A Framework for Viewing Atomic Events in Distributed Computations

Ajay D. Kshemkalyani

IBM Corporation, P. O. Box 12195, Research Triangle Park, NC 27709, USA
Email: ajayk@vnet.ibm.com

Abstract. We present a unifying framework for expressing and analyzing events at various levels of atomicity in distributed computations. 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. We identify and prove two 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.

1 Introduction

In the literature on distributed system executions (also known as computations), events have been implicitly modeled in the isolated contexts of various applications, e.g., designing communication primitives [2, 3, 7], global states [5], concurrency measures [6, 9], deadlock detection [12], clock systems [10, 14, 17], termination detection [16], mutual exclusion [20], debugging [8], fault-tolerance and transactions [4, 11, 19]. 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 is crucial in modeling distributed activities to provide different abstract views. Lamport also argued that it is useful to assume that primitive elements between which concurrency is modeled are nonatomic for studying basic questions about nonatomicity [15]. 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. We define system executions for the various levels of atomicity by first defining a system execution dealing with the most elementary events, suitably identified. We then hierarchically compose system executions of coarser levels of atomicity by using the system executions at a finer level of atomicity.

We also prove that each level of atomicity satisfies two 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 Defn. 3 and Theorems 1, 3, and 5 for the four levels of atomicity considered.) [Property **P2:**] The events at any level of atomicity ordered by the corresponding ordering relation form a partially ordered set (poset) (See Defn. 3 and Theorems 2, 4, and 6 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 the graph property P1 or P2 that applies to any one level of atomicity applies to all levels of atomicity.

Section 2 presents the system model. Section 3 presents the events at four levels of atomicity by a hierarchical composition, and gives their applications. Section 4 concludes. The full paper [13] includes the proofs of theorems stated here.

2 System Model

A distributed system is a set of processes connected by communication channels. Depending on the level of atomicity being modeled, both processes and channels are modeled as nodes, or only processes are modeled as nodes that communicate with each other. Let E be the set of the most elementary events in a system execution, i.e., a run of a computation. We assign a semantic meaning to E later. Events of E are partitioned into local computations at a node, assuming that each event of E occurs at one node only. Each local computation is a linearly ordered set. An event e in partition i is denoted e_i. The computation at node i is a sequence of events and the system computation is the collection of computations at the various nodes. The initial event in each partition i is $\perp_i$. For finite computations, the final event in each partition i is $\top_i$.

Nodes communicate with each other by passing messages. A channel cannot generate, consume, or alter messages, but can permute the order of delivery of messages. The local action of sending (receiving) a message is a send (receive) event. The message sent at any send event is distinct from all messages sent at other send events at the level of atomicity being considered. The transfer of a message between a pair of process nodes takes finite time on a global time scale but between a process node and a channel node, it is instantaneous. The set of events that occur on 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.

Events in a computation are ordered by the causality relation $<$ on E [14]. An edge that orders two events on the same node (different nodes) is termed a *local edge* (*message edge*). A cut C is a subset of E such that if $e_i \in C$ then $\forall e'_i : e'_i < e_i$, we have $e'_i \in C$. A *consistent cut* is a downward-closed subset of E in $(E, <)$. $\mathcal{C}$, the set of cuts of a poset $(E, <)$, forms a lattice $(\mathcal{C}, \subset)$ with the operations $\bigcup$ and $\bigcap$. $\mathcal{CC}$, the set of consistent cuts of a poset $(E, <)$, forms a sublattice of $\mathcal{C}$, as shown in [17].

We use the formalism of hierarchical views of a system execution introduced by Lamport [15] to define events at various levels of atomicity in terms of elementary actions in a system. The choice of actions treated as elementary is based on the need to model sufficiently fine-grained actions for the known applications.

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 $\langle \mathcal{A}_x, <_x \rangle$. $\mathcal{A}_x$ and $<_x$ are different for each level of atomicity x. The term "atom" will be used interchangably with "event"; individual events (or atoms) and the set of events (or atoms) are denoted A_x and $\mathcal{A}_x$, respectively, to emphasize their atomic nature. The subscript will be dropped when the context is clear.

Consider $(\mathcal{A}_\alpha, <_\alpha)$ and $(\mathcal{A}_\beta, <_\beta)$, where $\mathcal{A}_\alpha$ and $\mathcal{A}_\beta$ are sets and $<_\alpha$ and $<_\beta$ are relations on the elements of $\mathcal{A}_\alpha$ and $\mathcal{A}_\beta$, respectively. Let mapping μ_β be a one-many surjective mapping that maps each element A_β of $\mathcal{A}_\beta$ to a non-empty subset of $\mathcal{A}_\alpha$. If μ_β^{-1} is a function then $\mathcal{A}_\beta$ defines a partition on $\mathcal{A}_\alpha$ — this means each element A_α of $\mathcal{A}_\alpha$ is contained in exactly one element A_β of $\mathcal{A}_\beta$, and an element A_β may contain multiple elements from $\mathcal{A}_\alpha$. Each element A_β in $\mathcal{A}_\beta$ is a set that is a higher level

grouping of the events in $\mathcal{A}_\alpha$ that is of interest to some application. μ_β is specified so as to define meaningful events at an appropriate level of atomicity $(\mathcal{A}_\beta, <_\beta)$ in terms of the events specified at the level of finer atomicity in $(\mathcal{A}_\alpha, <_\alpha)$.

There are two cases to consider when we define a system execution $S_\beta = \langle \mathcal{A}_\beta, <_\beta \rangle$. (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_α at a finer level of atomicity. $\mathcal{A}_\beta$ contains events at a coarser level of atomicity than $\mathcal{A}_\alpha$. (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 $\langle \mathcal{A}_\beta, <_\beta \rangle$ where $\mathcal{A}_\beta$ is a set and $<_\beta$ is a ordering relation on $\mathcal{A}_\beta$.*

S_β is specified in terms of a mapping $\mu_\beta : S_\beta \longrightarrow S_\alpha$, where S_α is a system execution at a finer level of atomicity such that:

1. *μ_β maps each element in $\mathcal{A}_\beta$ to a subset of $\mathcal{A}_\alpha$.*
2. *μ_β defines $<_\beta$ in terms of $<_\alpha$.*

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., $\langle E, < \rangle$, for the system execution.

3 Modeling Events in a Distributed Computation

In Sections 3.1, 3.2, 3.3, and 3.4, we define four levels of atomicity S_{dist}, S_{SR}, S_{react}, and S_{TL}, respectively, in a hierarchical manner, starting with the finest level S_{dist} to which we assign the semantic model of $\langle E, < \rangle$.

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, they are not necessarily atomic. It is shown in [7] that all such constructs can be expressed as some combination of one of the following primitive events.

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 $\mathcal{PS}$.
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 it any incoming message that matches the parameters and the sender-id specified. This event can complete before the received message 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 $\mathcal{WRM}$.

The events in $\mathcal{PS}$, $\mathcal{WB}$, $\mathcal{WSM}$, $\mathcal{PR}$, and $\mathcal{WRM}$ occur on process nodes. In order that the computation can progress, 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) on a process node, there exists a corresponding receive event on the channel node. For each WB, WSM and WRM event (which are receive events) on a process node, there exists a corresponding send event on the channel node. The following definition captures this relation.

Definition 2 *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 at e.*

$match(e)$ exists and is unique. (Its definition can be extended to multicasts.) Blocking

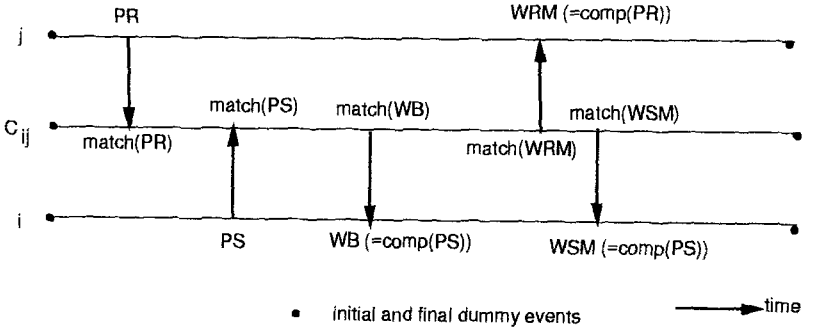

Fig. 1. Message Communication Events at the Finest Level of Atomicity.

and nonblocking, as well as synchronous and asynchronous sends and receives can be executed using the above primitive events [3, 7].

Figure 1 illustrates the effects of events PS, WB, WSM, PR and WRM, as well as Definition 2, by showing the message transfer from process i to process j on channel c_{ij}. 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 at most one of them would be used. The notation comp(PS) and comp(PR) for the events will be explained subsequently by Definition 4.

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

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

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

- $\mathcal{SD}_{dist} = \mathcal{PS} \bigcup \mathcal{PR} \bigcup \{match(WB) : WB \in \mathcal{WB}\} \bigcup \{match(WSM) : WSM \in \mathcal{WSM}\} \bigcup \{match(WRM) : WRM \in \mathcal{WRM}\}$

- $\mathcal{RC}_{dist} = \mathcal{WB} \bigcup \mathcal{WSM} \bigcup \mathcal{WRM} \bigcup \{match(PS) : PS \in \mathcal{PS}\} \bigcup \{match(PR) : PR \in \mathcal{PR}\}$
- $\mathcal{IN}_{dist} = \mathcal{IN}$

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

Definition 3 *System execution* $S_{dist} = \langle \mathcal{A}_{dist}, <_{dist} \rangle$, *where* $\mu_{dist}(S_{dist} \longrightarrow S_{dist})$ *is a 1-1 identity mapping. The semantic model of* S_{dist} *is* $(E, <)$, *where* $\mathcal{A}_{dist}$ *is* E *and* $<_{dist}$ *is the causality relation on* $\mathcal{A}_{dist}$.

From Definition 3, it follows that S_{dist} satisfies – [Property P1:] Atoms of $\mathcal{A}_{dist}$ partition events (atoms) in E, and [Property P2:] $(\mathcal{A}_{dist}, <_{dist})$ is a poset.

Applications: Complex communication constructs for specific communication styles, such as remote procedure calls (RPC) [2], conversations or dialogs [3], and messaging and queuing constructs, can be designed using PS, WB, WSM, PR, and WRM events of S_{dist}. The primitive events of S_{dist} can provide a yardstick for evaluating the flexibility of network programming style permitted by complex communication constructs. Another application is the design of nonblocking asynchronous programs at the application layer that use blocking synchronous communication at the transport layer between their output and input buffers. The synchronous communication between the sender's output buffer and the receiver's input buffer is done by a transport level acknowledgement. A specific example of this application is the implementation of causal ordering among message unicasts [18] without the application program blocking.

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 [2] or the CPI-C communications programming interface [3], 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.

Observe that in S_{dist}, a receive initiated by a PR event completes at the corresponding WRM event. Similarly, a send initiated by a PS event completes at the corresponding WB or WSM event. Based on this observation, we define the complement, (abbreviated *comp*), of these events to define the relation between events at a process node that complement other events on the same process node. The *comp* relation, along with the *match* relation (Definition 2) will be used to group events in S_{dist} together at the coarser level of atomicity S_{SR}.

Definition 4 $comp(e)$ *is defined as follows [7]:*

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.*

Any send or receive event e on a process node in S_{dist} identifies the set $\{ e, comp(e), match(e), match(comp(e)) \}$ – this set will form an atomic event in S_{SR}.

Definition 5 *System execution* $S_{SR} = (\mathcal{A}_{SR}, <_{SR})$ *is defined by a mapping* $\mu_{SR} : S_{SR} \longrightarrow S_{dist}$ *as follows:*

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

   Then, $A_{SR} <_{SR} A'_{SR}$ *iff key_member*$(A_{SR}) <_{dist}$ *key_member*(A'_{SR}).

It is shown in [13] 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 $\mathcal{A}_{dist}$. Note that even if $A_{SR} <_{SR} A'_{SR}$, it may be that $\exists\, A_{dist} \in A_{SR}\ \exists\, A'_{dist} \in A'_{SR} : A'_{dist} <_{dist} A_{dist}$.

Theorem 1 (P1:) *The atoms of* $\mathcal{A}_{dist}$ *are partitioned into atoms in* S_{SR}.

The proof of Theorem 1 [13] also shows that $\mathcal{A}_{SR}$ can be partitioned into $\mathcal{SD}_{SR}$, $\mathcal{RC}_{SR}$, and $\mathcal{IN}_{SR}$, where:

- $\mathcal{SD}_{SR} = \{\ A_{SR} \in \mathcal{A}_{SR} : key\_member(A_{SR}) \in \mathcal{PS}\ \}$
- $\mathcal{RC}_{SR} = \{\ A_{SR} \in \mathcal{A}_{SR} : key\_member(A_{SR}) \in \mathcal{WRM}\ \}$
- $\mathcal{IN}_{SR} = \{\ A_{SR} \in \mathcal{A}_{SR} : key\_member(A_{SR}) \in \mathcal{IN}\ \}$

Theorem 2 (P2:) *The atoms in* $\mathcal{A}_{SR}$ *ordered by* $<_{SR}$ *form poset* $(\mathcal{A}_{SR}, <_{SR})$.

The following corollary is used to analyze system executions S_{TL} in Section 3.4.

Corollary 1 $\mathcal{CC}_{SR}$, *the set of consistent cuts of poset* $(\mathcal{A}_{SR}, <_{SR})$, *forms a sublattice of* $\mathcal{C}_{SR}$, *the set of all cuts of* $(\mathcal{A}_{SR}, <_{SR})$. *(from Theorem 2 and [17]).*

Applications: There are many applications for which each complex send and receive construct, and internal event in the computation is explicitly modeled as a single event at the process nodes in S_{SR}. Global state and snapshot definition and computation [5], concurrency measures for a system execution [6, 9], clock systems for distributed computations [10, 14, 17], transfer of knowledge, checkpointing and recovery [4, 21], leader election, mutual exclusion algorithms [20], and distributed deadlock detection [12] all deal with send and receive events in the S_{SR} view of the system execution.

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 [16] and debugging [8], 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} and using regular expressions over SD_{SR}, RC_{SR} and IN_{SR} events, and the auxiliary event $C(\phi)$.

Definition 6 *System execution* $S_{react} = \langle A_{react}, <_{react} \rangle$ *is defined by a mapping* $\mu_{react} : S_{react} \longrightarrow S_{SR}$ *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* A_{SR} *and occur on node* x*, that satisfy the regular expression* $(\perp_x (IN_{SR}|RC_{SR})^*(IN_{SR}|SD_{SR})^*(C(\phi))$
   (b) $A^{x,i}_{react}, i > 1$ *is the maximal sequence of events that belong to* A_{SR} *and occur on node* x*, that satisfy the context-sensitive regular expression:* $A^{x,i-1}_{react} A^{x,i}_{react} = A^{x,i-1}_{react}(RC_{SR}(IN_{SR}|RC_{SR})^*(IN_{SR}|SD_{SR})^*(C(\phi))^*)$
2. $A_{react} <_{react} A'_{react}$ *iff* $(\exists A_{SR} \in A_{react}, \exists A'_{SR} \in A'_{react} : A_{SR} <_{SR} A'_{SR})$.

$A^{x,i}_{react}$ is the i^{th} reactive event on node x. The superscripts/subscript are dropped if there is no ambiguity. Figure 2 shows the reactive events in a distributed execution.

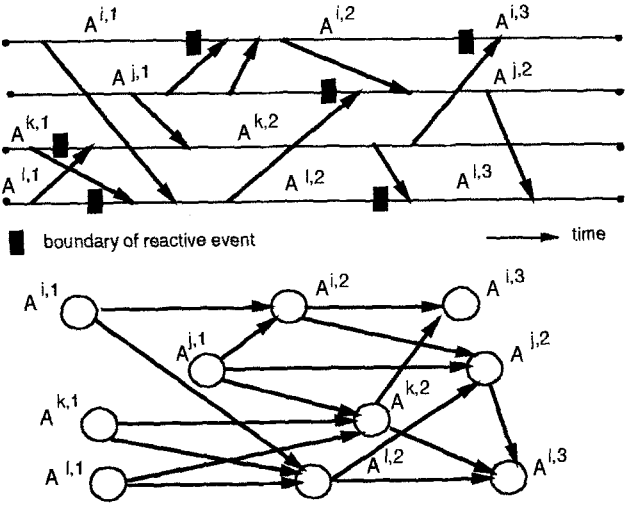

Fig. 2. Reactive Events.

Theorem 3 (P1:) *The atoms of* A_{SR} *are partitioned into atoms in* S_{react}.

Theorem 4 (P2:) *The atoms in* A_{react} *ordered by* $<_{react}$ *form poset* $(A_{react}, <_{react})$.

It follows that no event in A_{react} has both an edge that goes to another event in A_{react} and an incoming edge from that other event.

Applications: Computation termination [16] can be modeled by reactive events as follows. Consider a system in which: (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 can become idle 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 the number of $match(PS)$ events and $match(WRM)$ events is the same in the S_{dist} view.

A message race occurs at an RC_{SR} event if it can receive one of multiple messages. Debugging based on controlled execution of message races examines the possible

executions corresponding to one space-time diagram [8]. The definition of reactive events (Defn. 6) for debugging does not use any auxiliary event $C(\phi)$, i.e., ϕ=$false$. A message that could be received in a reactive event A may have been sent in a reactive event A' such that $A' < A \bigvee (A' \not< A \bigwedge A \not< A')$. For e.g., in Figure 2, if A is $A^{i,2}$, then A' is any of $A^{i,1}$, $A^{i,2}$, $A^{i,3}$, $A^{j,1}$, $A^{k,1}$, $A^{k,2}$, and $A^{l,1}$. During controlled (replay) executions for event A, such events A' are forced to complete before A begins, before 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 belong to multiple process nodes.

Definition 7 *A transitless cut TLC_{SR} is a consistent cut in $(\mathcal{A}_{SR}, <_{SR})$ such that the only ordering edges between it and the rest of $\mathcal{A}_{SR}$ are local edges at process nodes (defined in Section 2).*

The system state after the execution of events in a transitless cut is a *transitless global state*. Such states have the property that the effects of the past computation are contained in only local edges of process nodes in a S_{SR} view of the execution, viz., the process states, and no messages are in transit. We examine this level of atomicity using Corollary 1 [17] and properties of lattices, unlike previous work (see Applications).

Lemma 1 *$\mathcal{TLC}_{SR}$, the set of transitless cuts of a poset $(\mathcal{A}_{SR}, <_{SR})$, forms a sublattice of $\mathcal{CC}_{SR}$, the set of all consistent cuts of $(\mathcal{A}_{SR}, <_{SR})$, with operations $\bigcup$ and $\bigcap$.*

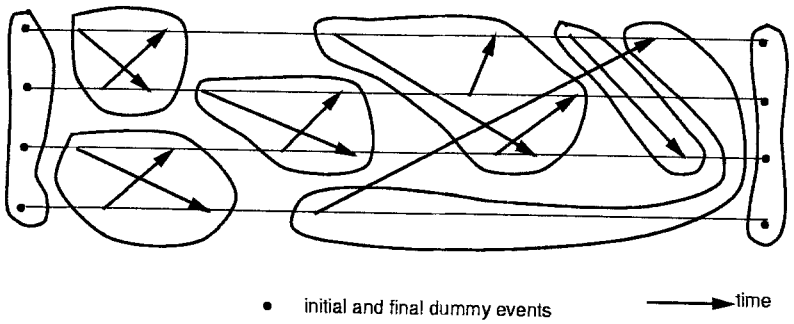

Fig. 3. Events between Transitless Global States.

From Lemma 1, 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 8 *System execution $S_{TL} = \langle \mathcal{A}_{TL}, <_{TL} \rangle$ is defined by a mapping μ_{TL} : $S_{TL} \longrightarrow S_{SR}$ as follows:*

1. $\mathcal{A}_{TL}$ = { $(TLC^u - TLC^l) : TLC^u, TLC^l \in \mathcal{TLC}_{SR} \bigwedge$ *length*$[TLC^l, TLC^u] = 1$ }
2. $<_{TL}$ *is the transitive closure of* $<_{tlc}$ *where* $(TLC^u - TLC^l) <_{tlc} (TLC'^u - TLC'^l)$ *iff* $(\exists e \in (TLC^u - TLC^l), \exists e' \in (TLC'^u - TLC'^l) : e <_{SR} e')$.

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 $TLC^u - TLC^l$) 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. This property is important and is used in the proof of Theorem 6. Figure 3 shows the events in S_{TL}. 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 $(\mathcal{A}_{SR}, <_{SR})$ 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 on processes in a S_{SR} view, and it affects the rest of the computation only through outgoing local edges on processes in the S_{SR} view.

Theorem 5 (P1:) *The atoms of* $\mathcal{A}_{SR}$ *are partitioned into atoms by* S_{TL}.
Theorem 6 (P2:) *The atoms in* $\mathcal{A}_{TL}$ *ordered by* $<_{TL}$ *form poset* $(\mathcal{A}_{TL}, <_{TL})$.

Applications: Transitless states are used in applications like fault-tolerance, checkpointing/recovery [4, 11, 21], synchronization [19], and transactions [4, 11]. Transitless states were forced in [11, 19] for synchronization and checkpointing/recovery. Transaction systems create transitless states at the end of each transaction using commit protocols [4]. In these applications, the transitless states along the boundaries of only certain events in S_{TL} are recorded; in case of failure, the most recent recorded transitless state is restored for recovery. Transitless states and their applications were also examined in [1]. Transitless states can also be shown to be useful to reset vector clocks [10, 17]; after reset at a transitless state, wrong inferences about causality cannot be drawn due to messages with high timestamp values sent before reset.

4 Discussion

We presented a unifying framework for expressing and analysing events at various levels of atomicity in distributed computations. In the framework, events at a coarser level of atomicity are defined in terms of events at a finer level of atomicity using hierarchical composition and lattices. The global states at various levels of atomicity correspond to embedded lattices of global states. The framework was applied to four levels of atomicity here, and can be applied to parallel system executions as shown in [13]. The system model can be varied to allow message losses and multicasts as in [13].

The system execution at every level of atomicity was shown to have two properties. [Property P1]: If S_β is defined in terms of S_α, 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 properties applies to all levels of atomicity. For example, the proof for execution S_{SR} that synchronous communication between processes guarantees causal ordering of message unicasts applies without

change to the proof for execution S_{dist} that asynchronous communication between processes, with synchronous communication over channels between the (infinite) output and input process buffers, respectively, guarantees causal ordering of message unicasts [18]. A second example is the reuse of concurrency measures described in S_{SR} [6, 9] for gauging concurrency of incremental debugging in S_{react}; this latter measure is useful to determine the number of nondeterministic and deterministic replays.

References

1. Ahuja, M., Kshemkalyani, A. D., Carlson, T.: A Basic Unit of Computation in Distributed Systems. Proc. 10th IEEE Int. Conf. Distrib. Comput. Systems (1990) 12–19
2. Ananda, A., Tay, B., Koh, E.: A Survey of Asynchronous RPC. ACM Operating Systems Review (1992)
3. Arnette, W., Kshemkalyani, A.D., Riley, W., Sanders, J., Schwaller, P., Terrien, J., Walker, J.: CPI-C: An API for Distributed Applications. IBM Systems Journal **34(3)** (1995) 501–518
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, Addison-Wesley (1987)
5. Chandy, K.M., Lamport, L.: Distributed Snapshots: Global States of a Distributed System. ACM Trans. Comput. Systems **3(1)** (1985) 63–75
6. Charron-Bost, B.: Measure of Parallelism of Distributed Computations. Proc. STACS 89, In LNCS 349 Springer-Verlag (1989) 434–445
7. Cypher, R., Leu, E.: Repeatable and Portable Message-Passing Programs. Proc. 13th ACM Symp. on Principles of Distributed Computing (Aug. 1994) 22–31.
8. Damodaran-Kamal, S., Francioni, J.: Nondeterminacy:Testing and Debugging in Message Passing Parallel Programs. ACM/ONR Workshop on Debugging (1993) 118–128
9. Fidge, C.A.: A Simple Run-Time Concurrency Measure. The Transputer in Australasia, Eds. T. Bossomaier, T. Hintz, J. Hulskamp, IOS Press (1990) 92–101
10. Fidge, C.A.: Timestamps in Message-Passing Systems That Preserve Partial Ordering. Australian Computer Science Communications **10(1)** (Feb. 1988) 56–66
11. Fisher, M., Griffeth, N., Lynch, N.: Global States in a Distributed System. IEEE Trans. Software Engineering **8(3)** (May 1982) 198–202
12. Kshemkalyani, A.D., Singhal, M.: On Characterization and Correctness of Distributed Deadlock Detection. Journal of Parallel and Distributed Computing **22(1)** (July 1994) 44–59
13. Kshemkalyani, A.D.: A Unifying Framework for Viewing Atomic Actions in Parallel and Distributed Systems. IBM Tech. Rep. TR29.2014 (1995)
14. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21(7)** (July 1978) 558–565
15. L. Lamport.: On Interprocess Communication, Part I: Basic Formalism, Part II: Algorithms. Distributed Computing, **1** (1986) 77–101
16. Mattern, F.: Algorithms for Distributed Termination Detection. Distributed Computing **2** (1987) 161–175
17. F. Mattern, Virtual Time and Global States of Distributed Systems. Parallel and Distributed Algorithms, North-Holland (1989) 215-226
18. Mattern, F., Fünfrocken, S.: A Nonblocking Lightweight Implementation of Causal Order Message Delivery. In LNCS 938 Springer-Verlag (1995) 197–213
19. Randell, B.: System Structure for Software Fault Tolerance. IEEE Trans. Software Engg. **1(2)** (1975) 220–232
20. Singhal, M.: A Taxonomy of Distributed Mutual Exclusion. Journal of Parallel and Distributed Computing, **18(1)** 1993 94–101
21. Strom, R.E., Yemini, S.: Optimistic Recovery in Distributed Systems. ACM Trans. on Computer Systems **3(3)** (1985) 204–226