{dist}, \sqsubset)$.*

Proof. Apply transformation $Swap$ to S_{dist} to yield S'_{dist}. Both S_{dist} and S'_{dist} give an identical S_{SR} view of the computation. In fact, S_{dist} and S'_{dist} are identical if only blocking calls are allowed.

View each $C_{SR} \in \mathscr{C}\mathscr{C}_{SR}$ as the set C_{dist} in the S'_{dist} view of the computation. From Definition 6 and transformation $Swap(S_{dist})$, the set C_{dist} contains a left-closed set of events in S'_{dist}. Hence, $C_{dist} \in \mathscr{C}\mathscr{C}'_{dist}$.

For any C_{SR}, $C'_{SR} \in \mathscr{C}\mathscr{C}_{SR}$, consider the consistent global cuts $(C_{SR} \cap C'_{SR}) \in \mathscr{C}\mathscr{C}_{SR}$ and $(C_{SR} \cup C'_{SR}) \in \mathscr{C}\mathscr{C}_{SR}$. In the S'_{dist} view, these cuts are $(C_{dist} \cap C'_{dist})$ and $(C_{dist} \cup C'_{dist})$ which belong to $\mathscr{C}\mathscr{C}'_{dist}$ because this set of left-closed sets is closed under union and intersection operations. Thus, $(\mathscr{C}\mathscr{C}_{SR}, \sqsubset)$ is embedded in the lattice $(\mathscr{C}\mathscr{C}'_{dist}, \sqsubset)$.

We say that $(\mathscr{C}\mathscr{C}_{SR}, \sqsubset)$ is embedded in the lattice $(\mathscr{C}\mathscr{C}_{dist}, \sqsubset)$ in the sense that the ordering of events in S_{dist} and S'_{dist} is the same when only the events $key\_member(A_{SR})$, for $A_{SR} \in \mathscr{A}_{SR}$, are considered. □

3.2.1. Applications

Many applications such as the following explicitly model each complex send and receive construct, and internal event in the computation as a single event at process nodes in S_{SR}. At any process, knowledge is gained at receive events, transferred out at send events, and conserved at internal events, as modeled at the S_{SR} level of atomicity [9]. Global state recordings in a distributed system are done at the level of granularity of S_{SR} [7, 8]. Specifically, states of local process nodes are recorded after the execution

of a consistent cut and the in-transit messages between two process nodes in this state are computed as a function of the collective recorded states of the two process nodes. Concurrency measures for a system execution [11, 12, 19] deal with different metrics for the number of concurrent send and receive events in the computation at the S_{SR} level of atomicity. For leader election and mutual exclusion [41], exclusive access to the critical section is gained and relinquished by the send and receive events in S_{SR}, according to specific protocols, whereas internal events do not affect access rights to the critical section. Clock systems [18, 29, 33] are designed so that the local clock at a process ticks at a message send event, a message receive event, or an internal event in the S_{SR} view. In optimistic checkpointing and rollback recovery, it does not suffice that the failed process rollback and reexecute from a prior checkpoint [42, 44]. Other processes also have to be rolled back to undo the effects caused by the failed process having sent and received information to the other processes before failure. The extent of the rollback of other processes depends on the send and receive events at the failed process and at the other processes and it suffices to model sends and receives in the S_{SR} view.

3.3. Reactive events

A coarser atomicity of events than that of SD_{SR}, RC_{SR} or IN_{SR} events is useful for applications such as termination detection [32, 45] and debugging [16, 35], even though it does not reflect all the concurrency of the original execution. Events at this coarser level of atomicity are reactive because the computation in an event begins in reaction to a received message. Thus, a reactive event begins when a node receives an external message, and then it does local processing and may send messages. The reactive event is defined to end when either: (i) an application-dependent locally determinable condition ϕ becomes true at a distinguished auxiliary event $C(\phi)$, or (ii) just before a message is received after this event has sent a message, in the S_{SR} view of the execution. We define system execution S_{react} in terms of system execution S_{SR} using regular expressions over SD_{SR}, RC_{SR} and IN_{SR} events, and the auxiliary event $C(\phi)$.

Definition 7. System execution $S_{react} = \langle \mathscr{A}_{react}, \prec_{react} \rangle$ is defined by a mapping μ_{react} : $S_{react} \rightarrow S_{SR}$ as follows:

1. Reactive atoms at any node x form a sequence $\langle A^{x,1}_{react}, A^{x,2}_{react}, A^{x,3}_{react}, \ldots \rangle$ where:
   (a) $A^{x,1}_{react}$ = the maximal sequence of events that belong to $\mathscr{A}_{SR}$ and occur at node x, that satisfy the regular expression $\langle \perp_x (IN_{SR}|RC_{SR})^*(IN_{SR}|SD_{SR})^*(C(\phi))^* \rangle$,
   (b) $A^{x,i}_{react}, i > 1$ is the maximal nonempty sequence of events that belong to $\mathscr{A}_{SR}$ and occur at node x, that satisfy the context-sensitive regular expression $A^{x,i-1}_{react} A^{x,i}_{react} = A^{x,i-1}_{react} \langle RC_{SR}(IN_{SR}|RC_{SR})^*(IN_{SR}|SD_{SR})^*(C(\phi))^* \rangle$.
2. For $A_{react} \neq A'_{react}$, we have $A_{react} \prec_{react} A'_{react}$ iff $(\exists A_{SR} \in A_{react}, \exists A'_{SR} \in A'_{react}: A_{SR} \prec_{SR} A'_{SR})$.

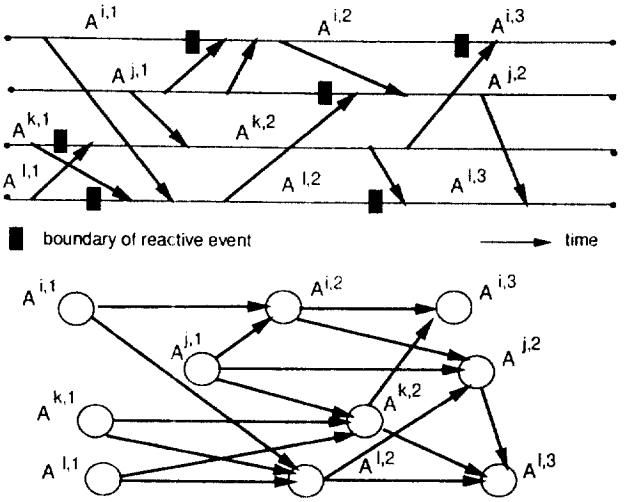

Fig. 2. Reactive events.

$A^{x,i}_{react}$ is the ith reactive event at node x. The superscripts/subscript are dropped if there is no ambiguity. Fig. 2 shows a timing diagram of the events in the S_{SR} view of a distributed computation, and the corresponding timing diagram of the events in the S_{react} view of the same computation.

We now show that S_{react} satisfies properties P1, P2, and P3 in Theorems 6, 7, and 8, respectively.

Theorem 6 (Property P1). *The atoms of $\mathcal{A}_{SR}$ are partitioned into atoms in S_{react}.*

Proof. It follows from Definition 7 that the intersection of any two events in $\mathcal{A}_{react}$ is the empty set and each event in $\mathcal{A}_{SR}$ belongs to some event in $\mathcal{A}_{react}$. Therefore, it follows that each SD_{SR}, RC_{SR}, and IN_{SR} event is assigned to one and only one event in $\mathcal{A}_{react}$. □

Theorem 7 (Property P2). *The atoms in $\mathcal{A}_{react}$ ordered by $\prec_{react}$ form poset $(\mathcal{A}_{react}, \prec_{react})$. (see [16] for a proof).*

Proof (*Sketch*). We sketch an alternate proof to that in [16]. Observe that each reactive event consists of two serial phases. The first phase is a receive phase in which messages are received but not sent. The second phase is a send phase in which messages are sent but not received. (For a reactive event that does not send messages, assume that a message is sent by the event to the next local reactive event.) Transitivity of $\prec_{react}$ follows from the definition and the above observation. We show its asymmetry by showing acyclicity in S_{react}. Consider the computation graph of $(\mathcal{A}_{react}, \prec_{react})$. The

proof uses induction on the length of the path in the computation graph between two reactive events. The induction hypothesis is that for any path in S_{react}, the send phase of the last reactive event begins after the receive phase of the first reactive event ends. Hence, any message sent in the last reactive event cannot be received in the first reactive event and the path is acyclic. □

It follows that no event in $\mathcal{A}_{react}$ has both an edge that goes to another event in $\mathcal{A}_{react}$ and an incoming edge from that other event. From Theorems 1 and 7, it follows that $(\mathcal{CC}_{react}, \sqsubset)$ forms a lattice.

Theorem 8 (Property P3). $(\mathcal{CC}_{react}, \sqsubset)$, *the lattice of consistent global states in* S_{react}, *is embedded in the lattice of consistent global states* $(\mathcal{CC}_{SR}, \sqsubset)$.

Proof. View each $C_{react} \in \mathcal{CC}_{react}$ as the set C_{SR} in the S_{SR} view of the computation. From Definition 7, the set C_{SR} is a left-closed set of events in S_{SR}. Hence, $C_{SR} \in \mathcal{CC}_{SR}$.

For any C_{react}, $C'_{react} \in \mathcal{CC}_{react}$, consider the consistent global cuts $(C_{react} \cap C'_{react}) \in \mathcal{CC}_{react}$ and $(C_{react} \cup C'_{react}) \in \mathcal{CC}_{react}$. In the S_{SR} view, these cuts are $(C_{SR} \cap C'_{SR})$ and $(C_{SR} \cup C'_{SR})$ which belong to $\mathcal{CC}_{SR}$ because this set of left-closed sets is closed under union and intersection operations. Thus, $(\mathcal{CC}_{react}, \sqsubset)$ is embedded in $(\mathcal{CC}_{SR}, \sqsubset)$. □

3.3.1. Applications

Computation termination [32, 45] can be modeled by reactive events as follows. Consider a system in which (i)–(iv) hold. (i) A process node is either idle or active. (ii) An idle process may have only a RC_{SR} event, at which time the process becomes active. (iii) An active process may have SD_{SR} events and RC_{SR} events. (iv) An active process can change state to idle at any time. A computation is *terminated* if each process is idle and the channels are empty. We express this as follows. Define ϕ as "there is no A_{SR} event waiting to occur". A process is idle if the reactive event has ended and presently there is no event waiting to occur, i.e., ϕ holds. A channel is empty if in the S_{SR} view, the number of send events at which a message was sent on that channel equals the number of receive events at which a message was received on that channel.

Reactive events are useful for debugging based on controlled reexecutions, as follows. A message race occurs at an RC_{SR} event if one of multiple messages can be received at the event. Debugging based on controlled reexecution of message races examines all the possible executions corresponding to one space-time diagram [16, 35]. Reasoning with the S_{react} view of the execution indicates how each RC_{SR} event can be presented with the maximum possible set of racing messages in the controlled re-executions. The definition of reactive events (Definition 7) for debugging does not use any auxiliary event $C(\phi)$, i.e., $\phi = false$. Observe that a message that could be received in a reactive event A may have been sent in a reactive event A' such that $A' \prec_{react} A \vee (A' \not\prec_{react} A \wedge A \not\prec_{react} A')$. For example, in Fig. 2, if A is $A^{(2)}$, then A' is

any of $A^{i,1}$, $A^{i,2}$, $A^{i,3}$, $A^{i,4}$, $A^{k,1}$, $A^{k,2}$, and $A^{k,4}$. To have controlled (replay) executions for event A, such events A' are first forced to complete before A begins so that the maximum possible number of messages race at event A. Controlled replay of A then generates all controlled executions of A by permuting the order of delivery of racing messages to RC_{SR} events in A.

3.4. Events between transitless cuts

System executions at the next higher level of atomicity S_{TL} are defined in terms of S_{SR}. Events at this level of atomicity occur at multiple process nodes and have applications such as resetting vector clocks, checkpointing and rollback recovery [6, 20, 44], synchronization [36], atomic transactions [6, 20], and fault tolerance [36].

Definition 8. A transitless cut TLC_{SR} is a consistent cut of $(\mathcal{A}_{SR}, \prec_{SR})$ such that the global state after the execution of TLC_{SR} has no messages in transit, i.e., in the computation graph of S_{SR}, the only ordering edges between TLC_{SR} and $\mathcal{A}_{SR}\backslash TLC_{SR}$ are local edges denoting direct local dependencies of $\prec_{SR}$ at process nodes (defined in Section 2.1).

The system state after the execution of events in a transitless cut is a *transitless global state*. Transitless global states have the property that the effects of the past computation are contained in only local edges of the computation graph S_{SR} (denoting direct local dependencies of $\prec_{SR}$ at process nodes), viz., the effects of the past computation are contained in the process states, because no messages are in transit. A transitless global state is like the *initial state* or the *final state* in the sense that only the states of process nodes determine the outcome of the future computation (if any). A transitless state denotes a notion of quiescence because processing activity at process nodes is not dependent on any messages in transit anywhere in the system. We analyze events at this level of atomicity using Theorem 1 and properties of lattices [17, 33]. Recall that from Theorems 1 and 4, it follows that $(\mathcal{CC}_{SR}, \sqsubset)$ forms a lattice. Let $\mathcal{TL}\mathcal{C}_{SR}$ be the set of all transitless cuts TLC_{SR}. We now show that $(\mathcal{TL}\mathcal{C}_{SR}, \sqsubset)$ is a sublattice of $(\mathcal{CC}_{SR}, \sqsubset)$.

Lemma 5. *$\mathcal{TL}\mathcal{C}_{SR}$, the set of all transitless cuts of a poset $(\mathcal{A}_{SR}, \prec_{SR})$, forms a lattice $(\mathcal{TL}\mathcal{C}_{SR}, \sqsubset)$ which is a sublattice of $(\mathcal{CC}_{SR}, \sqsubset)$, with operations $\cup$ and $\cap$.*

Proof. From Definition 8, $\mathcal{TL}\mathcal{C}_{SR} \subseteq \mathcal{CC}_{SR}$, and $\forall C \in \mathcal{TL}\mathcal{C}_{SR}$, there is no message edge from an event in C to an event in $\mathcal{A}_{SR}\backslash C$. For any $C^1, C^2 \in \mathcal{TL}\mathcal{C}_{SR}$, we need to show that (I) there is no message edge from an event in $C^1 \cap C^2$ to an event in $\mathcal{A}_{SR}\backslash(C^1 \cap C^2)$, and (II) there is no message edge from an event in $C^1 \cup C^2$ to an event in $\mathcal{A}_{SR}\backslash(C^1 \cup C^2)$.

(I) Assume there exists an $e \in C^1 \cap C^2$ such that there is a message edge from e to an event $e' \in (\mathcal{A}_{SR}\backslash(C^1 \cap C^2))$. e' belongs to each of C^1 and C^2 because $e \in C^1$ and

$e \in C^2$, and both $C^1, C^2 \in \mathcal{TLC}_{SR}$. It follows that $e' \in (C^1 \cap C^2)$, contradicting the assumption.

(II) Proof is analogous to that of (I). □

From Lemma 5, note that each member of lattice $\mathcal{TLC}_{SR}$ is a set of events in $\mathcal{A}_{SR}$. Henceforth, a member of $\mathcal{TLC}_{SR}$ will be denoted by TLC. For any two comparable elements TLC^u and TLC^l of a lattice, $length[TLC^l, TLC^u]$ is the length of the longest maximal chain in the lattice between TLC^l and TLC^u. We now define the system execution S_{TL} for transitless cuts using the lattice $\mathcal{TLC}_{SR}$ and S_{SR}.

Definition 9. System execution $S_{TL} = \langle \mathcal{A}_{TL}, \prec_{TL} \rangle$ is defined by a mapping $\mu_{TL} : S_{TL} \rightarrow S_{SR}$ as follows:

1. $\mathcal{A}_{TL} = \{(TLC^u \backslash TLC^l) : TLC^u, TLC^l \in \mathcal{TLC}_{SR} \wedge length[TLC^l, TLC^u] = 1\}$.
2. $\prec_{TL}$ is the transitive closure of $\prec_{tlc}$, where for any two distinct atoms in $\mathcal{A}_{TL}$, we have $(TLC^u \backslash TLC^l) \prec_{tlc} (TLC'^u \backslash TLC'^l)$ iff $(\exists e \in (TLC^u \backslash TLC^l), \exists e' \in (TLC'^u \backslash TLC'^l) : e \prec_{SR} e')$.

Events in $\mathcal{A}_{TL}$ change the system state from one transitless state to another. Events in $\mathcal{A}_{TL}$ are defined only in terms of the set difference of two elements (of the form $TLC^u \backslash TLC^l$) of lattice $\mathcal{TLC}_{SR}$ that are separated by a *length* of one. The same event may be expressible as the difference of more than one pair of transitless cuts. If so, then for any two such pairs of transitless cuts, at least two of the four cuts are incomparable. This property is important and will be used in the proof of Theorem 10. Fig. 3 is a timing diagram of the events in the S_{SR} view of a distributed computation, and the corresponding events in the S_{TL} view of that computation. Each event in $\mathcal{A}_{TL}$ is marked by encircling the elements of $\mathcal{A}_{SR}$ to which μ_{TL} maps it. There is an initial dummy event, and a final dummy event for terminating computations. All the edges of $(\mathcal{A}_{SR}, \prec_{SR})$ entering and leaving each event A_{TL} in $\mathcal{A}_{TL}$ are local edges. An event A_{TL} signifies that the computation it represents is affected only by the incoming local edges in a S_{SR} view, viz., the states of the processes at the start of the event, and it affects the rest of the computation only through outgoing local edges in the S_{SR} view, viz., the states of the processes at the end of the event. (Observe that an event A_{TL} may contain multicasts, and it allows lost messages, provided there are no message edges that originate within the event and terminate outside it.)

Theorem 9 (Property P1). *The atoms of $\mathcal{A}_{SR}$ are partitioned into atoms in S_{TL}.*

Proof. We need to show (I) each event in $\mathcal{A}_{SR}$ belongs to some event in $\mathcal{A}_{TL}$ and (II) the intersection of any two distinct events in $\mathcal{A}_{TL}$ is the empty set. Let TLC denote a member of $\mathcal{TLC}_{SR}$.

(I) Consider any chain $TLC^{\perp} (= \emptyset) \sqsubset TLC^1 \sqsubset TLC^2 \ldots \sqsubset TLC^{\top} (= \mathcal{A}_{SR})$ in $\mathcal{TLC}_{SR}$ such that TLC^i is covered by TLC^{i+1}, where $\forall i \in [\perp, \top)$, $TLC^i \sqsubset TLC^{\top}$. Such a chain exists in each lattice and is a maximal chain. Hence, $\forall i \in [\perp, \top)$,

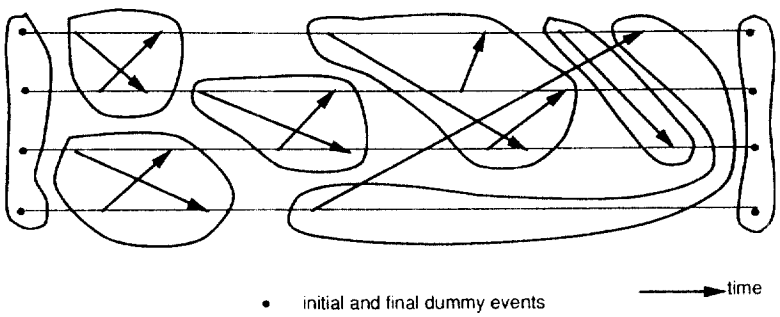

Fig. 3. Events between transitless global states.

$TLC^{i+1} \setminus TLC^i \in \mathscr{A}_{TL}$. $\forall A_{SR} \in \mathscr{A}_{SR}, \exists i : A_{SR} \in (TLC^{i-1} \setminus TLC^i)$. Hence, each event in $\mathscr{A}_{SR}$ belongs to some event in $\mathscr{A}_{TL}$.

(II) Consider $A_{TL} = TLC^{iu} \setminus TLC^{il}$, and $A'_{TL} = TLC^{ju} \setminus TLC^{jl}$, where $length[TLC^{il}, TLC^{iu}] = 1$ and $length[TLC^{jl}, TLC^{ju}] = 1$. We prove by contradiction.

Let us assume that $A_{TL} \cap A'_{TL} \neq \emptyset$ and A_{TL} and A'_{TL} are distinct. With this assumption, observe that

(1) $TLC^{il} \subseteq (TLC^{il} \cup TLC^{jl}) \cap TLC^{iu} \subset (TLC^{il} \cup TLC^{ju}) \cap TLC^{iu} \subseteq TLC^{iu}$.

(2) In (1), if $TLC^{il} = (TLC^{il} \cup TLC^{jl}) \cap TLC^{iu}$, then $TLC^{il} = (TLC^{il} \cup TLC^{jl}) \cap (TLC^{il} \cup A_{TL}) = [(TLC^{il} \cup TLC^{jl}) \cap TLC^{il}] \cup [(TLC^{il} \cup TLC^{jl}) \cap A_{TL}] = TLC^{il} \cup (TLC^{jl} \cap A_{TL})$. This implies $TLC^{jl} \cap A_{TL} \subseteq TLC^{il}$, implying $TLC^{jl} \cap A_{TL} = \emptyset$ because TLC^{il} and A_{TL} are disjoint.

(3) In (1), if $(TLC^{il} \cup TLC^{ju}) \cap TLC^{iu} = TLC^{iu}$, then we have $TLC^{iu} \subseteq TLC^{ju}$, implying $A_{TL} \subseteq TLC^{ju}$.

Observe that in (1), if both $TLC^{il} = (TLC^{il} \cup TLC^{jl}) \cap TLC^{iu}$ and $(TLC^{il} \cup TLC^{ju}) \cap TLC^{iu} = TLC^{iu}$ are true, then from (2) and (3), $A_{TL} \subseteq A'_{TL}$. However, we can always require that given A_{TL} and A'_{TL}, $A_{TL} \not\subseteq A'_{TL}$ (by swapping the two events at the start of this proof if $A_{TL} \subset A'_{TL}$). Hence, both $TLC^{il} = (TLC^{il} \cup TLC^{jl}) \cap TLC^{iu}$ and $(TLC^{il} \cup TLC^{ju}) \cap TLC^{iu} = TLC^{iu}$ cannot be true. But then it follows from (1) that $length[TLC^{il}, TLC^{iu}] > 1$, which violates the definition of A_{TL}. Therefore, the assumption that $A_{TL} \cap A'_{TL} \neq \emptyset$ cannot hold for distinct A_{TL} and A'_{TL}.

Theorem 10 (Property P2). *The atoms in $\mathscr{A}_{TL}$ ordered by $\prec_{TL}$ form poset $(\mathscr{A}_{TL}, \prec_{TL})$.*

Proof. Transitivity of $\prec_{TL}$ follows from Definition 9. We show that $\prec_{TL}$ is asymmetric. Let $A_{TL} \prec_{TL} A'_{TL}$. Recall from Definition 9 and the discussion following it that the same event in $\mathscr{A}_{TL}$ may be expressible as the difference of more than one pair of transitless cuts. Let $A_{TL} = TLC^{i,u} \setminus TLC^{i,l}$, where i enumerates each of the pairs whose difference is A_{TL}. Similarly define $A'_{TL} = TLC^{j,u} \setminus TLC^{j,l}$, where j enumerates each of the pairs whose difference is A'_{TL}. We have $\forall j, \exists i$, $TLC^{i,u} \sqsubset TLC^{j,l}$ because from Definition 9,

we can infer the existence of an interval $[TLC^{i,u}, TLC'^{j,l}]$. As $A_{TL} \subseteq TLC^{x,u}$, we now have (a) $A_{TL} \subseteq \bigcap_j TLC'^{j,l}$.

Now assume $A'_{TL} \prec_{TL} A_{TL}$. It follows that $\exists x \exists y$ such that $A_{TL} = TLC^{x,u} \setminus TLC^{x,l}$, $A'_{TL} = TLC'^{y,u} \setminus TLC'^{y,l}$ and $TLC'^{y,l} \sqsubset TLC^{x,l}$. Also, $A_{TL} \not\subseteq TLC'^{y,l}$, hence $A_{TL} \not\subseteq \bigcap_j TLC'^{j,l}$. But this contradicts $A_{TL} \subseteq \bigcap_j TLC'^{j,l}$, shown in (a) above. It follows that $A'_{TL} \not\prec_{TL} A_{TL}$. □

From Theorems 1 and 10, it follows that $(\mathcal{CC}_{TL}, \sqsubset)$ forms a lattice.

Theorem 11 (Property P3). *$(\mathcal{CC}_{TL}, \sqsubset)$, the lattice of consistent global states in S_{TL}, is embedded in the lattice of consistent global states $(\mathcal{CC}_{SR}, \sqsubset)$.*

Proof. From Lemma 5, $(\mathcal{TL}'\mathcal{C}_{SR}, \sqsubset)$ is a sublattice of $(\mathcal{CC}_{SR}, \sqsubset)$. We only need to show that $(\mathcal{CC}_{TL}, \sqsubset)$ is the lattice $(\mathcal{TL}'\mathcal{C}_{SR}, \sqsubset)$. This follows from Definition 9 and the observation that $\mathcal{CC}_{TL}$, the set of consistent cuts of S_{TL}, is exactly the set of transitless cuts in S_{SR}. □

Note that there can exist computations which have no transitless states other than the initial state and the final state. However, multiple other transitless states can exist in the computation. Some of these transitless states that naturally occur or are induced in the distributed computation are of special interest and are discussed next.

3.4.1. Applications

A transitless global state offers the convenience that it isolates and confines the effects of the past computation to only the states of process nodes, which determine the outcome of the future computation. Transitless states are therefore used in applications like fault tolerance [36], checkpointing/recovery [6, 20, 44], and transactions [6, 20], in which a past global state may need to be restored. Transitless states are created through synchronization at the cost of restricted interprocess communication to provide fault tolerance [36]. The transaction model of [20] uses a non-intrusive scheme of replicating parts of the database to create and record a transitless state. Transaction systems which require that all or none of the transaction's effects be made permanent in case of failure create transitless states at the end of each transaction using commit protocols [6]. In all these applications, transitless states during the computation are created at meaningful points and recorded. Note that transitless states after the execution of only certain events in S_{TL} are recorded, i.e., in the execution traced by any maximal chain in $(\mathcal{CC}_{TL}, \sqsubset)$, not every transitless state that exists after the execution of each event in $\mathcal{A}_{TL}$ is recorded; in case of failure, the most recent recorded transitless state is restored for recovery, fault-tolerance, or undoing the transaction.

Transitless states can also be shown to be useful to reset vector clocks [18, 33]. These clocks have the property that for $e, e' \in \mathcal{A}_{SR}$, $e \prec_{SR} e'$ iff $T(e) < T(e')$, where $T(e)$ denote the clock value at which event e occurred. When vector clocks at all the nodes are reset at a transitless state, wrong inferences about causality cannot be drawn

due to receipt of messages with high timestamp values sent before reset. Specifically, if a message sent before clock reset is received at e' after clock reset (thus, e' has a high timestamp), then another event e which is concurrent with e' may have a lower timestamp than that of e'. This violates the vector clock property that $T(e) < T(e')$ iff $e \prec_{SR} e'$.

A stable property in a distributed system is a global property which once true, remains true forever [38]. Transitless states occur in the computation, projected on either all or some process nodes, when certain stable properties associated with quiescence of message activity, such as distributed deadlock [27] or computation termination [32] become true in the system. Transitless states also occur when cooperating processes suspend execution after completing their subtasks to reach a *barrier synchronization* or a *rendezvous* [43] in the computation.

4. Discussion

Modeling events at different levels of atomicity in distributed system executions is crucial in modeling distributed activities to provide different abstract views, simplifying reasoning for the programmer and system designer, and studying questions about nonatomicity and concurrency. In the past, the events at different levels of atomicity had only been implicitly modeled in the isolated contexts of their applications. This paper presented a unifying framework for expressing and analyzing events at various levels of atomicity and showed that the various levels of atomicity are related to each other. In the framework, a system execution at a coarser level of atomicity is defined in terms of a system execution at a finer level of atomicity using hierarchical composition [30]; thus, events at any level of atomicity are composed of events at a finer degree of atomicity. The framework was applied to four levels of atomicity of events in distributed executions and we described the applications that have found it most useful to model system executions at each of the levels. In [23], we show how the framework can be applied to parallel system executions. It is straightforward to accommodate message losses and multicasts by varying the system model.

The system execution at every level of atomicity was shown to have three properties. (Property P1) If S_β is defined in terms of S_γ, then the atoms in S_α are partitioned into atoms in S_β. (Property P2) The atoms at any level of atomicity form a poset ordered by an ordering relation for that level of atomicity. Therefore, any result or proof that applies to one level of atomicity and is based on the above graph properties applies to all levels of atomicity. For example, the proof for execution S_{SR} that synchronous communication between application processes guarantees causal ordering of message unicasts applies without change to the proof for execution S_{dist} that asynchronous communication between application processes, with synchronous communication over channels between the (infinite) output and input process buffers, respectively, as described in Section 3.1.1, guarantees causal ordering of message unicasts [34]. A second example is the reuse of concurrency measures formulated for the S_{SR} view of a computation

[11, 12, 19]. These measures are based on properties of a partial order. From property P2, they apply to system executions at all levels of atomicity. For example, these measures can be used to gauge concurrency for incremental debugging in S_{react} and determine the number of nondeterministic and deterministic replays needed. A third example is the following. To increase concurrency, long transactions can be chopped into smaller pieces based on the principle "if the pieces of the transaction execute serially, then the transaction executes serially" [40]. Well-known results in serializability theory for the execution S_{SR} that considers read and write operations of a transaction can be applied to the pieces of a transaction for execution S_{TL}.

The paper also showed that each level of atomicity satisfied the following property. (Property P3) The global states at the various levels of atomicity defined using hierarchical composition correspond to embedded lattices of global states. This demonstrates how different abstract views of the same distributed computation can be provided to various applications, based on their need for information hiding. By choosing the appropriate level of abstraction, the application designer's task of designing and verifying the application is simplified.

Observe that the event ordering relation at each level of atomicity captures some notion of causality, meaningful for that level of atomicity. Although the event ordering relation for S_{SR} is accepted as the causality relation in most literature on distributed systems ([39] gives a good survey), it is not an absolute definition of causality. As shown in Section 3.2, it is possible that for two ordered events at the same process in S_{SR}, their component events in S_{dist} may be interleaved. Causality between distributed nonatomic events is studied in [22, 24–26]. [22, 24] examine the causality between a pair of nonatomic events where each nonatomic event is itself a collection of linearly ordered subevents. [22, 25, 26] examine the causality between a pair of nonatomic events where each nonatomic event is itself a collection of subevents that are partially ordered.

In summary, the formalism of this paper united various views of a distributed system execution at different levels of atomicity, provided a unifying way of modeling concurrency in the various views, and examined the applications of the various levels of atomicity considered. The paper also showed that the results of any level of atomicity that are based on the graph poset properties of the execution are applicable to all levels of atomicity. The formalism and methodology are applicable to other views of distributed system executions, as well as to views of parallel system executions.

Acknowledgements

The comments of the anonymous referees were very valuable in improving the clarity of this paper.

References

[1] M. Ahuja, A.D. Kshemkalyani, T. Carlson, A basic unit of computation in distributed systems, Proc. 10th IEEE Int. Conf. Distrib. Comput. Systems, 1990, pp. 12–19.

[2] A. Ananda, B. Tay, E. Koh, A survey of asynchronous remote procedure calls, ACM Operating Systems Rev. 26 (2) (1992) 92–109.

[3] E. Arjomandi, M.J. Fischer, N. Lynch, Efficiency of synchronous versus asynchronous distributed systems, J. ACM 30 (3) (1983) 448–456.

[4] W. Arnette, A.D. Kshemkalyani, W. Riley, J. Sanders, P. Schwaller, J. Terrien, J. Walker, CPI-C: An API for distributed applications, IBM Systems J. 34 (3) (1995) 501–518.

[5] J.A. Bergstra, J.W. Klop, Process theory based on bisimulation semantics, in: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency, Lecture Notes in Computer Science, vol. 354, Springer, Berlin, 1988, pp. 50–122.

[6] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.

[7] L. Bougé, Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP, Theoret. Comput. Sci. 49 (1987) 145–169.

[8] K.M. Chandy, L. Lamport, Distributed snapshots: global states of a distributed system, ACM Trans. Comput. Systems 3 (1) (1985) 63–75.

[9] K.M. Chandy, J. Misra, How processes learn, Distributed Comput. 1 (1986) 40–52.

[10] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[11] B. Charron-Bost, Combinatorics and geometry of consistent cuts: application to concurrency theory, in: J.-C. Bermond, M. Raynal (Eds.), Proc. Internat. Workshop on Distributed Algorithms, Lecture Notes in Computer Science, vol. 392, Springer, Berlin, 1989, pp. 45–56.

[12] B. Charron-Bost, Coupling coefficients of a distributed execution, Theoret. Comput. Sci. 110 (2) (1993) 341–376.

[13] B. Charron-Bost, F. Mattern, G. Tel, Synchronous, asynchronous and causally ordered communication, Distributed Comput. 9 (4) (1996) 173–191.

[14] R. Cypher, E. Leu, Repeatable and portable message-passing programs, Proc. 13th ACM Symp. on Principles of Distributed Computing, August 1994, pp. 22–31.

[15] R. Cypher, E. Leu, Semantics of blocking and nonblocking send and receive primitives, Proc. IEEE Int. Conf. Parallel Processing, August 1994, pp. 729–735.

[16] S. Damodaran-Kamal, J. Francioni, Nondeterminacy: testing and debugging in message-passing parallel programs, ACM SIGPLAN Notices 28 (12) (1993) 118–128.

[17] T. Donnellan, Lattice Theory, Pergamon Press, Oxford, 1968.

[18] C.A. Fidge, Timestamps in message-passing systems that preserve partial ordering, Australian Comput. Sci. Comm. 10 (1) (1988) 56–66.

[19] C.A. Fidge, A simple run-time concurrency measure, in: T. Bossomaier, T. Hintz, J. Hulskamp (Eds.), The Transputer in Australasia, IOS Press, Amsterdam, 1990, pp. 92–101.

[20] M. Fischer, N. Griffeth, N. Lynch, Global states in a distributed system, IEEE Trans. Software Eng. 8 (3) (1982) 198–202.

[21] W. Janssen, Layered design of parallel systems, Ph.D. Thesis, University of Twente, The Netherlands, September 1994.

[22] A.D. Kshemkalyani, Temporal interactions of intervals in distributed systems, Tech. Report TR 29.1933, IBM, September 1994.

[23] A.D. Kshemkalyani, A unifying framework for viewing atomic actions in parallel and distributed systems, Tech. Report TR29.2014, IBM, 1995.

[24] A.D. Kshemkalyani, Temporal interactions of intervals in distributed systems, J. Comput. System Sci. 52 (2) (1996) 287–298.

[25] A.D. Kshemkalyani, Synchronization for distributed real-time applications, Proc. 5th IEEE Workshop on Parallel and Distributed Real-time Systems, April 1997, pp. 81–90.

[26] A.D. Kshemkalyani, Relative timing constraints between complex events, Proc. 8th IASTED Conf. on Parallel and Distributed Computing and Systems, October 1996, pp. 324–326.

[27] A.D. Kshemkalyani, M. Singhal, On characterization and correctness of distributed deadlock detection, J. Parallel Distributed Comput. 22 (1) (1994) 44–59.

[28] A.D. Kshemkalyani, M. Singhal, Necessary and sufficient conditions on the information for causal message ordering and their optimal implementation, Distributed Computing, to appear (also available as Tech. Report TR 29.2040, IBM, July 1995).
[29] L. Lamport, Time, clocks, the ordering of events in a distributed system, Comm. ACM 21 (7) (1978) 558–565.
[30] L. Lamport, On interprocess communication, Part I: basic formalism, Part II: algorithms, Distributed Comput. 1 (1986) 77–101.
[31] N. Lynch, M. Tuttle, An introduction to input/output automata, Tech. Report MIT/LCS/TM-373, Massachussetts Institute of Technology, November 1988.
[32] F. Mattern, Algorithms for distributed termination detection, Distributed Comput. 2 (3) (1987) 161–175.
[33] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard et al. (Eds.), Proc. Workshop on Parallel and Distributed Algorithms, North-Holland, 1989, Amsterdam, pp. 215–226.
[34] F. Mattern, S. Fünfrocken, A nonblocking lightweight implementation of causal order message delivery, in: K. Birman, F. Mattern, A. Schiper (Eds.), Theory and Practice in Distributed Systems, Lecture Notes in Computer Science, vol. 938, Springer, Berlin, 1995, pp. 197–213.
[35] R. Netzer, B. Miller, Optimal tracing and replay for debugging message-passing parallel programs, Proc. Supercomputing, November 1992, pp. 502–511.
[36] B. Randell, System structure for software fault tolerance, IEEE Trans. Software Eng. 1(2) (1975) 220–232.
[37] M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement it, Inform. Processing Lett. 39 (1991) 343–350.
[38] A. Schiper, A. Sandoz, Strong stable properties in distributed systems, Distributed Comput. 8 (2) (1994) 93–103.
[39] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: in search of the holy grail, Distributed Comput. 7 (3) (1994) 149–174.
[40] D. Shasa, F. Lirbat, E. Simon, P. Valduriez, Transaction chopping: algorithms and performance studies, ACM Trans. Database Systems 22 (3) (1995) 325–363.
[41] M. Singhal, A taxonomy of distributed mutual exclusion, J. Parallel Distributed Comput. 18 (1) (1993) 94–101.
[42] M. Singhal, F. Mattern, An optimality proof of asynchronous recovery algorithms in distributed systems, Inform. Processing Lett. 55 (3) (1995) 468–475.
[43] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI: The Complete Reference, The MIT Press, Cambridge, MA, 1996.
[44] R.E. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Trans. Comput. Systems 3(3) (1985) 204–226.
[45] R.W. Topor, Termination detection for distributed computations, Inform. Processing Lett. 18 (1) (1984) 33–36.