Using Operational Intuition about Events and Causality in Assertional Proofs

Ching-Tsun Chou ⟨chou@cs.ucla.edu⟩

Computer Science Department, University of California at Los Angeles
Los Angeles, CA 90024, U.S.A.

Abstract. There are two approaches to reasoning about distributed algorithms. In the *operational* approach, one reasons "dynamically" about the events that can occur and the temporal precedence relation (called *causality*) between those events. In the *assertional* approach, one reasons "statically" about reachable states by means of assertions (called *invariants*) that are true for all reachable states. The purpose of this paper is to show that the operational and assertional approaches can be combined in such a way that the advantages of both approaches are attained. On the one hand, operational reasoning about events and causality affords an *intuitive* way to think about and understand distributed algorithms. On the other hand, assertional proof techniques allow such intuitive understanding to be used to guide and structure *formal* correctness proofs. By combining these two approaches, the difficult task of verifying distributed algorithms is made easier. It is also shown that the notions of events and causality can be used to motivate proofs based both on forward simulation/history variables and on backward simulation/prophecy variables.

(Last revised: 26 February 1995)

1 Introduction

There are two approaches to reasoning about distributed algorithms. In the *operational* approach, one reasons "dynamically" about the events that can occur and the temporal precedence relation (called *causality*) between those events. In the *assertional* approach, one reasons "statically" about reachable states by means of assertions (called *invariants*) that are true for all reachable states. While most people seem to find it easier and more natural to reason about events and causality than to approximate reachable states with invariants, most successful methods in practice for reasoning about distributed algorithms are assertional [3, 8, 11, 12, 14]. Lamport [10] describes the situation thus:

> Most computer scientists find it natural to reason about a concurrent program in terms of its behaviors—the sequence of events generated by its execution. Experience has taught us that such reasoning is not reliable; we have seen too many convincing proofs of incorrect algorithms. This has led to assertional proof methods, in which one reasons about the program's state instead of its behavior. Unlike behavioral reasoning, assertional proofs can be formalized—*i.e.*, reduced to a series of precise steps that can, in principle, be machine-verified.

The purpose of this paper is to show that the operational and assertional approaches can in fact be combined in such a way that the advantages of both approaches are attained. On the one hand, operational reasoning about events and causality affords an *intuitive* way to think about and understand distributed algorithms. On the other hand, assertional proof techniques allow such intuitive understanding to be used to guide and structure *formal* correctness proofs. By combining these two approaches, the difficult task of verifying distributed algorithms is made easier.

This work arises from our dissatisfaction with the current assertional methods for reasoning about distributed algorithms [3, 8, 11, 12, 14]. While these methods do provide a repertoire of proof techniques by means of which all distributed algorithms can in principle be verified, the application of assertional techniques in practice is still an *ad hoc*, trial-and-error process that needs a great deal of ingenuity. A good example of this trial-and-error process is the construction of invariants, which is a major part of any assertional proof [11]. The invariants one finds in the literature often seem to be "pulled out of a hat" with little or no explanation as to how they are invented. Anyone who has tried to devise invariants for nontrivial algorithms can understand the need for a systematic way of thinking that can make the construction of invariants less trial-and-error. This paper shows, among other things, that the notions of events and causality support such a systematic way of thinking.

It should be pointed out that *not* everyone agrees with our view that the current assertional methods are inadequate in practice. In response to an earlier version of this paper [4], Vaandrager [20] argues that the current assertional methods already provide sufficient guidance for the practical verification of distributed algorithms. This difference in opinion can only be settled, to the extent that it can be settled, by comparing the proofs produced according to the two approaches. The readers are invited to make such comparisons.

As mentioned earlier, we propose to use the the notions of events and causality as the basis of an intuitive understanding of distributed algorithms. By *events* we mean the (names of) occurrences of atomic actions in an execution of a distributed algorithm and by *causality* we mean the essential temporal precedence relation between events that is respected by all possible interleavings of concurrent events in that execution. The computation of a distributed algorithm can be viewed as the (generally nondeterministic) unfolding of a causality relation between events. Normally there are much fewer possible causality relations than possible interleavings, so thinking about events and causality is often an excellent way to understand a distributed algorithm. Thus, given a distributed algorithm Π^D, it is often easy to write down an "event view" Π^E whose sole purpose is to generate the events and causality relations that Π^D can generate. Then, to

$$\begin{tikzcd} \Pi^E \arrow[r, phantom, "\models"] & P^E \arrow[d, "P^E \{\!\!\{R^{-1}\}\!\!\} P^D"] \\ \Pi^D \arrow[u, "\mathsf{Sim}(R)(\Pi^D)(\Pi^E)"] \arrow[r, phantom, "\models"] \arrow[ur, phantom, "\Downarrow"] & P^D \end{tikzcd}$$

Fig. 1. The general scheme of our method

prove that Π^D satisfies a temporal property P^D (denoted $\Pi^D \models P^D$), it is sufficient to prove that Π^E satisfies a temporal property P^E (denoted $\Pi^E \models P^E$) which has the same form as P^D, provided that there are a *simulation* R from Π^D to Π^E (denoted $\mathsf{Sim}(R)(\Pi^D)(\Pi^E)$) and a *translation* from P^E to P^D via the inverse of R (denoted $P^E \{\!\!\{R^{-1}\}\!\!\} P^D$). This is depicted schematically in Figure 1. A major portion of the proof of the concrete program Π^D (including all temporal reasoning!) is carried out in terms of the abstract program Π^E. This is desirable not only because Π^E is more abstract and hence easier to reason about than Π^D, but also because different concrete programs may share the same abstract program whose proof can then be reused. Also, writing down the simulation R is not hard; in practice, it essentially amounts to expressing the current state of Π^D in terms of that of Π^E.

Unlike most simulation techniques in the literature [13], our formulation of simulation does not distinguish between external and internal actions or state variables. Instead, we use simulations to establish a relation between concrete and abstract executions and then translate properties of the latter into those of the former via that relation. Our formulation of translation is based on a novel interpretation of Hoare triples [7], which turn out to have many nice properties (called *Reduction Lemmas*) that allow one to reduce the proof of a translation relation between two complex properties *of the same form* to the proofs of translation relations between their corresponding constituents. The Reduction Lemmas can also be used to give a simple treatment of property preservation via simulation that is applicable to linear-time, branching-time, modal, and fixpoint logics [5].

The notions of events and causality are not new [9, 16]. A variety of *non*-interleaving models of concurrency have been proposed in the literature, such as partial orders of events [9, 19], event structures [16], Mazurkiewicz traces [15], and asynchronous transition systems [19]. While a great deal of theoretical investigation has been conducted, none of these works addresses the practical problem of how to verify realistic distributed algorithms using these models. We hope that this paper is a small step in that direction. Also, it should be noted that although we use the intuition about events and causality as an *informal* guide for structuring proofs, the *formal* foundation of our method is still the interleaving view of concurrency.

Our ideas will be explained in terms of the verification of DSum, which is a simple but typical distributed algorithm adapted from Segall's PIF (propagation of information with feedback) algorithm [18]. Section 2 describes DSum and the properties that we wish to prove about it. Section 3 describes an event view of DSum and shows how operational intuition about events and causality can be used to guide the proof of the event view. Section 4 relates the distributed view (Section 2) and the event view (Section 3) of DSum as outlined in Figure 1, in order to deduce the desired properties of the former from those of the latter. Section 5 describes another event view of DSum that uses a prophecy variable [1] (the event view in Section 3 uses only history variables) and shows that, although the event view with a prophecy variable is simpler, the proof needed to relate it to the distributed view is more complicated. Section 6 contains some concluding remarks. An appendix summarizes the miscellaneous facts and notations about sets, graphs, automata, properties, simulation, and translation needed in this paper, which the reader should skim through before proceeding further.

The symbol "$\triangleq$" means "equals by definition"; "iff" means "if and only if".

STATE VARIABLES:

$$
\begin{array}{rll}
pc \in \mathsf{Node}(G) \rightarrow \{\mathtt{Idle}, \mathtt{Busy}\} & & \text{(* Program counter *)} \\
par \in \mathsf{Node}(G) \rightarrow \mathsf{Node}(G_r) & & \text{(* Parent *)} \\
acc \in \mathsf{Node}(G) \rightarrow \mathcal{W} & & \text{(* Accumulator *)} \\
cnt \in \mathsf{Node}(G) \rightarrow \mathcal{N} & & \text{(* Counter *)} \\
mq \in \mathsf{Link}(G_r) \rightarrow \mathcal{W}^* & & \text{(* Message queue *)}
\end{array}
$$

INITIAL CONDITION:

$(\,\forall\, i \in \mathsf{Node}(G) : pc(i) = \mathtt{Idle}\,) \wedge$

$(\,\forall\, l \in \mathsf{Link}(G_r) : mq(l) = \textbf{if } l = (r, r') \textbf{ then } \langle\!\langle \mathsf{O} \rangle\!\rangle \textbf{ else } \langle\!\langle\ \rangle\!\rangle\,)$

ACTIONS:

$\mathtt{Recv}(i, j)$

Precondition:

$i \in \mathsf{Node}(G) \wedge j \in \mathsf{Nbr}(G_r)(i) \wedge (mq(j,i) \neq \langle\!\langle\ \rangle\!\rangle) \wedge (pc(i) = \mathtt{Idle})$

Effect:

$pc(i) := \mathsf{Busy}\ ; \quad par(i) := j\ ; \quad acc(i) := wt(i)\ ;$

$\textbf{let } K = \mathsf{Nbr}(G_r)(i) \setminus \{j\} \textbf{ in}$

$\quad cnt(i) := |K|\ ; \quad \textbf{for } k \in K \textbf{ do } mq(i,k) := mq(i,k)^\frown \langle\!\langle \mathsf{O} \rangle\!\rangle\ ;$

$mq(j,i) := \mathsf{tail}(mq(j,i))\ ;$

$\mathtt{Recv}'(i, j)$

Precondition:

$i \in \mathsf{Node}(G) \wedge j \in \mathsf{Nbr}(G_r)(i) \wedge (mq(j,i) \neq \langle\!\langle\ \rangle\!\rangle) \wedge (pc(i) = \mathtt{Busy})$

Effect:

$acc(i) := acc(i) \oplus \mathsf{head}(mq(j,i))\ ; \quad cnt(i) := cnt(i) - 1\ ;$

$mq(j,i) := \mathsf{tail}(mq(j,i))\ ;$

$\mathtt{Send}(i)$

Precondition:

$i \in \mathsf{Node}(G) \wedge (pc(i) = \mathsf{Busy}) \wedge (cnt(i) = 0)$

Effect:

$mq(i, par(i)) := mq(i, par(i))^\frown \langle\!\langle acc(i) \rangle\!\rangle\ ;$

$pc(i) := \mathtt{Idle}\ ; \quad par(i) := ?\ ; \quad acc(i) := ?\ ; \quad cnt(i) := ?\ ;$

Fig. 2. Prog^D: Distributed view of DSUM

2 DSum: A Distributed Summation Algorithm

Let G be a *connected* graph whose nodes represent autonomous processors and whose links represent communication channels via which the processors can send messages to each other, and $wt \in \mathsf{Node}(G) \rightarrow \mathcal{W}$ be an assignment of weights to nodes in G. The purpose of DSum is to compute the sum of wt's of all nodes in G, denoted $sum \triangleq \sum i \in \mathsf{Node}(G) : wt(i)$. Let r (for *root*) be a dummy node that is *not* in G but is connected to exactly one node r' in G, and G_r be the graph obtained by adding the edge $\{r, r'\}$ to G. The root r has no program and is always idle; it serves the purpose of allowing the nodes in G to be treated uniformly.

DSum is formalized as the automaton Prog^D shown in Figure 2, where the superscript ^{D} indicates that this is the "distributed view" of DSum; in the next section, the "event view" of DSum will be given the superscript ^{E}. Prog^D has four state variables at each node n in G and a message queue over each link l in G_r, whose intuitive meanings are given by comments in Figure 2. Initially, all we know is that each node's program counter is `Idle` and there is a single message O in transit from r to r'. Prog^D has three kinds of actions: $\mathtt{Recv}(i,j)$ is the action of i receiving its first message from j, marking j as its parent, setting its accumulator to its wt, and sending O to each of its neighbors except j; $\mathtt{Recv}'(i,j)$ is the action of i receiving a non-first message from j and adding that message to its accumulator; $\mathtt{Send}(i)$ is the action of i, having received a message from each of its neighbors, sending the value of its accumulator to its parent and then erasing (i.e., assigning arbitrary values to) all its state variables except its program counter. Prog^D terminates when there is a single message in transit from r' to r, which must be the desired sum.

Despite the apparent simplicity of Prog^D, it is not at all easy to write an invariant for Prog^D which is preserved by each of its actions and from which useful properties of Prog^D can be deduced. The only way to fully appreciate the difficulty is to try to do so oneself. Those who cannot afford the time needed are referred to [4], the third section of which gives a possible scenario of invariant development for Prog^D.

What we wish to prove about Prog^D is the following three temporal properties:

$$\mathsf{Prog}^D \models \Box(\mathsf{Done}^D \Rrightarrow \mathsf{Final}^D) \tag{1}$$

$$\mathsf{Prog}^D \models \Box(\mathsf{Done}^D \Rrightarrow \Box\, \mathsf{Done}^D) \tag{2}$$

$$\mathsf{Prog}^D \models \Diamond\, \mathsf{Done}^D \tag{3}$$

where Done^D signifies the termination of Prog^D:

$$\mathsf{Done}^D(pc, par, acc, cnt, mq) \triangleq \exists\, w \in \mathcal{W} : mq(r', r) = \langle\!\langle w \rangle\!\rangle$$

and Final^D characterizes the desired final states of Prog^D:

$$\begin{array}{l} \mathsf{Final}^D(pc, par, acc, cnt, mq) \triangleq \\ \quad (\forall\, i \in \mathsf{Node}(G) : pc(i) = \mathtt{Idle}) \wedge \\ \quad (\forall\, l \in \mathsf{Link}(G_r) : mq(l) = \textbf{if } l = (r', r) \textbf{ then } \langle\!\langle sum \rangle\!\rangle \textbf{ else } \langle\!\langle\, \rangle\!\rangle\,) \end{array}$$

(1) says that whenever Prog^D terminates, its state is in a desired configuration (viz., partial correctness), (2) says that once Prog^D terminates, it stays that way forever, and (3) says that Prog^D must eventually terminate.

3 An Event View of DSum

Before describing formally an event view of DSum, let us consider a few simple scenarios. First, suppose G_r is the tree shown in the left half of Figure 3. DSum can generate only one causality relation in this graph, which is outlined in the right half of Figure 3, where $\mathtt{R}_{i,j}$ (respectively, $\mathtt{S}_{i,j}$) denotes the event of node i receiving its first message from (sending its last message to) node j,

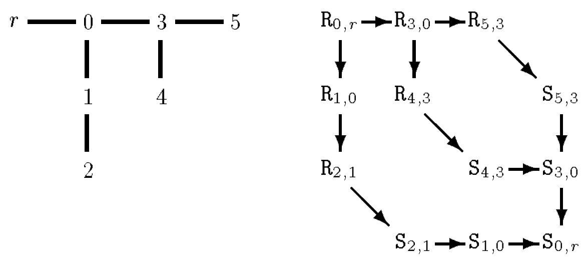

Fig. 3. The causality relation of DSUM in a tree

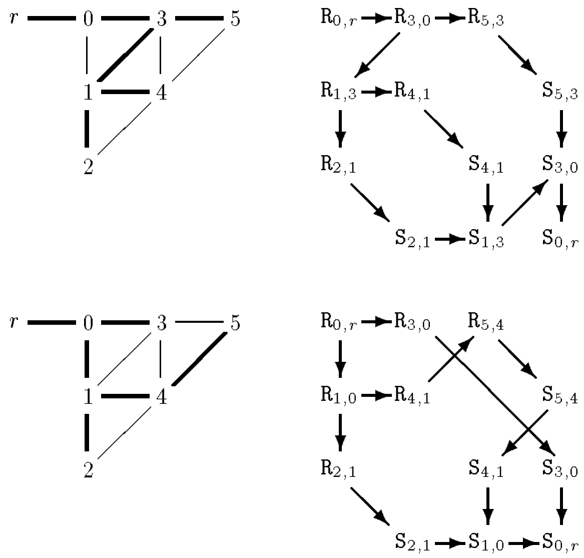

Fig. 4. Two causality relations of DSUM in a connected graph

and the arrows depict the causality relation. The causality relation captures only the essential temporal precedence between events. For example, $\mathtt{R}_{3,0}$ and $\mathtt{S}_{5,3}$ are ordered since $\mathtt{R}_{3,0}$ must occur before $\mathtt{S}_{5,3}$, while $\mathtt{R}_{1,0}$ and $\mathtt{S}_{5,3}$ are not ordered since they can occur in either order. In order not to clutter the diagram, the events of nodes receiving non-first messages are not shown, but it is not hard to see where they should be placed.

DSUM is "essentially deterministic" in a tree, in that it can generate only one causality relation in a tree. In a general (connected) graph, DSUM can generate many causality relations and hence is "essentially nondeterministic". For example, consider the graph shown in the upper-left quadrant of Figure 4 (ignore the distinction between thick and thin edges for the moment). A possible execution of DSUM in this graph goes like this: 0 receives its first message from the root and sends messages to 1 and 3; 3 receives its first message from 0 and sends messages to 1, 4, and 5; 1 receives its first message from 3; 4 receives its first message from 1; ...; and so on. The causality relation corresponding to this execution is outlined in the upper-right quadrant of Figure 4 and the spanning tree induced by the parent (*par*) pointers is indicated by thick edges in the graph in the upper-left quadrant. (Again, in order not to clutter the diagram, the events of nodes receiving non-first messages are not shown, and they are not hard to place either.) Note that if 1 receives its first message from 0 instead of 3, the execution may lead to the causality relation outlined in the lower-right quadrant of Figure 4 and the spanning tree shown in the lower-left quadrant. A little reflection shows that for *every* spanning tree T of G_r, there is an execution of DSUM in which T is chosen as the spanning tree induced by the parent pointers. Which spanning tree is chosen is not pre-determined, but depends on the relative speeds of messages. Thus DSUM is highly nondeterministic in general graphs even after the nondeterminism in the ordering of concurrent events has been factored out. Still, using the notions of events and causality, one can easily visualize the computation of DSUM as is done in Figure 4. Such easy visualization is quite impossible if one thinks purely in terms of states and transitions. This fact is a strong argument for using operational intuition about events and causality even in assertional proofs based on states and transitions.

An *event view* of DSUM is an operational representation of the events and causality relations that DSUM can generate. A choice in the formulation of such an event view is whether the spanning tree induced by the parent pointers is constructed piece by piece in the course of execution, or is chosen in one stroke at the very beginning of execution. The former needs only history variables (hence the subscript $_H$) and is described below, whereas the latter needs a prophecy variable (hence the subscript $_P$) and is described in Section 5.

The event view of DSUM with only history variables is formalized as the automaton Prog^E_H shown in Figure 5. Prog^E_H has only two state variables: T_H, which records the tree induced by the parent pointers of those nodes that have received at least one message, and O_H, which records the set of events that have occurred. Note that the set Event^E_H does *not* contain $\mathtt{Recv}(i,j)$ events, since their occurrences can be deduced from T_H and need not be recorded. Initially, T_H contains the root as its only node and no event has occurred.

The causality relations of Prog^E_H are expressed in terms of T_H and O_H by specifying the (immediate) *cause* $\mathsf{Cause}^E_H(ev)$ of each event $ev \in \mathsf{Event}^E_H$. Intuitively, Cause^E_H says that for $\mathtt{Recv}'(i,j)$ to occur, i must have received at least one message (signified by $i \in \mathsf{Node}(T_H)$) and either j is a kid of i in T_H and has sent its last message (to j), or j is in T_H but the edge $\{i,j\}$ is not in T_H, and that for $\mathtt{Send}(i)$ to occur, i must have received a message from each of its neighbors. The $\mathtt{Recv}(i,j)$ action extends the tree T_H by the edge $\{i,j\}$. The occurrence of an event $ev \in \mathsf{Event}^E_H$ is controlled by the causality relation Cause^E_H, which is determined by T_H.

Unlike Prog^D, the invariant of Prog^E_H is easy to write and simply says that T_H is always a tree containing the root and whenever an event has occurred, its cause must be true:

$$\mathsf{Inv}^E_H(T_H, O_H) \;\triangleq\; \mathsf{Tree}(T_H) \wedge r \in \mathsf{Node}(T_H) \wedge \forall\, ev \in O_H : \mathsf{Cause}^E_H(ev)(T_H, O_H) \tag{4}$$

The invariance of Inv^E_H is easy to prove and states that:

$$\mathsf{Prog}^E_H \;\models\; \Box\, \mathsf{Inv}^E_H \tag{5}$$

The termination of Prog^E_H is signified by the occurrence of $\mathtt{Send}(r')$:

$$\mathsf{Done}^E_H(T_H, O_H) \;\triangleq\; \mathtt{Send}(r') \in O_H$$

Clearly, once an event is added to O_H, it stays in O_H forever. In particular, once Prog^E_H terminates, it stays that way forever:

$$\mathsf{Prog}^E_H \;\models\; \Box(\mathsf{Done}^E_H \Rrightarrow \Box\, \mathsf{Done}^E_H) \tag{6}$$

The basic liveness properties of Prog^E_H are:[1]

$$\forall\, n < |\mathsf{Node}(G_r)| :\; \mathsf{Prog}^E_H \;\models\; (|\mathsf{Node}(T_H)| = n) \leadsto (|\mathsf{Node}(T_H)| > n) \tag{7}$$

which says that T_H must get bigger and bigger, and:

$$\forall\, ev \in \mathsf{Event}^E_H :\; \mathsf{Prog}^E_H \;\models\; \mathsf{Cause}^E(ev) \leadsto ev \in O_H \tag{8}$$

[1] Note that we are abusing notation in (7): by $(|\mathsf{Node}(T_H)| = n)$ we really mean a condition C such that $C(T_H, O_H) = (|\mathsf{Node}(T_H)| = n)$; similar remarks apply to (8), (9), and (13) as well.

EVENTS:

$\mathsf{Event}^E_H \triangleq \{\, \mathtt{Recv}'(i,j), \mathtt{Send}(i) \mid i \in \mathsf{Node}(G) \wedge j \in \mathsf{Nbr}(G_r)(i) \,\}$

STATE VARIABLES:

$T_H \in \mathsf{Subgraph}(G_r)$ (* Tree constructed so far *)

$O_H \in \mathcal{P}(\mathsf{Event}^E_H)$ (* Events occurred so far *)

INITIAL CONDITION:

$(T_H = \{r\}) \wedge (O_H = \emptyset)$

CAUSALITY:

$\mathsf{Cause}^E_H(\mathtt{Recv}'(i,j))(T_H, O_H) \triangleq$
$(\, i \in \mathsf{Node}(T_H) \wedge j \in \mathsf{Kid}(T_H)(i) \wedge \mathtt{Send}(j) \in O_H \,) \vee$
$(\, i \in \mathsf{Node}(T_H) \wedge j \in \mathsf{Node}(T_H) \wedge \{i,j\} \notin \mathsf{Edge}(T_H) \,)$

$\mathsf{Cause}^E_H(\mathtt{Send}(i))(T_H, O_H) \triangleq$
$i \in \mathsf{Node}(T_H) \wedge \forall\, j \in \mathsf{Nbr}(G_r)(i) \setminus \{\mathsf{Par}(T_H)(i)\} : \mathtt{Recv}'(i,j) \in O_H$

ACTIONS:

$\mathtt{Recv}(i,j)$

Precondition:

$i \in \mathsf{Node}(G) \wedge j \in \mathsf{Nbr}(G_r)(i) \wedge i \notin \mathsf{Node}(T_H) \wedge j \in \mathsf{Node}(T_H)$

Effect:

$T_H := T_H \cup \{\{i,j\}\}$;

ev

Precondition:

$ev \in \mathsf{Event}^E_H \wedge \mathsf{Cause}^E_H(ev)(T_H, O_H) \wedge ev \notin O_H$

Effect:

$O_H := O_H \cup \{ev\}$;

Fig. 5. Prog^E_H: Event view of DSum with only history variables

which says that once the cause of an event becomes true, the event itself must eventually occur. Both (7) and (8) are easy to prove. Since T_H never shrinks, (7) guarantees that T_H must be a spanning tree of G_r from some point on:

$$\mathsf{Prog}^E_H \models \Diamond\Box(\mathsf{Node}(T_H) = \mathsf{Node}(G_r)) \tag{9}$$

Once T_H becomes a spanning tree of G_r, (8) guarantees that $\mathtt{Recv}'(i,j)$ must eventually occur for all $\{i,j\}$ not an edge in T_H, then $\mathtt{Send}(i)$ must eventually occur for all i a leaf in T_H, then $\mathtt{Send}(i)$ must eventually occur for all i one tree edge removed from the leaves in T_H, ..., all the way up the tree T_H until $\mathtt{Send}(r')$ occurs. Therefore, Prog^E_H must eventually terminate:

$$\mathsf{Prog}^E_H \models \Diamond\, \mathsf{Done}^E_H \tag{10}$$

Properties (5), (6), and (10) of Prog^E_H are the counterparts of properties (1), (2), and (3) of Prog^D, respectively. The next section shows how to deduce the latter from the former using Figure 1.

4 Relating Distributed and Event Views of DSum

The following relation $\mathsf{Rel}^{D,E}_{H}$ is a forward simulation from Prog^{D} to Prog^{E}_{H}:

$$\begin{aligned}\mathsf{Rel}^{D,E}_{H}(s^D, s^E_H) \;\triangleq\; & \mathsf{Inv}^E_H(s^E_H) \wedge (\,\forall\, i \in \mathsf{Node}(G) : \mathsf{Rel}^{D,E}_{N,H}(i)(s^D, s^E_H)\,) \\ & \wedge (\,\forall\, l \in \mathsf{Link}(G_r) : \mathsf{Rel}^{D,E}_{L,H}(l)(s^D, s^E_H)\,)\end{aligned}$$

where Inv^E_H is the invariant of Prog^E_H (Definition (4)) and $\mathsf{Rel}^{D,E}_{N,H}(i)$ (respectively, $\mathsf{Rel}^{D,E}_{L,H}(l)$) essentially expresses the local state of node i in G (link l in G_r) in Prog^D in terms of the state of Prog^E_H (note that a truly functional relation from s^E_H to s^D cannot be used because most state variables of Prog^D have arbitrary initial and final values):

$$\begin{aligned}
&\mathsf{Rel}^{D,E}_{N,H}(i)((pc, par, acc, cnt, mq), (T_H, O_H)) \;\triangleq \\
&\quad \mathbf{if}\; i \in \mathsf{Node}(T_H) \wedge \mathtt{Send}(i) \notin O_H \;\mathbf{then} \\
&\qquad (\, pc(i) = \mathtt{Busy}\,) \wedge (\, par(i) = \mathsf{Par}(T_H)(i)\,) \wedge \\
&\qquad \mathbf{let}\; J = \{ j \mid \mathtt{Recv}'(i,j) \in O_H \} \;\mathbf{in} \\
&\qquad\; (\, cnt(i) = |\mathsf{Nbr}(G_r)(i) \setminus (\{\mathsf{Par}(T_H)(i)\} \cup J)|\,) \wedge \\
&\qquad\; (\, acc(i) = wt(i) \oplus \textstyle\sum j \in \mathsf{Kid}(T_H)(i) \cap J : \textstyle\sum k \in \mathsf{Desc}(T_H)(j) : wt(k)\,) \\
&\quad \mathbf{else} \\
&\qquad (\, pc(i) = \mathtt{Idle}\,)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Rel}^{D,E}_{L,H}(i,j)((pc, par, acc, cnt, mq), (T_H, O_H)) \;\triangleq \\
&\quad \mathbf{if}\; \mathtt{Send}(i) \in O_H \wedge (j = \mathsf{Par}(T_H)(i)) \wedge \mathtt{Recv}'(j,i) \notin O_H \\
&\qquad \mathbf{then}\; (\, mq(i,j) = \langle\!\langle \textstyle\sum k \in \mathsf{Desc}(T_H)(i) : wt(k) \rangle\!\rangle\,) \;\mathbf{else} \\
&\quad \mathbf{if}\; i \in \mathsf{Node}(T_H) \wedge \{i,j\} \notin \mathsf{Edge}(T_H) \wedge \mathtt{Recv}'(j,i) \notin O_H \\
&\qquad \mathbf{then}\; (\, mq(i,j) = \langle\!\langle \mathsf{O} \rangle\!\rangle\,) \\
&\qquad \mathbf{else}\; (\, mq(i,j) = \langle\!\langle\; \rangle\!\rangle\,)
\end{aligned}$$

The proof of $\mathsf{FSim}(\mathsf{Rel}^{D,E}_{H})(\mathsf{Prog}^D)(\mathsf{Prog}^E_H)$ (i.e., **FS1**–**FS3** of Section A.5) is omitted due to space limitations; they are tedious but not difficult.

To translate properties (5), (6), and (10) of Prog^E_H into the desired properties (1), (2), and (3) of Prog^D, we need to prove the following translation relations:

$$\begin{aligned}
(\,\Box\, \mathsf{Inv}^E_H\,) \;& \{\!\!\{ \Box(\mathsf{Rel}^{D,E}_{H})^{-1} \}\!\!\} \;(\,\Box(\mathsf{Done}^D \Rrightarrow \mathsf{Final}^D)\,) \\
(\,\Box(\mathsf{Done}^E_H \Rrightarrow \Box\, \mathsf{Done}^E_H)\,) \;& \{\!\!\{ \Box(\mathsf{Rel}^{D,E}_{H})^{-1} \}\!\!\} \;(\,\Box(\mathsf{Done}^D \Rrightarrow \Box\, \mathsf{Done}^D)\,) \\
(\,\Diamond \mathsf{Done}^E_H\,) \;& \{\!\!\{ \Box(\mathsf{Rel}^{D,E}_{H})^{-1} \}\!\!\} \;(\,\Diamond \mathsf{Done}^D\,)
\end{aligned}$$

which can be reduced, using the Reduction Lemmas in the same manner as illustrated by the proof of (24) in Section A.6, to the following three proof obligations:

$$\begin{aligned}
&(\mathsf{Inv}^E_H) \;\{\!\!\{ (\mathsf{Rel}^{D,E}_{H})^{-1} \}\!\!\} \;(\mathsf{Done}^D \Rrightarrow \mathsf{Final}^D) \\
&(\mathsf{Done}^E_H) \;\{\!\!\{ (\mathsf{Rel}^{D,E}_{H})^{-1} \}\!\!\} \;(\mathsf{Done}^D) \\
&(\mathsf{Done}^D) \quad \{\!\!\{ \mathsf{Rel}^{D,E}_{H} \}\!\!\} \quad (\mathsf{Done}^E_H)
\end{aligned}$$

all of which can be proved using the definition of translation (16) directly; again, the proofs are omitted due to space limitations. Finally, Theorem 2 of Section A.6 (i.e., the precise statement of Figure 1) is used to complete the proof.

Note that, by existentially quantifying away the "event state" s^E_H, we can obtain an invariant Inv^D of Prog^D from the simulation relation $\mathsf{Rel}^{D,E}_H$:

$$\mathsf{Inv}^D(s^D) \triangleq \exists\, s^E_H : \mathsf{Rel}^{D,E}_H(s^D, s^E_H)$$

Also note that the proof of invariance of Inv^D using **I1**–**I3** of Section A.3 (with $J = K = \mathsf{Inv}^D$) is essentially the same as **FS1** and **FS2** in the simulation proof for $\mathsf{Rel}^{D,E}_H$. So our method can be viewed as a generalization of the traditional invariant method [10].

5 Another Event View of DSum

An alternative event view of DSum is formalized by the automaton Prog^E_P shown in Figure 6. Prog^E_P is almost identical to Prog^E_H except that, whereas Prog^E_H enlarges T_H edge by edge until T_H becomes a spanning tree of G_r, Prog^E_P sets T_P to an arbitrary spanning tree of G_r at the beginning and thenceforth keeps T_P unchanged. Consequently, the occurrences of $\mathtt{Recv}(i, j)$ actions cannot be deduced from T_P and must be recorded in O_P. Note that saying $\mathsf{Recv}^E_P(i)(T_P, O_P)$ in Prog^E_P is equivalent to saying $i \in \mathsf{Node}(T_H)$ in Prog^E_H.

Prog^E_H and Prog^E_P differ in when "essential" (i.e., not related to the ordering of concurrent events) nondeterministic choices are made: Prog^E_H makes such choices one by one in the course of execution, while Prog^E_P makes them all in one stroke at the very beginning and is essentially deterministic thenceforth. So T_P is a prophecy variable in the sense of [1]. Prog^E_P is conceptually simpler and, as shown below, slightly easier to reason about than Prog^E_H. However, this apparent simplicity is deceptive, since the simulation proof relating Prog^E_P to Prog^D is more complex than that relating Prog^E_H to Prog^D. The reason is that there is neither a forward nor a backward simulation from Prog^D to Prog^E_P, so an intermediate program Π must be devised such that there are a forward simulation from Prog^D to Π and a backward simulation from Π to Prog^E_P (see [13]). We will use Prog^E_H as Π.

Prog^E_P has a simple invariant which says that T_P is always a spanning tree of G_r and whenever an event has occurred, its cause must be true:

$$\mathsf{Inv}^E_P(T_P, O_P) \triangleq \mathsf{Tree}(T_P) \land (\mathsf{Node}(T_P) = \mathsf{Node}(G_r)) \land \forall\, ev \in O_P : \mathsf{Cause}^E_P(ev)(T_P, O_P)$$

The invariance of Inv^E_P is easy to prove and states that:

$$\mathsf{Prog}^E_P \models \Box\, \mathsf{Inv}^E_P \tag{11}$$

The termination of Prog^E_P is signified by the occurrence of $\mathtt{Send}(r')$:

$$\mathsf{Done}^E_P(T_P, O_P) \triangleq \mathtt{Send}(r') \in O_P$$

Clearly, once an event is added to O_P, it stays in O_P forever. In particular, once Prog^E_P terminates, it stays that way forever:

$$\mathsf{Prog}^E_P \models \Box(\mathsf{Done}^E_P \Rightarrow \Box\, \mathsf{Done}^E_P) \tag{12}$$

The basic liveness property of Prog^E_P is:

$$\forall\, ev \in \mathsf{Event}^E_P : \ \mathsf{Prog}^E_P \models \mathsf{Cause}^E(ev) \leadsto ev \in O_P \tag{13}$$

which is easy to prove. Since T_P is always a spanning tree of G_r, (13) can be used to prove, in a manner similar to how (8) is used to prove (10) for Prog^E_H, that Prog^E_P must eventually terminate:

$$\mathsf{Prog}^E_P \models \Diamond\, \mathsf{Done}^E_P \tag{14}$$

When compared with the proof of Prog^E_H in Section 3, the above proof is only slightly simpler, in that there is no need to prove that T_P will eventually become a spanning tree of G_r, as in the case of T_H.

EVENTS:

$\mathsf{Event}_P^E \triangleq \{ \mathtt{Recv}(i,j), \mathtt{Recv}'(i,j), \mathtt{Send}(i) \mid i \in \mathsf{Node}(G) \wedge j \in \mathsf{Nbr}(G_r)(i) \}$

STATE VARIABLES:

$T_P \in \mathsf{Subgraph}(G_r)$ (* Tree that will be constructed *)
$O_P \in \mathcal{P}(\mathsf{Event}_P^E)$ (* Events occurred so far *)

INITIAL CONDITION:

$\mathsf{Tree}(T_P) \wedge (\mathsf{Node}(T_P) = \mathsf{Node}(G_r)) \wedge (O_P = \emptyset)$

AUXILIARY DEFINITION:

$\mathsf{Recv}_P^E(i)(T_P, O_P) \triangleq (i = r) \vee \mathtt{Recv}(i, \mathsf{Par}(T_P)(i)) \in O_P$

CAUSALITY:

$\mathsf{Cause}_P^E(\mathtt{Recv}(i,j))(T_P, O_P) \triangleq$
$\mathsf{Recv}_P^E(j)(T_P, O_P) \wedge (j = \mathsf{Par}(T_P)(i))$

$\mathsf{Cause}_P^E(\mathtt{Recv}'(i,j))(T_P, O_P) \triangleq$
$(\mathsf{Recv}_P^E(i)(T_P, O_P) \wedge j \in \mathsf{Kid}(T_P)(i) \wedge \mathtt{Send}(j) \in O_P) \vee$
$(\mathsf{Recv}_P^E(i)(T_P, O_P) \wedge \mathsf{Recv}_P^E(j)(T_P, O_P) \wedge \{i, j\} \notin \mathsf{Edge}(T_P))$

$\mathsf{Cause}_P^E(\mathtt{Send}(i))(T_P, O_P) \triangleq$
$\mathsf{Recv}_P^E(i)(T_P, O_P) \wedge \forall j \in \mathsf{Nbr}(G_r)(i) \setminus \{\mathsf{Par}(T_P)(i)\} : \mathtt{Recv}'(i,j) \in O_P$

ACTIONS:

ev

Precondition:

$ev \in \mathsf{Event}_P^E \wedge \mathsf{Cause}_P^E(ev)(T_P, O_P) \wedge ev \notin O_P$

Effect:

$O_P := O_P \cup \{ev\}$;

Fig. 6. Prog_P^E: Event view of DSum with a prophecy variable

The following relation $\mathsf{Rel}_{H,P}^E$ is a backward simulation from Prog_H^E to Prog_P^E:

$$\begin{aligned}
&\mathsf{Rel}_{H,P}^E((T_H, O_H), (T_P, O_P)) \triangleq \\
&\mathsf{Inv}_H^E(T_H, O_H) \wedge \mathsf{Tree}(T_P) \wedge (\mathsf{Node}(T_P) = \mathsf{Node}(G_r)) \wedge (\mathsf{Edge}(T_P) \supseteq \mathsf{Edge}(T_H)) \wedge \\
&(O_P = O_H \cup \{\mathtt{Recv}(i, \mathsf{Par}(T_H)(i)) \mid i \in \mathsf{Node}(T_H) \setminus \{r\}\})
\end{aligned} \tag{15}$$

To prove $\mathsf{BSim}(\mathsf{Rel}_{H,P}^E)(\mathsf{Prog}_H^E)(\mathsf{Prog}_P^E)$, we need to prove **BS1**–**BS4** of Section A.5 with the aid of an invariant of Prog_H^E. A suitable invariant is, of course, Inv_H^E (Definition (4)). Due to space limitations, we outline only the proof of **BS4**, which states that for any (T_H, O_H) satisfying Inv_H^E, there exist at least one and at most finitely many (T_P, O_P) such that $\mathsf{Rel}_{H,P}^E((T_H, O_H), (T_P, O_P))$ holds. That there exists at least one such (T_P, O_P) is because T_H is a subtree of G_r and hence can be extended into a spanning tree T_P of G_r. That there exist at most finitely many such (T_P, O_P) is because G_r is finite and hence has only finitely many spanning trees. Note that, in

(15), O_P is completely determined by T_H and O_H.

To translate properties (11), (12), and (14) of Prog_P^E into properties (5), (6), and (10) of Prog_H^E, we need to prove the following translation relations:

$$(\,\Box\,\mathsf{Inv}_P^E\,)\ \{\!\{\Box(\mathsf{Rel}_{H,P}^E)^{-1}\}\!\}\ (\,\Box\,\mathsf{Inv}_H^E\,)$$

$$(\,\Box(\mathsf{Done}_P^E \Rrightarrow \Box\,\mathsf{Done}_P^E)\,)\ \{\!\{\Box(\mathsf{Rel}_{H,P}^E)^{-1}\}\!\}\ (\,\Box(\mathsf{Done}_H^E \Rrightarrow \Box\,\mathsf{Done}_H^E)\,)$$

$$(\,\Diamond\mathsf{Done}_P^E\,)\ \{\!\{\Box(\mathsf{Rel}_{H,P}^E)^{-1}\}\!\}\ (\,\Diamond\mathsf{Done}_H^E\,)$$

which can be reduced, using the Reduction Lemmas, to the following three proof obligations:

$$(\mathsf{Inv}_P^E)\ \{\!\{(\mathsf{Rel}_{H,P}^E)^{-1}\}\!\}\ (\mathsf{Inv}_H^E)$$

$$(\mathsf{Done}_P^E)\ \{\!\{(\mathsf{Rel}_{H,P}^E)^{-1}\}\!\}\ (\mathsf{Done}_H^E)$$

$$(\mathsf{Done}_H^E)\ \ \{\!\{\mathsf{Rel}_{H,P}^E\}\!\}\ \ (\mathsf{Done}_P^E)$$

all of which can be trivially proved using the definition of translation (16) directly. It was already shown in the last section that properties (5), (6), and (10) of Prog_H^E can be translated into the desired properties (1), (2), and (3) of Prog^D. Finally, Theorem 2 of Section A.6 is used to complete the proof.

6 Concluding Remarks

In this paper we showed how to use operational intuition about events and causality to guide and structure the assertional proof of a simple distributed algorithm. We feel that our proof is better motivated and more structured than Vaandrager's proof of the same algorithm [20], which was constructed without the help of the notions of events and causality. The readers are invited to compare the two proofs. We believe that our method can be scaled up to produce manageable proofs of algorithms as complex as the minimum spanning tree algorithm of Gallager, Humblet, and Spira [6]. But, of course, only actual experience will be able to tell whether or not our belief is justified.

We also showed that the notions of events and causality can be used to motivate proofs based both on forward simulation/history variables and on backward simulation/prophecy variables. Although the event view Prog_P^E that uses a prophecy variable is simpler than the event view Prog_H^E that uses only history variables, the proof relating Prog_P^E to the distributed view Prog^D is more complex than the one relating Prog_H^E to Prog^D; indeed, the former contains the latter as a part. The reason for this is that a backward simulation proof needs an invariant of the concrete program separate from the simulation relation, while a forward simulation proof does not. Since the very point of simulation techniques lies in the possibility of "working abstractly", the need for a separate invariant of the concrete program makes the benefit of backward simulation dubious, when a forward simulation exists.

A Appendix

A.1 Sets

For any sets S and T, $|S|$ is the cardinality of S, $\mathcal{P}(S)$ is the power set of S, S^* is the set of finite sequences on S, $S{\times}T$ is the cartesian product of S and T, and $S\rightarrow T$ is the set of functions from S to T. Sequences are enclosed by $\langle\!\langle \cdots \rangle\!\rangle$, sequence concatenation is denoted by $\frown$, the head of a sequence s is denoted by $\mathsf{head}(s)$, and the tail by $\mathsf{tail}(s)$. For example, $\langle\!\langle 1,2,3\rangle\!\rangle \frown \langle\!\langle 4,5\rangle\!\rangle = \langle\!\langle 1,2,3,4,5\rangle\!\rangle$, $\mathsf{head}(\langle\!\langle 1,2,3\rangle\!\rangle) = 1$, and $\mathsf{tail}(\langle\!\langle 1,2,3\rangle\!\rangle) = \langle\!\langle 2,3\rangle\!\rangle$. $\mathcal{N} \triangleq \{0,1,2,\ldots\}$ is the set of natural numbers. $\mathcal{W}$ is a set of "weights" on which there is an "addition" operation $\oplus$ that is associative and commutative and has an identity O. Therefore, for any *finite* set S and any function $f \in S\rightarrow\mathcal{W}$, the "sum" of $f(x)$'s for all $x \in S$, denoted $\sum x \in S : f(x)$, is well-defined. Identifiers written in `typewriter` font are literals, which are all distinct and may have arguments.

A.2 Graphs

In this paper, every graph is *undirected* and *finite* and may have at most one edge between two nodes. Hence an edge connecting nodes i and j can be identified with the set $\{i, j\}$. A *link* is a pair (i, j) such that $\{i, j\}$ is an edge. Let G be a graph. The sets of nodes, edges, links, and subgraphs of G are denoted by $\mathsf{Node}(G)$, $\mathsf{Edge}(G)$, $\mathsf{Link}(G)$, and $\mathsf{Subgraph}(G)$, respectively. For any node i in G, $\mathsf{Nbr}(G)(i) \triangleq \{j \in \mathsf{Node}(G) \mid \{i, j\} \in \mathsf{Edge}(G)\}$ is the set of *neighbors* of i in G. We say that G is a *tree*, denoted $\mathsf{Tree}(G)$, iff there is exactly one path between any two nodes in G, where a path can pass through an edge at most once.

Let T be a tree and r (for "root"), i, and j be nodes in T. We say that i is a *descendent* of j in T (denoted $i \in \mathsf{Desc}(T)(j)$) iff the (unique) path from i to r in T passes through j. Note that $j \in \mathsf{Desc}(T)(j)$. We say that i is a *kid* of j in T (denoted $i \in \mathsf{Kid}(T)(j)$), or equivalently, that j is the *parent* of i in T (denoted $j = \mathsf{Par}(T)(i)$), iff $i \neq r$, $\{i, j\} \in \mathsf{Edge}(G)$ and $i \in \mathsf{Desc}(T)(j)$.

A.3 Automata

In this paper, programs are represented by automata.

An *automaton* is a quadruple $\Pi = (S, I, A, T)$, where S is a set of *states*, $I \subseteq S$ is a nonempty set of *initial states*, A is a set of *actions* containing a special action $\mathtt{Stutter}$, and $T \subseteq S \times A \times S$ is a set of *transitions* such that $\forall\, s, s' \in S : (s, \mathtt{Stutter}, s') \in T \Leftrightarrow (s = s')$. An action $a \in A$ is *enabled* at a state $s \in S$ iff there is an a-transition from s: $\mathsf{En}(a)(s) \triangleq \exists\, s' \in S : (s, a, s') \in T$. An *execution* of Π is a (finite or infinite) sequence of alternating states and actions (which ends in a state if finite), $\langle\!\langle s_0, a_0, s_1, a_1, s_2, \cdots, s_n, a_n, s_{n+1}, \cdots \rangle\!\rangle$, such that:

E1. $s_0 \in I$ **E2.** $\forall\, n \in \mathcal{N} : (s_n, a_n, s_{n+1}) \in T$

An infinite execution is *fair* iff:

E3. $\forall\, a \in A \setminus \{\mathtt{Stutter}\} : (\, \exists^\infty n \in \mathcal{N} : (a_n = a)\,) \vee (\, \exists^\infty n \in \mathcal{N} : \neg \mathsf{En}(a)(s_n)\,)$
where "$\exists^\infty$" means "there exist infinitely many".

A terminating computation can be modeled by an infinite execution in which the terminal state is repeated *ad infinitum* by the stuttering action $\mathtt{Stutter}$. Hence there is no need to include finite executions in the *semantics* of Π, $[\![\, \Pi \,]\!]$, which is defined to be the set of all fair infinite executions of Π. The fairness assumption **E3** rules out those executions that settle into perpetual stuttering prematurely, thus ensuring progress. The idea of using stuttering is borrowed from [8, 11].

A state s of Π is *reachable* iff it is the last state of some finite execution of Π; the set of reachable states of Π is denoted by $\mathcal{R}(\Pi)$. In practice, it is very hard to characterize $\mathcal{R}(\Pi)$ for nontrivial Π. Instead, one works with approximations: a set $J \subseteq S$ is an *invariant* of Π iff $\mathcal{R}(\Pi) \subseteq J$. To prove that J is an invariant of Π, it is sufficient to find a $K \subseteq S$ such that:

I1. $I \subseteq K$ **I2.** $\forall (s, a, s') \in T : s \in K \Rightarrow s' \in K$ **I3.** $K \subseteq J$

A simple inductive argument on the lengths of finite executions shows that **I1**–**I3** imply the invariance of J (and K as well).

Our automata are essentially I/O automata [12, 20] with the distinction between input, output, and internal actions removed. To specify automata, we use a notation borrowed from the I/O automata literature [12, 20] in which the states are specified by a set of *state variables* and the actions are specified by their *preconditions* (i.e., enabling conditions) and *effects*. In the effect part, a command of the form $x :=\ ?$ is executed by setting x to a nondeterministically chosen value of the appropriate type. Also, the $\mathtt{Stutter}$ action will not be explicitly specified.

A.4 Temporal Logic

In this paper, properties of programs are expressed in temporal logic.

Let X be a set. A *property* A on X is identified with the set of elements of X that satisfy A. For any $x \in X$, the expressions "x satisfies A", "$A(x)$", and "$x \in A$" are used interchangeably. For any family $\{A_i \subseteq X \mid i \in I\}$ of properties on X, the intersection $\bigcap i \in I : A_i$ (respectively, the union $\bigcup i \in I : A_i$) can be viewed as the (potentially infinitary) *conjunction* (*disjunction*) of all A_i's. For any properties A and B on X, $\sim A \triangleq X \backslash A$ is the *negation* of A, and $A \Rrightarrow B \triangleq (\sim A) \cup B$ is the *implication* from A to B. Note that $\sim A$ and $A \Rrightarrow B$ depend on the underlying set X, which in this paper is always clear from context.

Properties on states are called *conditions* and properties on executions are called *temporal properties*. All temporal properties used in this paper are constructed from conditions using the logical operators defined in the last paragraph and the temporal operators defined below:

$$\begin{aligned}
\mathsf{PS}(C)\langle\!\langle s_0, a_0, s_1, \cdots \rangle\!\rangle &\triangleq C(s_0) \\
(\Box P)\langle\!\langle s_0, a_0, s_1, \cdots \rangle\!\rangle &\triangleq \forall\, n \in \mathcal{N} : P\langle\!\langle s_n, a_n, s_{n+1}, \cdots \rangle\!\rangle \\
(\Diamond P)\langle\!\langle s_0, a_0, s_1, \cdots \rangle\!\rangle &\triangleq \exists\, n \in \mathcal{N} : P\langle\!\langle s_n, a_n, s_{n+1}, \cdots \rangle\!\rangle \\
P \leadsto Q &\triangleq \Box(P \Rrightarrow \Diamond Q)
\end{aligned}$$

where C is a condition and P and Q are temporal properties. PS *coerces* a condition into a temporal property by evaluating it at the first states of executions; $\Box P$ (respectively, $\Diamond P$) means that P *always* (*sometime*) holds; $P \leadsto Q$ (P *leads to* Q) means that whenever P holds, Q holds then or later. Whenever there is no confusion, we will drop PS from expressions; for instance, instead of writing $\mathsf{PS}(C) \leadsto \mathsf{PS}(D)$, we will simply write $C \leadsto D$, for conditions C and D.

An automaton Π *satisfies* a temporal property P iff every execution of Π satisfies P:

$$\Pi \models P \triangleq [\![\, \Pi \,]\!] \subseteq P$$

A.5 Simulation of Programs

Let $\Pi^\flat = (S^\flat, I^\flat, A, T^\flat)$ and $\Pi^\sharp = (S^\sharp, I^\sharp, A, T^\sharp)$ be respectively a "concrete" and an "abstract" automata that share a common set A of actions. Let $R \subseteq S^\flat \times S^\sharp$ be a relation. We say R is a *forward simulation* from $\Pi^\flat$ to $\Pi^\sharp$, denoted $\mathsf{FSim}(R)(\Pi^\flat)(\Pi^\sharp)$, iff:

FS1. $\forall\, s^\flat \in I^\flat : \exists\, s^\sharp \in I^\sharp : (s^\flat, s^\sharp) \in R$
FS2. $\forall\, a \in A : \forall\, (s^\flat, s^\sharp) \in R : \forall\, t^\flat : (s^\flat, a, t^\flat) \in T^\flat \Rightarrow$
$\exists\, t^\sharp : (s^\sharp, a, t^\sharp) \in T^\sharp \wedge (t^\flat, t^\sharp) \in R$
FS3. $\forall\, a \in A : \forall\, (s^\flat, s^\sharp) \in R : \mathsf{En}(a)(s^\sharp) \Rightarrow \mathsf{En}(a)(s^\flat)$

We say R is a *backward simulation* from $\Pi^\flat$ to $\Pi^\sharp$, denoted $\mathsf{BSim}(R)(\Pi^\flat)(\Pi^\sharp)$, iff:

BS1. $\forall\, (s^\flat, s^\sharp) \in R : s^\flat \in I^\flat \Rightarrow s^\sharp \in I^\sharp$
BS2. $\forall\, a \in A : \forall\, (t^\flat, t^\sharp) \in R : \forall\, s^\flat : (s^\flat, a, t^\flat) \in T^\flat \wedge s^\flat \in \mathcal{R}(\Pi^\flat) \Rightarrow$
$\exists\, s^\sharp : (s^\sharp, a, t^\sharp) \in T^\sharp \wedge (s^\flat, s^\sharp) \in R$
BS3. $\forall\, a \in A : \forall\, (s^\flat, s^\sharp) \in R : \mathsf{En}(a)(s^\sharp) \Rightarrow \mathsf{En}(a)(s^\flat)$
BS4. $\forall\, s^\flat \in \mathcal{R}(\Pi^\flat) : \exists^{<\infty} s^\sharp : (s^\flat, s^\sharp) \in R$

where "$\exists^{<\infty}$" means "there exist at least one and at most finitely many".

Lemma 1. $\mathsf{Sim}(R)(\Pi^\flat)(\Pi^\sharp) \Rightarrow \forall\, x^\flat \in [\![\, \Pi^\flat \,]\!] : \exists\, x^\sharp \in [\![\, \Pi^\sharp \,]\!] : (x^\flat, x^\sharp) \in \Box R$
where $\mathsf{Sim} \in \{\mathsf{FSim}, \mathsf{BSim}\}$ and:

$$(\langle\!\langle s^\flat_0, a_0, s^\flat_1, \cdots \rangle\!\rangle, \langle\!\langle s^\sharp_0, a_0, s^\sharp_1, \cdots \rangle\!\rangle) \in \Box R \;\Leftrightarrow\; \forall\, n \in \mathcal{N} : (s^\flat_n, s^\sharp_n) \in R$$

Due to space limitations, the proof of Lemma 1 is omitted; essentially the same theorems can be found in [13]. Also, Lemma 1 is still valid if, in **BS2** and **BS4**, $\mathcal{R}(\Pi^\flat)$ is replaced by any invariant of $\Pi^\flat$.

A.6 Translation of Properties

Let X and Y be two arbitrary sets and $L \subseteq X \times Y$ be a relation. We say a property $A \subseteq X$ can be *translated* into another property $B \subseteq Y$ via L iff for any x and y related by L, if x satisfies A then y satisfies B:

$$A \,\{\!\!\{L\}\!\!\}\, B \;\triangleq\; \forall (x, y) \in L : A(x) \Rightarrow B(y) \tag{16}$$

The $X \,\{\!\!\{L\}\!\!\}\, Y$ notation is inspired by the (original) notation for Hoare triples [7]; indeed, if L is the relational semantics of a command, then $X \,\{\!\!\{L\}\!\!\}\, Y$ *is* a Hoare triple. For later use, we define the *inverse* $L^{-1} \subseteq Y \times X$ by $(y, x) \in L^{-1} \Leftrightarrow (x, y) \in L$.

Let $\Pi^\flat$ and $\Pi^\sharp$ be as specified in Section A.5, $P^\flat$ and $P^\sharp$ be temporal properties on the executions of $\Pi^\flat$ and $\Pi^\sharp$ respectively, and $R \subseteq S^\flat \times S^\sharp$ be a relation. It follows immediately from Lemma 1 and Definition (16) that:

Theorem 2. $\mathsf{Sim}(R)(\Pi^\flat)(\Pi^\sharp) \wedge P^\sharp \,\{\!\!\{\Box(R^{-1})\}\!\!\}\, P^\flat \Rightarrow (\, \Pi^\sharp \models P^\sharp \Rightarrow \Pi^\flat \models P^\flat \,)$
where $\mathsf{Sim} \in \{\mathsf{FSim}, \mathsf{BSim}\}$. (This is the precise statement of Figure 1.)

Theorem 2 would be virtually useless if translations of the form $P \,\{\!\!\{\Box R\}\!\!\}\, Q$ could only be proved by appealing to Definition (16) directly, since P and Q can be complex temporal properties. Fortunately, the proof of a translation relation between two complex properties *of the same form* can be reduced to the proofs of translation relations between their corresponding constituents:

Theorem 3. [Reduction Lemmas]

$$B \,\{\!\!\{L^{-1}\}\!\!\}\, A \;\Rightarrow\; (\sim A) \,\{\!\!\{L\}\!\!\}\, (\sim B) \tag{17}$$
$$B_1 \,\{\!\!\{L^{-1}\}\!\!\}\, A_1 \,\wedge\, A_2 \,\{\!\!\{L\}\!\!\}\, B_2 \;\Rightarrow\; (A_1 \Rrightarrow A_2) \,\{\!\!\{L\}\!\!\}\, (B_1 \Rrightarrow B_2) \tag{18}$$
$$(\forall\, i \in I : A_i \,\{\!\!\{L\}\!\!\}\, B_i) \;\Rightarrow\; (\textstyle\bigcap i \in I : A_i) \,\{\!\!\{L\}\!\!\}\, (\textstyle\bigcap i \in I : B_i) \tag{19}$$
$$(\forall\, i \in I : A_i \,\{\!\!\{L\}\!\!\}\, B_i) \;\Rightarrow\; (\textstyle\bigcup i \in I : A_i) \,\{\!\!\{L\}\!\!\}\, (\textstyle\bigcup i \in I : B_i) \tag{20}$$
$$C \,\{\!\!\{R\}\!\!\}\, D \;\Rightarrow\; (\mathsf{PS}(C)) \,\{\!\!\{\Box R\}\!\!\}\, (\mathsf{PS}(D)) \tag{21}$$
$$P \,\{\!\!\{\Box R\}\!\!\}\, Q \;\Rightarrow\; (\Box P) \,\{\!\!\{\Box R\}\!\!\}\, (\Box Q) \tag{22}$$
$$P \,\{\!\!\{\Box R\}\!\!\}\, Q \;\Rightarrow\; (\Diamond P) \,\{\!\!\{\Box R\}\!\!\}\, (\Diamond Q) \tag{23}$$

where C and D are conditions, P and Q are temporal properties, A, B, A_i's, and B_i's are arbitrary properties, and I is an arbitrary index set. None of (17)–(23) is hard to prove.

As an example of the application of the Reduction Lemmas, the following theorem (where C and D are conditions):

$$D_1 \,\{\!\!\{R^{-1}\}\!\!\}\, C_1 \,\wedge\, C_2 \,\{\!\!\{R\}\!\!\}\, D_2 \;\Rightarrow\; (C_1 \leadsto C_2) \,\{\!\!\{\Box R\}\!\!\}\, (D_1 \leadsto D_2) \tag{24}$$

can be proved using (22), (18), (23), and (21) (in that order). So the task of proving the consequent of (24) can be reduced to that of proving the antecedents of (24), which involves no temporal reasoning. In general, if P and Q are two temporal properties *of the same form* whose primitive constituents are conditions, then the task of proving $P \,\{\!\!\{\Box R\}\!\!\}\, Q$ can be reduced to proving translation relations between conditions and hence will not involve any temporal reasoning.

References

1. M. Abadi and L. Lamport, "The Existence of Refinement Mappings", *Theoretical Computer Science*, Vol. 82, No. 2, pp. 253–284, 1991.
2. J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (Ed.), *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1988.
3. K. Mani Chandy and Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

4. Ching-Tsun Chou, "Practical Use of the Notions of Events and Causality in Reasoning about Distributed Algorithms", UCLA CSD Report #940035, Oct. 1994. (Available via WWW at ⟨ftp://ftp.cs.ucla.edu/pub/chou/nil.ps⟩.)
5. Ching-Tsun Chou, "A Simple Treatment of Property Preservation via Simulation", working paper, Feb. 1995.
6. R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 1, pp. 66–77, Jan. 1983.
7. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, Vol. 12, No. 10, pp. 576–583, Oct. 1969.
8. Bengt Jonsson, "Compositional Specification and Verification of Distributed Systems", *ACM Trans. on Programming Languages and Systems*, Vol. 16, No. 2, pp. 259–303, Mar. 1994.
9. Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, Jul. 1978.
10. Leslie Lamport, "An Assertional Correctness Proof of a Distributed Algorithm", *Science of Computer Programming*, Vol. 2, pp. 175–206, 1982.
11. Leslie Lamport, "The Temporal Logic of Actions", *ACM Trans. on Programming Languages and Systems*, Vol. 16, No. 3, pp. 872–923, May 1994.
12. Nancy A. Lynch and Mark Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms", in [17], pp. 137–151.
13. Nancy A. Lynch and Frits W. Vaandrager, "Forward and Backward Simulations, Part I: Untimed Systems", Report CS-R9313, CWI, Amsterdam, Mar. 1993.
14. Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.
15. Antoni Mazurkiewicz, "Basic Notions of Trace Theory", in [2], pp. 285–363.
16. M. Nielsen, G. Plotkin, and G. Winskel, "Petri Nets, Event Structures and Domains, Part I", *Theoretical Computer Science*, Vol. 13, pp. 85–108, 1981.
17. *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, Aug. 1987.
18. Adrian Segall, "Distributed Network Protocols", *IEEE Trans. on Information Theory*, Vol. 29, No. 1, pp. 23–35, Jan. 1983.
19. M.W. Shields, "Concurrent Machines", *The Computer Journal*, Vol. 28, No. 5, pp. 449–465, 1985.
20. Frits W. Vaandrager, "Verification of a Distributed Summation Algorithm", Report CS-R9505, CWI, Amsterdam, Jan. 1995.

This article was processed using the LaTeX macro package with LLNCS style