A Logic of Events *

Mark Bickford
ORA

Robert L. Constable
Cornell University

February 10, 2003

Abstract

There is a well-established theory and practice for creating correct-by-construction functional programs by extracting them from constructive proofs of assertions of the form $\forall x : A.\exists y : B.R(x, y)$. There have been several efforts to extend this methodology to concurrent programs, say by using linear logic, but there is no practice and the results are limited.

In this paper we define a logic of events that justifies the extraction of *correct distributed processes* from constructive proofs that system specifications are achievable, and we describe an implementation of an *extraction process* in the context of constructive type theory. We show that a class of *message automata,* similar to IO automata and to active objects, are realizers for this logic. We provide a relative consistency result for the logic. We show an example of protocol derivation in this logic, and show how to embed temporal logics such as $TLA+$ in the event logic.

1 Introduction

The idea of creating functional programs that are *correct-by-construction* is old and well-studied [20, 9, 22, 19, 47, 52]. Several implementations by extraction have been built based on the concept of *proofs-as-programs* (e.g. Alf, MetaPRL, Nuprl, Coq, Lego), and many interesting examples are well-known, including solutions of Higman's lemma [51] and a recent program for Buchberger's Gröbner basis algorithm [57]. The extracted functional programs are called *realizers* for propositions. In this paper we deal with logics such as constructive type theory, in which all provable assertions have realizers.

For many years researchers have tried to extend this methodology to concurrent programs by extending the proofs-as-programs principle to something worthy of the name *proofs-as-processes* principle. In 1994 Samson Abramsky wrote an article [4] under this title in which linear logic was the basic logic and certain nondeterministic programs in [10] were considered as realizers. Robin Milner and his students also took up this challenge, and there are now a number of results along these lines [7, 49].

In this paper we look at a different approach to the problem. We aim to extract distributed systems from proofs of system specifications that arise in practice. The abstract realizers are called *message automata,* and they resemble the IO automata of Lynch and Tuttle [43], and the active objects of Chandy [15].

* This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research, under Grant N00014-01-1-0765, and by DARPA grant F30602-98-2-0198 and NSF grant CCR-0208536.

The specification language arose from our experience in describing and proving properties of implemented protocols in systems such as Ensemble [13, 12, 14, 31, 32, 37, 36, 41, 58], UAV [35], and MediaNet [54].
Our approach to presenting the logic is to follow Martin-Löf's discipline for type theory; that is, present the computation system first and then introduce types and logic as a way of classifying and making assertions about data. In our case it will be *assertions about distributed computations* of these automata which operate by sending messages and reacting to the receipt of messages. These computations give rise to an *event system* which is the computational model for our logic.

2 The Computation System

2.1 Message Automata

A message automaton is a nondeterministic state machine. Its actions are to *send* and *receive* messages, and to execute *internal state transitions.* Leaving aside more detailed type constraints, a message automaton will be characterized by three types: St, Act, and Msg, which are the states, actions and messages, respectively.
Message automata are elements of the following dependent record type:

$$\begin{array}{l} \{\ St, Act, Msg : Type; init : St; \\ \quad f : (Act + Msg) \to St \to St; \\ \quad send : (Act + Msg) \to St \to MsgList\ \} \end{array}$$

A possible computation is a stream of alternating *states, queues* and *events,* say

$$s_0, q_0, a_0, s_1, q_1, a_1, \ldots\ .$$

If event a_i is an internal action, then

$$s_{i+1} = f(a_i)(s_i) \text{ and } q_{i+1} = enq(send(a_i)(s_i)q_i).$$

If a_i is a message receive, then $s_{i+1} = f(a_i)(s_i)$, and

$$q_{i+1} = enq(send(a_i)(s_i)\ deq?(a_i, q_{i+1})),$$

where $deq?$ takes the received message from the queue.

2.2 Distributed Systems

Given a message automaton M, there may be many possible computations consistent with it. If there are no messages, then M can act like an ordinary nondeterministic automaton (finite or infinite state). We are interested in computations that arise from interaction with an environment which creates messages; typically the messages are sent by other automata. We consider only *fair computations,* in which every message that is sent will be received.
We focus on collections of message automata, say $M_1, M_2, \ldots, M_k$, that are connected in a network by the links. We assume that $Links$ forms a directed graph with M_i at the nodes. We speak of M_i as *located* at a node. Each link l has a *source* (src) and a *destination* (dst). Associated with each link is a list of messages that originate at the source and arrive at the destination. We call such a collection a *distributed system.*
Our execution model assumes that at each location there is a computation; that is a stream of alternating states and actions, and the links are message queues.

2.3 Possible Computations

The *possible computations* (or *worlds*) of a distributed system is a collection of computations at each node which are compatible. We define these in terms of an idealized global discrete progression of time indexed by the natural numbers $0, 1, 2, \ldots$. This notion of time will not be reflected in the logic.

Intuitively a possible computation arises as follows. At time 0 each M_i is in a designated initial state $s(i, 0)$. At time 1, if some M_i can take an action, it may advance to state $s(i, 1)$. Not all M_i that can act are required to take a step, but eventually there must be a time t at which it will move. If an action results in a send, say $< m, l >$, then m is added to the message queue from $src(l)$ to $dst(l)$, and the state is changed. If there is a message on l at time t, then a possible action is a receive at $dst(l)$, and the message is removed from the fifo queue.

The collective state of the distributed system is given by $s(i, t)$, the state of M_i at time $t \in \mathbb{N}$, and $a(i, t)$, the action taken at time t — which can be null. The collective state also keeps track of the messages sent by M_i at time t, $msg(i, t)$. This is a list of the message and the link on which it is sent. For convenience, we also have $link(l, t)$, the list of messages on link l before time t; all the receive actions on l before time t, $rcvs(l, t)$; and all the sends before t, $sends(l, t)$.

The lists $link(l, t)$, $rcvs(l, t)$ and $sends(l, t)$ form queues. We can test for emptiness, find the head, know the length, etc. We assume that the links are reliable (no message is lost) and fifo. We assume that the computation is *fair*, that is, for every queue, infinitely often it is either empty or a receive action occurs at its destination automaton.

2.4 Refinements of the Automata and Frame Conditions

We refine the definition of message automata by being more detailed about the structure of the state and by typing the operations. For example, the automaton will declare its state variables, say x_j. By convention the only changes to state variables are given by actions that explicitly mention those variables. Since we want each clause of the definition of an automaton to be meaningful on its own, we can't rely on this convention, so we have to *state explicitly exactly which actions effect which variables.* We do this with a *frame* condition; $frame(x)$ is a list of all the actions that can change variable x. We do the same for message sends; $sframe(l)$ will list all actions that can send on link l, and moreover, we will refine the notion of messages to include tags, so $sframe$ will have inputs $< tag, link >$.

2.5 Typing and Examples

Message automata are formalized in the type theory on which the logic of events is based. Our investigations started with such a formalization [11, 21]. We leave these details to the examples that appear later.

3 Event Systems

We want an abstract model that can capture the observable features of a distributed system. The fundamental types are *locations* and *events* which we can think of as space and time coordinates, as in Lamport [38]. Information is stored at a location as the value of a state variable or an *observable* and information is passed from one location to another along *links* in the form of *messages*.

A message will consist of a link, a tag, and a value whose type may depend on the link and the tag.

$$\begin{array}{rcl}
Msg(Lnk, Tag, M) & \equiv & l : Lnk \times tg : Tag \times M(l, tg) \\
msg(l, tg, v) & \equiv & \langle l, t, v \rangle \\
mlnk(msg(l, tg, v)) & \equiv & l \\
mtag(msg(l, tg, v)) & \equiv & tg \\
mval(msg(l, tg, v)) & \equiv & v \\
haslink(l, ms) & \equiv & (mlnk(ms) = l) \\
hastag(tg, ms) & \equiv & (mtag(ms) = tg) \\
Msg_l(Lnk, Tag, M) & \equiv & \{ms : Msg(Lnk, Tag, M)) \mid haslink(l, ms)\} \\
onlink(l, mss) & \equiv & [ms \in mss \mid haslink(l, ms)] \\
onlinktagged(l, tg, mss) & \equiv & [ms \in mss \mid haslink(l, ms) \wedge hastag(tg, ms)]
\end{array}$$

Every event will have a kind, a value, and a location. So an event is a point in spacetime. The receipt of a message $msg(l, tg, v)$ will be one kind of event, and there will also be local events whose kinds are in a type of action names A.

$$\begin{array}{rcl}
Knd(Lnk, Tag, A) & \equiv & Lnk \times Tag + A \\
isrcv(k) & \equiv & isl(k) \\
islocal(k) & \equiv & isr(k) \\
rcv_l(tg) & \equiv & inl\langle l, tg \rangle \\
local(a) & \equiv & inr(a) \\
lnk(rcv_l(tg)) & \equiv & l \\
tag(rcv_l(tg)) & \equiv & tg \\
act(local(a)) & \equiv & a \\
kindcase(f, g, k) & \equiv & \textbf{if } islocal(k) \textbf{ then } f(act(k)) \textbf{ else } g(lnk(k), tag(k))
\end{array}$$

An *event system* is a structure consisting types, operations, and axioms. There are six types E, Loc, Lnk, X, A, Tag for the events, locations, links, observables, local action kinds, and message tags. These must all be discrete types – equality on each type is decidable. The operations include src and dst which assign source and destination location to the links, forming a graph structure on the locations and links. Operations loc, $kind$, and val extract the location, kind, and value from an event. Operations **when**, **after**, and **initially** observe the values of the observables at the points in spacetime. Messages must originate at some point in spacetime, and the operations $sends$, $sender$, and $index$ define this structure. The $sends(l, e)$ of an event e on link l will be a list of messages on that link that originate at e. We build the semantics of message delivery into our model in a way that makes every link into a reliable fifo channel. Thus every message is eventually received, and for a receive event e', the operations $sender(e')$ and $index(e')$ will provide the originator of the message received and the index of that message in the list that originated there. The temporal order structure on our spacetime is provided by two orderings on events $<_{loc}$ and $\prec$, as in Lamport [38]. The local ordering $<_{loc}$ is a total, discrete, well-founded, linear ordering on events with the same location. So, at each location, if there are any events, there must be a $<_{loc}$-minimal event satisfying the predicate $first$, and every non-minimal event e must have an immediate local predecessor $pred(e)$.
The causal ordering $\prec$ is also well-founded and is the transitive closure of $<_{loc}$ and the relation that a receive event e is preceded by $sender(e)$.

$$
\begin{array}{rcl}
\mathbb{D} & \equiv & \{T : \mathbb{U} \mid \forall x, y : T.\, Decidable(x = y \in T)\} \\
ES & \equiv & E : \mathbb{D} \,\times\, Loc : \mathbb{D} \,\times\, Lnk : \mathbb{D} \\
& & X : \mathbb{D} \,\times\, A : \mathbb{D} \,\times\, Tag : \mathbb{D} \\
& & \times\, T : Loc \to X \to \mathbb{U} \\
& & \times\, V : Loc \to Knd(Lnk, Tag, A) \to \mathbb{U} \\
& & \times\, M : Lnk \to Tag \to \mathbb{U} \\
& & \times\, src : Lnk \to Loc \\
& & \times\, dst : Lnk \to Loc \\
& & \times\, loc : E \to Loc \\
& & \times\, kind : E \to Knd(Lnk, Tag, A) \\
& & \times\, val : e : E \to V(loc(e), kind(e)) \\
& & \times\, \textbf{when} : x : X \to e : E \to T(loc(e), x) \\
& & \times\, \textbf{after} : x : X \to e : E \to T(loc(e), x) \\
& & \times\, \textbf{initially} : x : X \to i : Loc \to T(i, x) \\
& & \times\, sends : l : Lnk \to E \to List(Msg_l(Lnk, Tag, M)) \\
& & \times\, sender : \{e : E \mid isrcv(kind(e))\} \to E \\
& & \times\, index : \{e : E \mid isrcv(kind(e))\} \to \mathbb{N}_{\|sends(lnk(kind(e)), sender(e))\|} \\
& & \times\, first : E \to \mathbb{P} \\
& & \times\, pred : \{e : E \mid \neg first(e)\} \to E \\
& & \times\, <_{loc} : E \to E \to \mathbb{P} \\
& & \times\, \prec : E \to E \to \mathbb{P} \\
& & \times\, p : ESAxioms(E, Loc, Lnk, \ldots, pred, <_{loc}, \prec)
\end{array}
$$

$$
\begin{array}{l}
AntiReflexive(T, Rel) \;\equiv \\
\quad \forall x : T.\, \neg R(x, x) \\
Transitive(T, Rel) \;\equiv \\
\quad \forall x_1, x_2, x_3 : T.\, (Rel(x_1, x_2) \,\wedge\, Rel(x_2, x_3)) \;\Rightarrow\; Rel(x_1, x_3) \\
WellFounded(T, Rel) \;\equiv \\
\quad \forall P : T \to \mathbb{P}.\, (\forall x' : T.\, (\forall x : T.\, Rel(x, x') \;\Rightarrow\; P(x)) \;\Rightarrow\; P(x')) \;\Rightarrow \\
\qquad\qquad\qquad \forall x : T.\, P(x)
\end{array}
$$

$$
\begin{array}{rcl}
\langle e_1, n_1 \rangle \;<_{loc}\; \langle e_2, n_2 \rangle & \equiv & e_1 \;<_{loc}\; e_2 \;\vee\; (e_1 = e_2 \;\wedge\; n_1 < n_2) \\
emsg(e) & \equiv & msg(link(kind(e)), tag(kind(e)), val(e))
\end{array}
$$

$$
ESAxioms(E, Loc, Lnk, \ldots, pred, <_{loc}, \prec) \quad \equiv
$$

$$
Transitive(<_{loc}) \tag{1}
$$

$$WellFounded(<_{loc}) \tag{2}$$

$$\forall e, e' : E.\ loc(e) = loc(e') \Leftrightarrow (e <_{loc} e' \vee e = e' \vee e' <_{loc} e) \tag{3}$$

$$\forall e : E.\ Decidable(first(e)) \tag{4}$$

$$\forall e : E.\ first(e) \Leftrightarrow \forall e_1 : E.\ \neg(e_1 <_{loc} e) \tag{5}$$

$$\forall e : E.\ \neg first(e) \Rightarrow pred(e) <_{loc} e \wedge \forall e' : E.\ \neg(pred(e) <_{loc} e' <_{loc} e) \tag{6}$$

$$\forall e : E.\ first(e) \Rightarrow x \textbf{ when } e = x \textbf{ initially } loc(e) \tag{7}$$

$$\forall e : E.\ \neg first(e) \Rightarrow x \textbf{ when } e = x \textbf{ after } pred(e) \tag{8}$$

$$Transitive(\prec) \tag{9}$$

$$WellFounded(\prec) \tag{10}$$

$$\forall e : E.\ isrcv(kind(e)) \Rightarrow nth(index(e), sends(link(kind(e)), sender(e))) = emsg(e) \tag{11}$$

$$\forall e, e' : E.\ e <_{loc} e' \Rightarrow e \prec e' \tag{12}$$

$$\forall e : E.\ isrcv(kind(e)) \Rightarrow sender(e) \prec e \tag{13}$$

$$\forall e, e' : E.\ e \prec e' \Rightarrow (\neg first(e') \wedge e \preceq pred(e')) \vee (isrcv(kind(e')) \wedge e \preceq sender(e')) \tag{14}$$

$$\forall e : E.\ isrcv(kind(e)) \Rightarrow loc(e) = dst(lnk(kind(e))) \tag{15}$$

$$\forall e : E.\ \forall l : Lnk.\ loc(e) \neq src(l) \Rightarrow sends(l, e) = nil \tag{16}$$

$$\forall e_1, e_2 : E.\ \forall l : Lnk.\ isrcv_l(kind(e_1)) \wedge isrcv_l(kind(e_2)) \Rightarrow \langle sender(e_1), index(e_1)\rangle <_{loc} \langle sender(e_2), index(e_2)\rangle \Leftrightarrow e_1 <_{loc} e_2 \tag{17}$$

$$\forall e : E.\ \forall l : Lnk.\ \forall n : \mathbb{N}_{\|sends(l,e)\|}.\ \exists e' : E.\ isrcv_l(kind(e')) \wedge sender(e') = e \wedge index(e') = n \tag{18}$$

3.1 Consequences of the axioms

We state as lemmas some properties that follow from the axioms.

$$AntiReflexive(<_{loc}) \tag{19}$$

$$AntiReflexive(\prec) \tag{20}$$

$$\forall e, e' : E.\ e <_{loc} e' \Leftrightarrow \neg first(e') \wedge e \leq_{loc} pred(e') \tag{21}$$

$$\forall e, e' : E.\ e <_{loc} e' \wedge \forall e_1 : E.\ \neg(e <_{loc} e_1 <_{loc} e') \Rightarrow e = pred(e') \tag{22}$$

$$\forall e, e' : E.\ Decidable(e <_{loc} e') \tag{23}$$

$$\forall e, e' : E.\ Decidable(e \prec e') \tag{24}$$

$$\forall e : E.\ \forall l : Lnk.\ \forall tg : Lbl.\ \forall v : M(l, tg).\ msg(l, tg, v) \in sends(l, e) \Rightarrow \exists e' = rcv_l(tg)(v).\ e \prec e' \tag{25}$$

proofs: Lemmas 19 and 20 follow from the general fact that

$$WellFounded(Rel) \Rightarrow AntiReflexive(Rel)$$

Suppose $e <_{loc} e'$. From axiom (4) and axiom (5) we conclude $\neg first(e')$, and from axiom (6) we conclude

$$pred(e') <_{loc} e' \wedge \forall e'' : E. \neg(pred(e') <_{loc} e'' <_{loc} e')$$

So $\neg(pred(e') <_{loc} e)$ and hence, from axiom (3), $e \leq_{loc} pred(e')$, which proves lemma 21. If we also have $\forall e_1 : E. \neg(e <_{loc} e_1 <_{loc} e')$ then $\neg e <_{loc} pred(e')$, so $e = pred(e')$, which proves lemma 22.

We may now prove lemma 23 by induction, using axiom (2). By lemma 21 it's enough to decide $\neg first(e') \wedge e \leq_{loc} pred(e')$, but this is decidable by axiom (4), the induction hypothesis, and the decidability of equality in E. The proof of lemma 24 is similar. Using the other axioms we can show that axiom (14) can be proved as an if and only if statement, and hence it is enough to show that its righthand side is decidable. This follows from the induction hypothesis and the decidability of equality in E, and decidability of $first$ and $isrcv$.

If $msg(l, tg, v) \in sends(l, e)$ then for some $n < \|sends(l, e)\|$, $msg(l, tg, v) = nth(n, sends(l, e))$. By axiom (18) there is an e' such that

$$isrcv_l(kind(e')) \wedge sender(e') = e \wedge index(e') = n$$

So, by axiom (11),

$$val(e') = mval(msg(l, tg, v)) \wedge tg = mtag(msg(l, tg, v))$$

That implies that $e' = rcv_l(tg)(v)$ and since $e = sender(e')$ we have $e \prec e'$ by axiom (13). This proves lemma 25.

3.2 Local histories

An event system is a rich enough structure that we can define various "history" operators that list or count previous events having certain properties. Because we can define operators like these we do not need to add "history variables" to the states in order to write specifications and and prove them. The basic history operator lists all the prior events at a location.

Definition

$$\begin{array}{rcl}
\textbf{before}(e) & = & \textbf{if } first(e) \textbf{ then } [] \textbf{ else } pred(e) :: \textbf{before}(pred(e)) \\
\textbf{between}(e_1, e_2) & = & [e' \in \textbf{before}(e_2) \mid e_1 <_{loc} e'] \\
rcvs(l, \textbf{before}(e)) & = & [e' \in \textbf{before}(e) \mid isrcv(kind(e')) \wedge lnk(kind(e')) = l] \\
rcvs(l, tg, \textbf{before}(e)) & = & [e' \in rcvs(l, \textbf{before}(e)) \mid tag(kind(e')) = tg] \\
snds(l, \textbf{before}(e)) & = & concatenate([sends(l, e') \mid e' \in \textbf{before}(e)]) \\
snds(l, \textbf{before}(e, n)) & = & snds(l, \textbf{before}(e)) \textbf{ append } firstn(n-1, sends(l, e)) \\
snds(l, tg, \textbf{before}(e)) & = & [m \in snds(l, \textbf{before}(e)) \mid tag(m) = tg]
\end{array}$$

Using these operators we can state the following important lemma.

Lemma Fifo

$$\forall e' : E.\, \forall l : Lnk.\, isrcv_l(e') \;\Rightarrow$$
$$snds(l, \textbf{before}(sender(e'), index(e'))) = [emsg(e) \mid e \in rcvs(l, \textbf{before}(e'))]$$

proof: The proof is by induction on $<_{loc}$. Suppose $isrcv_l(e')$. If

$$snds(l, \textbf{before}(sender(e'), index(e'))) = \textbf{ nil}$$

then $rcvs(l, \textbf{before}(e'))$ must also be **nil** because, if $e \;<_{loc}\; e'$ is a receive on l then by axiom (17), $\langle sender(e), index(e)\rangle \;<_{loc}\; \langle sender(e'), index(e')\rangle$ which makes $snds(l, \textbf{before}(sender(e'), index(e')))$ non empty.
Otherwise, let

$$ms = last(snds(l, \textbf{before}(sender(e'), index(e'))))$$

then for some $\langle e, n\rangle \;<_{loc}\; \langle sender(e'), index(e')\rangle$,

$$snds(l, \textbf{before}(sender(e'), index(e'))) = snds(l, \textbf{before}(e, n))\ \textbf{append}\ [ms]$$

By axiom (18), $\exists e'' : E.\, isrcv_l(kind(e'')) \;\wedge\; sender(e'') = e \;\wedge\; index(e'') = n$ By axiom (17), $e'' \;<_{loc}\; e'$, so by induction,

$$snds(l, \textbf{before}(e, n)) = [emsg(e) \mid e \in rcvs(l, \textbf{before}(e''))]$$

If there were an e''' with $isrcv_l(e''')$ and $e'' \;<_{loc}\; e''' \;<_{loc}\; e'$ then by axiom (17)

$$\langle e, n\rangle \;<_{loc}\; \langle sender(e'''), index(e''')\rangle \;<_{loc}\; \langle sender(e'), index(e')\rangle$$

So, $nth(index(e'''), sends(l, sender(e'''))$ would come after ms in $snds(l, \textbf{before}(sender(e'), index(e')))$ contradicting the choice of ms as the last of the list. Thus $rcvs(l, \textbf{before}(e')) = rcvs(l, \textbf{before}(e''))$ **append** $[e'']$ and since, by axiom (11), $ms = emsg(e'')$, we have

$$snds(l, \textbf{before}(sender(e'), index(e'))) = [emsg(e) \mid e \in rcvs(l, \textbf{before}(e'))]$$

$\square$

Corollary

$$kind(e') = rcv_l(tg) \;\Rightarrow$$
$$\|snds(l, tg, \textbf{before}(sender(e'), index(e')))\| = \|rcvs(l, tg, \textbf{before}(e'))\|$$

$\square$

3.3 Event system shorthands

We make some shorthand notations:

$\forall e@i.\, \phi \;\equiv$
$\quad \forall e : E.\, loc(e) = i \;\Rightarrow\; \phi$
$\forall e@i = pred(e').\, \phi \;\equiv$

$$\forall e, e' : E.\, loc(e) = i \;\wedge\; e = pred(e') \;\Rightarrow\; \phi$$

$$\forall e@i = k(v).\, \phi \;\equiv$$

$$\forall e : E.\, \forall v : V(i,k).\, loc(e) = i \;\wedge\; kind(e) = k \;\wedge\; val(e) = v \;\Rightarrow\; \phi$$

$$\forall e = k(v).\, \phi \;\equiv$$

$$\forall e : E.\, \forall i : Loc.\, \forall v : V(i,k).\, loc(e) = i \;\wedge\; kind(e) = k \;\wedge\; val(e) = v \;\Rightarrow\; \phi$$

$$\exists e@i.\, \phi \;\equiv$$

$$\exists e : E.\, loc(e) = i \;\wedge\; \phi$$

$$\exists e@i = k(v).\, \phi \;\equiv$$

$$\exists e : E.\, \exists v : V(i,k).\, loc(e) = i \;\wedge\; kind(e) = k \;\wedge\; val(e) = v \;\wedge\; \phi$$

$$\exists e = k(v).\, \phi \;\equiv$$

$$\exists e : E.\, \exists i : Loc.\, \exists v : V(i,k).\, loc(e) = i \;\wedge\; kind(e) = k \;\wedge\; val(e) = v \;\wedge\; \phi$$

$$\exists e' >_{loc} e.\, \phi \;\equiv$$

$$\exists e' : E.\, e \;<_{loc}\; e' \wedge\; \phi$$

$$\exists e' <_{loc} e.\, \phi \;\equiv$$

$$\exists e' : E.\, e' \;<_{loc}\; e \wedge\; \phi$$

$$\exists e' \geq_{loc} e.\, \phi \;\equiv$$

$$\exists e' : E.\, e \;\leq_{loc}\; e' \wedge\; \phi$$

$$\exists e' \leq_{loc} e.\, \phi \;\equiv$$

$$\exists e' : E.\, e' \;\leq_{loc}\; e \wedge\; \phi$$

3.4 Change operator

Definition

$$\begin{array}{rcl} x \,\Delta\, e & = & (x \textbf{ after } e \neq x \textbf{ when } e) \\ \Delta(x,e) & = & \|[e_1 \in \textbf{before}(e) \mid x \,\Delta\, e_1]\| \\ & & \text{(only defined when } T(loc(e),x) \text{ has decidable equality)} \\ x \,\Delta_n\, e & = & 0 < n \;\wedge\; \Delta(x,e) = n-1 \;\wedge\; x \,\Delta\, e \end{array}$$

The formula $x \,\Delta\, e$ is true when event e makes a change in state variable x. The formula $\Delta(x,e) = n$ is true when there have been exactly n changes to x strictly before event e. The formula $x \,\Delta_n\, e$ is true when there have been exactly n changes to x upto and including event e and one of the changes is at e.

Properties of Δ

Suppose that, $x \in X$, $i \in Loc$, and $T(i,x) \in \mathbb{D}$, i.e. the type of x at location i has decidable equality. Then,

$$\forall e@i.\, \forall n : \mathbb{N}. \tag{26}$$

$$x \,\Delta\, e,\; \Delta(x,e) = n, \text{ and } x \,\Delta_n\, e \text{ are decidable}$$

$$\forall e@i.\, e \;<_{loc}\; e' \;\Rightarrow\; \Delta(x,e) \leq \Delta(x,e') \tag{27}$$

$$\forall e'@i.\ \Delta(x,e') = n \ \wedge \ e = pred(e') \ \Rightarrow \quad \Delta(x,e) = n \ \vee \ x \ \Delta_n \ e \tag{28}$$

$$\forall e@i.\ \Delta(x,e) = 0 \ \Rightarrow \ x \ \textbf{when} \ e = x \ \textbf{initially} \ i \tag{29}$$

$$\forall e'@i.\ x \ \textbf{when} \ e' \neq x \ \textbf{initially} \ i \ \Rightarrow \ \exists e <_{loc} e'.\ x \ \Delta \ e$$

proof: lemma (26). If $loc(e) = i$, then x **when** e and x **after** e have type $T(i,x)$ so if equality in $T(i,x)$ is decidable, then $x \ \Delta \ e$ is decidable. We can then prove that the other predicates, $\Delta(x,e) = n$ and $x \ \Delta_n \ e$ are decidable, by induction on $<_{loc}$. Essentially, they are defined by bounded quantification over the predecessors of e from the decidable $x \ \Delta \ e$.
lemma (27) follows by induction on $<_{loc}$.
lemma (28). Under the hypotheses,

$$n = \Delta(x,e') = \Delta(x,e) + \textbf{if} \ x \ \Delta \ e \ \textbf{then} \ 1 \ \textbf{else} \ 0$$

If $x \ \Delta \ e$ then $x \ \Delta_n \ e$ and otherwise $\Delta(x,e) = n$.
lemma (29) is proved by induction on $<_{loc}$. If e has no predecessors then x **when** $e' = x$ **initially** i so the assertion is true. If $e_1 = pred(e)$ and $\Delta(x,e) = 0$ then, by lemma (28), $\Delta(x,e_1) = 0$, so, by induction, x **when** $e_1 = x$ **initially** i. Also, $\neg(x \ \Delta \ e_1)$, so x **when** $e_1 = x$ **after** $e_1 = x$ **when** e, and hence x **when** $e = x$ **initially** i.
lemma (30). Under the decidability assumption, $\exists e <_{loc} e'.\ x \ \Delta \ e$ is decidable. If it is true then the assertion is true. If it is false, then $\Delta(x,e') = 0$, so by lemma (29), x **when** $e = x$ **initially** i, which contradicts the hypothesis.

□

4 Worlds

4.1 Definition of World

A *world* is a generalized trace of the execution of a distributed system. It has locations and links from a graph $\langle Loc, Lnk, src, dst : Lnk \rightarrow Loc \rangle$. Time is modeled as the natural numbers $\mathbb{N}$. By observing the system at every location i and every time t, we have a state $s(i,t)$, an action $a(i,t)$, and a list of messages $m(i,t)$. The state $s(i,t)$ is the state of the part of the system at location i at time t. We assume that the type of the state at location i does not change with time, and we use a general model of state as a record. A record is a dependent function. If X is a type and if $dec : X \rightarrow \mathbb{U}$ is a type assignment, then the record type $Record(X, dec)$ is

$$\begin{aligned} Record(X, dec) \quad &\equiv \quad x : X \rightarrow dec(x) \\ r.x \quad &\equiv \quad r(x) \text{ , for } r \in Record(X, dec) \text{ and } x \in X \end{aligned}$$

A world contains a type X of state variable names and and a type assignment $T : Loc \rightarrow X \rightarrow \mathbb{U}$. The state at location i of the world will have type $Record(X, T(i))$.
The action $a(i,t)$ is the action that was chosen by the system to be executed next at location i and time t. It will always be possible that no action was taken at i, t so we must have a null action. Other action will be local actions with names taken from a type of action names A, and also the action of receiving a message. Every action will have a kind of one of these three forms (null, local, or receive), and also a value whose type depends on the kind and location of the action.

$$Action(Lnk, Tag, A, dec) \ \equiv$$

$$
\begin{aligned}
&Unit + \ k : Knd(Lnk, Tag, A) \ \times \ dec(k) \\
&isnull(inl(x)) = true \\
&isnull(inr(x)) = false \\
&kind(inr(\langle k, v\rangle)) = k \\
&val(inr(\langle k, v\rangle)) = v \\
&isrcv_l(a) = \neg isnull(a) \ \wedge \ isrcv(kind(a)) \ \wedge \ lnk(kind(a)) = l \\
&isrcv_{l,tg}(a) = isrcv_l(a) \ \wedge \ tag(kind(a)) = tg
\end{aligned}
$$

The messages $m(i, t)$ are the list of messages sent from location i at time t. For messages, we use the message type $Msg(Lnk, Tag, dec)$ defined earlier.

$$
\begin{aligned}
World \quad \equiv \quad & Loc : \mathbb{D} \ \times \ Lnk : \mathbb{D} \ \times \ src, dst : Lnk \to Loc \\
& \times X : \mathbb{D} \ \times \ A : \mathbb{D} \ \times \ Tag : \mathbb{D} \\
& \times T : Loc \to X \to \mathbb{U} \\
& \times TA : Loc \to A \to \mathbb{U} \\
& \times M : Lnk \to Tag \to \mathbb{U} \\
& \times s : i : Loc \to \mathbb{N} \to Record(X, T(i)) \\
& \times a : i : Loc \to \mathbb{N} \to Action(A, Lnk, Tag, kindcase(TA(i), M)) \\
& \times m : i : Loc \to \mathbb{N} \to List(Msg(Lnk, Tag, M))
\end{aligned}
$$

If $w : World$ is a world, then we write w_{Loc}, w_{Lnk}, $\ldots$, w_s, w_a, and w_m for the components of w.

4.2 Fair-Fifo Worlds

We next define a *fair-fifo* world. We first note that, given world w, we can find all the messages sent on link l and all and receive actions that have occurred on link l before time t:

$$
\begin{aligned}
m(w, l, t) \quad &= \quad onlink(l, w_m(w_{src}(l), t)) \\
snds(w, t, l) \quad &= \quad concatenate[m(w, l, t_1) \mid t_1 \in [0, t)] \\
rcvs(w, t, l) \quad &= \quad [a \in [w_a(w_{dst}(l), t_1) \mid t_1 \in [0, t)] \mid isrcv_l(a)]
\end{aligned}
$$

The send and receive messages before time t define an implicit queue, and we can test whether the queue for link l is empty and for whether message ms is at the head of the queue for its link:

$$
\begin{aligned}
&isempty(w, t, l) \ \equiv \\
&\qquad \|snds(w, t, l)\| \leq \|rcvs(w, t, l)\| \\
&ishd(w, t, ms)) = \\
&\qquad let\ s = snds(w, t, mlnk(ms))\ in \\
&\qquad let\ r = rcvs(w, t, mlnk(ms))\ in \\
&\qquad \|s\| > \|r\| \ \wedge \ s[\|r\|] = ms
\end{aligned}
$$

$$
\begin{array}{ll}
 & FairFifo(w) \equiv \\
(1) & \quad (\forall i : w_{Loc}.\, \forall t : \mathbb{N}.\, \forall l : w_{Lnk}.\, w_{src}(l) \neq i \;\Rightarrow \\
 & \qquad\qquad onlink(l, w_m(i,t)) = \textbf{ nil }) \\
(2) & \quad \wedge\, (\forall i : w_{Loc}.\, \forall t : \mathbb{N}.\, isnull(w_a(i,t)) \;\Rightarrow \\
 & \qquad\qquad w_s(i,t+1) = w_s(i,t) \;\wedge\; w_m(i,t) = \textbf{ nil }) \\
(3) & \quad \wedge\, (\forall i : w_{Loc}.\, \forall t : \mathbb{N}.\, \forall l : w_{Lnk}.\, \forall tg : w_{Tag}.\, isrcv_{l,tg}(w_a(i,t)) \;\Rightarrow \\
 & \qquad\qquad w_{dst}(l) = i \;\wedge\; ishd(w, t, msg(l, tg, val(w_a(i,t))))) \\
(4) & \quad \wedge\, (\forall l : w_{Lnk}.\, \exists^{\infty} t : \mathbb{N}.\, (isrcv_l(w_a(w_{dst}(l), t)) \;\vee\; isempty(w, t, l)))
\end{array}
$$

The first clause says that location i can only send message on links whose source is i. The second clause says that a null action leaves the state unchanged and sends no messages. The third clause says that a receive action at location i must be on a link whose destination is i and whose message is at the head of the queue. The fouth clause is the fairness clause. It says that for every queue, infinitely often either the queue is empty or a receive event occurs at its destination.

4.3 Event System of a World

If w is a fair-fifo world, then we can construct an event system from w. The types Loc, Lnk, X, A, Tag are already in w, so we have to define the type E of events and define all the operations on events and show that the axioms are satisfied. Our events will be the points $\langle i, t\rangle$ in spacetime at which an action occured in w.

$$
\begin{array}{rcl}
w_E &=& \{\langle i,t\rangle : w_{Loc} \times \mathbb{N} \mid \neg isnull(w_a(i,t))\} \\
w_{loc}(\langle i,t\rangle) &=& i \\
w_{time}(\langle i,t\rangle) &=& t \\
w_{action}(\langle i,t\rangle) &=& w_a(i,t) \\
w_{state}(\langle i,t\rangle) &=& w_s(i,t) \\
w_{state'}(\langle i,t\rangle) &=& w_s(i,t+1) \\
w_{init}(i) &=& w_s(i,0) \\
w_{msgs}(\langle i,t\rangle) &=& w_m(i,t)
\end{array}
$$

For and event $e \in w_E$ we have $\neg isnull(w_{action}(e))$ so we may define

$$
\begin{array}{rcl}
w_{kind}(e) &=& kind(w_{action}(e)) \\
w_{val}(e) &=& val(w_{action}(e))
\end{array}
$$

The type of the value of an event can be determined from its location and kind using the type assignments w_{TA} and w_M as follows:

$$
w_V(i,k) \quad = \quad kindcase(w_{TA}(i), w_M, k)
$$

The observation operators are defined in the obvious way:

$$
w_{when}(x,e) \quad = \quad w_{state}(e).x
$$

$$\begin{array}{rcl} w_{after}(x, e) & = & w_{state'}(e).x \\ w_{initially}(x, i) & = & w_{init}(i).x \\ w_{sends}(l, e) & = & onlink(l, w_{msgs}(e)) \end{array}$$

The local ordering operations are also straightforward.

$$\begin{array}{rcl} w_{first}(\langle i, t\rangle) & = & \forall t' : \mathbb{N}.\ t' < t \ \Rightarrow\ isnull(w_a(i, t')) \\ w_{pred}(\langle i, t\rangle) & = & \langle i, greatest\ t' < t. \neg isnull(w_a(i, t'))\rangle \\ w_{<_{loc}}(\langle i, t\rangle, \langle j, t'\rangle) & = & i = j \ \wedge\ t < t' \end{array}$$

To define the $sender$ and $index$ operations that match a receive event to its origin, we first define a $match$ with the same $snds$ and $rcvs$ functions used in defining $FairFifo$.

$$\begin{array}{rcl} match(l, t, n, t') & = & n < \|m(w, l, t)\| \ \wedge \\ & & \|rcvs(w, t', l)\| = \|snds(w, t, l)\| + n \end{array}$$

Then, we define $sender$ and $index$ as follows

$$\begin{array}{rcl} w_{sender}(\langle j, t'\rangle) & = & \text{let } l = lnk(w_{kind}(\langle j, t'\rangle)) \text{ in} \\ & & \langle src(l), \mu t < t'.\ \exists n : \mathbb{N}.\ match(l, t, n, t')\rangle \\ w_{index}(\langle j, t'\rangle) & = & \text{let } l = lnk(kind(\langle j, t'\rangle)) \text{ in} \\ & & \text{let } \langle i, t\rangle = sender(\langle j, t'\rangle) \text{ in} \\ & & \mu n.\ match(l, t, n, t') \end{array}$$

Finally, the causal ordering $\prec$ is defined as a transitive closure

$$\begin{array}{rcl} w_{\prec} & = & \text{transitive closure}(w_{<_{loc}} \cup \ \mapsto), \\ & & \ \text{where} \\ e \ \mapsto\ e' & = & isrcv(w_{kind}(e')) \ \wedge\ e = w_{sender}(e') \end{array}$$

Putting all of these defined operations together, we have the event structure defined by the world

$$\begin{array}{rcl} Ev(w) & = & \langle w_E, w_{Loc}, w_{Lnk}, w_X, w_A, w_{Tag}, w_T, w_V, w_M, \\ & & w_{src}, w_{dst}, w_{loc}, w_{kind}, w_{val}, w_{when}, w_{after}, w_{initially}, \\ & & w_{sends}, w_{sender}, w_{index}, w_{first}, w_{pred}, w_{<_{loc}}, w_{\prec}\rangle \end{array}$$

Theorem (World-Event-System)

$$\forall w : World.\ FairFifo(w) \ \Rightarrow\ ESAxioms(Ev(w))$$

proof:

axiom 1, 2, and 3 These follow from the definitions of w_{loc} and $w_{<_{loc}}$, which make $w_{<_{loc}}$ on events at a fixed location isomorphic to $<$ on a subset of the natural numbers.

axiom 4 w_{first} is defined by a bounded quantification and $isnull$ is decidable, so w_{first} is decidable.

axiom 5 Follows from the definitions of w_{first}, $w_{<_{loc}}$, and w_E.

axiom 6 Follows from the definitions of w_{pred}, $w_{<_{loc}}$, and w_E.

axiom 7 If $w_{first}(e)$ where $e = \langle i, t \rangle$ then all actions $w_a(i, t')$ for $t' < t$ are null. So by clause (2) of $FairFifo$ by induction we have $w_s(i, t) = w_s(i, 0)$, and the axiom follows from the definitons of w_{when} and $w_{initially}$.

axiom 8 Similarly if $\langle i, t \rangle = w_{pred}(\langle i, t' \rangle)$ then all actions $w_a(i, t'')$ for $t < t'' < t'$ are null. Hence by clause (2) of $FairFifo$, $w_s(i, t') = w_s(i, t + 1)$, and the axiom follows from the definitions of w_{after} and s_{when}.

axiom 9 By definition, $w_{\prec}$ is a transitive closure.

axiom 10 Since $w_{\prec}$ is the transitive closure of two relations, $w_{<_{loc}}$ and $\mapsto$, it's enough to show that each of these relations agrees with the order of w_{time}. The first relation, $w_{<_{loc}}$ does, by definition. For the second, suppose $\langle i, t \rangle = sender(\langle j, t' \rangle)$, then by definition, $t = \mu t < t'. \exists n : \mathbb{N}.\ match(l, t, n, t')$ so $t < t'$. But we haven't yet shown that it exists, i.e. that w_{sender} is well-defined.

To show that, suppose that $isrcv(w_{kind}(\langle j, t' \rangle))$. Then, for some l and tg we have $isrcv_{l,tg}(w_a(j, t'))$, so by clause (3) of $FairFifo$, we have $w_{dst}(l) = j \wedge ishd(w, t', msg(l, tg, val(w_a(j, t'))))$. This means that for $s = snds(w, t', l)$ and $r = rcvs(w, t', l)$, we have $\|s\| > \|r\| \wedge s[\|r\|] = msg(l, tg, val(w_a(j, t')))$. But $snds(w, t', l)$ is the concatenation of $m(w, l, t)$ for $t < t'$, so for some such t, $\|snds(w, t, l)\| < \|r\| < \|snds(w, t, l)\| + \|m(w, l, t)\|$, and this implies that there is an $n < \|m(w, l, t)\|$ such that $\|rcvs(w, t', l)\| = \|snds(w, t, l)\| + n$, so $match(l, t, n, t')$. This argument shows that w_{sender} and w_{index} are both well-defined.

axiom 11 This axiom follows from the previous argument, since under the assumption that $isrcv_{l,tg}(w_a(j, t'))$ we found that $w_{sender}(\langle j, t' \rangle) = \langle w_{src}(l), t \rangle$ and $w_{index}(\langle j, t' \rangle) = n$ for t and n satisfying $nth(n, m(w, l, t)) = msg(l, tg, val(w_a(j, t')))$. But $m(w, l, t) = onlink(l, w_m(w_{src}(l), t)) = w_{sends}(l, \langle w_{src}(l), t \rangle)$, so we have

$$nth(w_{index}(\langle j, t' \rangle), w_{sends}(l, w_{sender}(\langle j, t' \rangle))) = msg(l, tg, val(w_a(j, t')))$$

and hence the tag and val components are equal as asserted in the axiom.

axiom 12 By definition, $w_{\prec}$ contains $w_{<_{loc}}$.

axiom 13 By definition, $w_{\prec}$ contains $\mapsto$, and this implies the axiom.

axiom 14 If $e \prec e'$ then since $w_{\prec}$ is defined to be the transitive closure of two relations, there must be a chain of these relations connecting e and e'.

If the last link of the chain is $e'' <_{loc} e'$ then we have $\neg first(e')$ and $loclee''pred(e')$ and $e \preceq e''$, and so by transitivity we have the first possibility.

If the last link in the chain is $e'' \mapsto e'$, then e' is a receive and $e'' = sender(e')$ and $e \preceq sender(e')$, so we have the second possibilty.

axiom 15 This follow from clause (3) of $FairFifo$.

axiom 16 Once the definitions are unfolded, this axiom is exactly clause (1) of $FairFifo$.

axiom 17 If $e_1 = \langle j_1, t_1' \rangle$ and $e_2 = \langle j_2, t_2' \rangle$ satisfy the hypotheses of the axiom then, as in the proofs of axioms 10 and 11, we must have t_1, n_1, t_2, and n_2 such that

$$\begin{array}{rcl} w_{sender}(e_1) & = & \langle w_{src}(l), t_1 \rangle \\ w_{index}(e_1) & = & n_1 < \|m(w,l,t_1)\| \\ \|rcvs(w,t_1',l)\| & = & \|snds(w,t_1,l)\| + n_1 \\ w_{sender}(e_2) & = & \langle w_{src}(l), t_2 \rangle \\ w_{index}(e_2) & = & n_2 < \|m(w,l,t_2)\| \\ \|rcvs(w,t_2',l)\| & = & \|snds(w,t_2,l)\| + n_2 \end{array}$$

If $sender(e_1) \ <_{loc}\ sender(e_2)$ then $t_1 < t_2$ and this implies, by definition of $snds(w,t_2,l)$, that $\|snds(w,t_1,l)\| + n_1 < \|snds(w,t_2,l)\|$, and hence that $\|rcvs(w,t_1',l)\| < \|rcvs(w,t_2',l)\|$. This implies that $t_1' < t_2'$ and hence $e_1 \ <_{loc}\ e_2$. If $sender(e_1) = sender(e_2) \ \wedge \ n_1 < n_2$ then we reach the same conclusion. So, $\langle sender(e_1), index(e_1) \rangle \ <_{loc}\ \langle sender(e_2), index(e_2) \rangle \ \Rightarrow$ $e_1 \ <_{loc}\ e_2$.

To show the reverse implication, suppose $e_1 \ <_{loc}\ e_2$. By axiom (3) we have either $\langle sender(e_1), index(e_1) \rangle \ <_{loc}\ \langle sender(e_2), index(e_2) \rangle$ or $\langle sender(e_2), index(e_2) \rangle \ <_{loc}$ $\langle sender(e_1), index(e_1) \rangle$ or
$\langle sender(e_1), index(e_1) \rangle = \langle sender(e_2), index(e_2) \rangle$. The first case is what we want to prove, and the second case implies $e_2 \ <_{loc}\ e_1$ (by the previous argument) which contradicts our hypothesis. In the third case, $t_1 = t_2$ and $n_1 = n_2$, and this implies that $\|rcvs(w,t_1',l)\| = \|rcvs(w,t_2',l)\|$. But, this is impossible since our assumption implies that $t_1' < t_2'$ and hence $\|rcvs(w,t_1',l)\| < \|rcvs(w,t_2',l)\|$.

axiom 18 This axiom says that every message that is sent will be received. We prove by induction on m that

$$\begin{array}{l} \forall m : \mathbb{N}.\ \forall l : w_{Lnk}.\ \forall t : \mathbb{N}. \\ m \le \|snds(w,t,l)\| \ \Rightarrow\ \exists t' \ge t.\ m \le \|rcvs(w,t',l)\| \end{array}$$

When $m = 0$ we can take $t' = 0$ and the assertion holds. Assume it holds for m and prove it for $m+1$. So let l and t be such that $m + 1 \le \|snds(w,t,l)\|$. By induction, we can find $t'' \ge t$ such that $m \le \|rcvs(w,t'',l)\|$. By the fairness clause (4) of $FairFifo$, we may choose $t' \ge t''$ such that

$$isrcv_l(w_a(w_{dst}(l),t)) \ \vee\ isempty(w,t,l)$$

In the first case, we have

$$m \le \|rcvs(w,t'',l)\| < 1 + \|rcvs(w,t',l)\| = \|rcvs(w,t'+1,l)\|$$

so $m + 1 \le \|rcvs(w,t'+1,l)\|$. In the second case, by definition of $isempty$,

$$\|rcvs(w,t',l)\| \ge \|snds(w,t',l)\| \ge \|snds(w,t,l)\| \ge m+1$$

So, in either case, $\exists t' \ge t.\ \|rcvs(w,t',l)\| \ge (m+1)$ and that completes the proof of the claim.

Now, to prove axiom 18, we let $e = \langle i, t\rangle$ be an event, and l be a link and suppose $n < \|sends(l, e)\|$. Then $\|snds(w, t+1, l)\| > \|snds(w, t, l)\| + n$. By the claim, we can find t'' such that $\|rcvs(w, t'', l)\| > \|snds(w, t, l)\| + n$. This implies that, for $j = w_{dst}(l)$, there is a t' such that

$$isrcv_l(kind(w_a(j, t'))) \ \wedge \ \|rcvs(w, t', l)\| = \|snds(w, t, l)\| + n$$

So, we have $match(l, t, n, t')$. If we let $e' = \langle j, t'\rangle$, then e' is an event, $lnk(kind(e')) = l$, and $sender(e') = e$ and $index(e') = n$.

□

5 Message-Automata

Event systems and worlds are infinite objects, but they arise from the behaviors of distributed systems where, at each location, only a finite program constrains the behavior. We call our representations of these finite programs message-automata. To make our representations finite we need to replace infinite things like total type assignments with finite approximations, so we need some notation for finite partial functions.

5.1 Finite partial functions

A finite partial function f from A to B has the type $f : A \rightarrow_{fpf} B$. Its domain is $dom(f)$, and we define

$$\begin{array}{rcl} f(x)?z & \equiv & \textbf{if } x \in dom(f) \textbf{ then } f(x) \textbf{ else } z \\ Z \ =! \ f(x) \ \Rightarrow \ t(Z) & \equiv & (x \in dom(f)) \Rightarrow t(f(x)) \end{array}$$

For finite partial functions $f, g : A \rightarrow_{fpf} B$ we define:

$$\begin{array}{rcl} f \ \subseteq \ g & \equiv & \forall x : A.\, x \in dom(f) \ \Rightarrow \ x \in dom(g) \ \wedge \ f(x) = g(x) \\ f \ \| \ g & \equiv & \forall x : A.\, x \in dom(f) \ \cap \ dom(g) \ \Rightarrow \ f(x) = g(x) \\ f \ \oplus \ g & \equiv & \lambda x.\, \textbf{if } x \in dom(g) \textbf{ then } g(x) \textbf{ else } f(x) \end{array}$$

lemma

$$\forall f, g : A \rightarrow_{fpf} B.\, f \ \| \ g \ \Rightarrow \ f \ \subseteq \ f \ \oplus \ g \wedge g \ \subseteq \ f \ \oplus \ g$$

lemma

$$\begin{array}{l} \forall f, g : A \rightarrow_{fpf} B.\, f \ \subseteq \ g \ \Rightarrow \ \forall x : A.\, \forall p : B \rightarrow \mathbb{P}. \\ (Z \ =! \ g(x) \ \Rightarrow \ p(Z)) \ \Rightarrow \ (Z \ =! \ f(x) \ \Rightarrow \ p(Z)) \end{array}$$

5.2 Definition of Message-Automata

The message-automata share with the worlds and the event systems the same spaces of names for state variables, local action kinds, and message tags. So we will have parameters X, A, and Tag as before, but, where a world has, at each location i, type assignments $T(i) : X \rightarrow \mathbb{U}$, $TA(i) : A \rightarrow \mathbb{U}$, and $M : Lnk \to Tag \to \mathbb{U}$, a message-automaton will know only its input and ouput links In, and Out, and its type assignments (declarations) will be finite

$$\begin{aligned}
&ds : X \to_{fpf} \mathbb{U}\\
&da : A \to_{fpf} \mathbb{U}\\
&din : In \times Tag \to_{fpf} \mathbb{U}\\
&dout : Out \times Tag \to_{fpf} \mathbb{U}
\end{aligned}$$

The domain of ds is the set of declared state variables, the domain of da is the set of declared local actions, the domain of din is the set of declared input message types, and the domain of $dout$ is the set of declared output message types.

The state of a message-automaton will be the record defined by its declarations ds. We can define this type using the dependent function type $Record(X, dec)$ used in the worlds by extending the finite partial function ds to a total function. We do this by assigning the type Top to any undeclared state variable.

$$State(X, ds) \equiv Record(X, \lambda x.\ ds(x)?Top)$$

The type of output messages that the automaton has declared is defined in a similar way

$$Message(Lnk, Tag, dout) \equiv Msg(Lnk, Tag, \lambda p.\ dout(p)?Top)$$

The kinds of actions that the automaton has declared and that can have effects on the state are a subset of the kinds $Knd(Lnk, Tag, A)$

$$\begin{aligned}
&Kind(Lnk, Tag, A, da, din) \equiv\\
&\quad \{k : Knd(Lnk, Tag, A) \mid kindcase(\lambda a.\ a \in dom(da), \lambda p.\ p \in dom(din), k)\}
\end{aligned}$$

$$ktype(da, din, k) \quad \equiv kindcase(da, din, k)$$

In addition, to its declarations, the message-automaton does the following things

$init$ It constrains the initial values of the state variables. So, it has a finite partial function $init$ of type $x : dom(ds) \to_{fpf} (ds(x) \to \mathbb{P})$. Thus, if x is in the domain of $init$ then x is a declared state variable and $init(x)$ is a predicate on the declared type $ds(x)$ of state variable x.

pre It declares preconditions on its local actions. So, it has a finite partial function pre of type

$$a : dom(da) \to_{fpf} (State(X, ds) \to da(a) \to \mathbb{P})$$

Thus, if a is in the domain of pre then a is a declared local action and $pre(a)$ is a predicate on the state and the declared type $da(a)$ of the action.

ef It declares the effects of actions (local and input) on state variables. So, it has a finite partial function ef of type

$$\begin{aligned}
&\langle k, x\rangle : Kind(Lnk, Tag, A, da, din) \times dom(ds) \to_{fpf}\\
&(State(X, ds) \to ktype(da, din, k) \to ds(x))
\end{aligned}$$

Thus, if $\langle k, x\rangle$ is in the domain of ef then k is a declared kind (either a local action or a receive of an input message) and x is a declared state variable, and $ef(\langle k, x\rangle)$ is a function from the state and the type of the action to the type $ds(x)$ of x. This function defines how the new value of x will be computed from the current state and the value of the action.

$send$ It declares the messages sent by actions. So, it has a finite partial function $send$ of type

$$\langle k, x\rangle : Kind(Lnk, Tag, A, da, din) \times dom(ds) \rightarrow_{fpf}$$
$$(State(X, ds) \rightarrow ktype(da, din, k) \rightarrow List(Message(Lnk, Tag, dout)))$$

Thus, if $\langle k, x\rangle$ is in the domain of $send$ then k is a declared kind (either a local action or a receive of an input message) and x is a declared state variable, and $snd(\langle k, x\rangle)$ is a function from the state and the type of the action to the type of lists of output messages.

$frame$ It declares implicit effects. By convention, the effects that are explicitly given are the only actions that affect the given state variables. So the implicit effect of any other action is to leave the state of variable unchanged. Since we want each clause of a message-automaton to be meaningful on its own, we can't depend on such contextual conventions, so we have to make the implicit effects explicit in so-called *frame* clauses. The message-automaton has a finite partial function $frame$ of type $dom(ds) \rightarrow_{fpf} List(Kind(Lnk, Tag, A, da, din))$. So if x is in the domain of $frame$ then x is a declared state variable and $frame(x)$ is a list of actions kinds that contains all the kinds that affect x.

$sframe$ It declares implicit sends. By convention, the sends that are explicitly given are the only actions that send messages on the given link with the given tag. So the implicit sends of any other action is to send no messages of the given link and tag. We make the implicit sends explicit in *sframe* clauses. The message-automaton has a finite partial function $sframe$ of type $Out \times Tag \rightarrow_{fpf} List(Kind(Lnk, Tag, A, da, din))$. So if $\langle l, tg\rangle$ is in the domain of $sframe$ then l is an output link and $sframe(\langle l, tg\rangle)$ is a list of actions kinds that contains all the kinds that send messages with tag tg on link l.

Putting all of these pieces into a structure we define the type of message-automata:

$$
\begin{aligned}
&MsgA \equiv \\
&\quad \times X : \mathbb{D} \times A : \mathbb{D} \times Tag : \mathbb{D} \times Lnk : \mathbb{D} \\
&\quad \times In : \{T : \mathbb{D} \mid T \subseteq Lnk\} \times Out : \{T : \mathbb{D} \mid T \subseteq Lnk\} \\
&\quad \times ds : X \rightarrow_{fpf} \mathbb{U} \\
&\quad \times da : A \rightarrow_{fpf} \mathbb{U} \\
&\quad \times din : In \times Tag \rightarrow_{fpf} \mathbb{U} \\
&\quad \times dout : Out \times Tag \rightarrow_{fpf} \mathbb{U} \\
&\quad \times init : x : dom(ds) \rightarrow_{fpf} (ds(x) \rightarrow \mathbb{P}) \\
&\quad \times pre : a : dom(da) \rightarrow_{fpf} (State(X, ds) \rightarrow da(a) \rightarrow \mathbb{P}) \\
&\quad \times ef : \langle k, x\rangle : Kind(Lnk, Tag, A, da, din) \times dom(ds) \rightarrow_{fpf} \\
&\quad\quad (State(X, ds) \rightarrow ktype(da, din, k)) \rightarrow ds(x)) \\
&\quad \times send : \langle k, x\rangle : Kind(Lnk, Tag, A, da, din) \times dom(ds) \rightarrow_{fpf} \\
&\quad\quad (State(X, ds) \rightarrow ktype(da, din, k)) \rightarrow List(Message(Lnk, Tag, dout))) \\
&\quad \times frame : dom(ds) \rightarrow_{fpf} List(Kind(Lnk, Tag, A, da, din)) \\
&\quad \times sframe : Out \times Tag \rightarrow_{fpf} List(Kind(Lnk, Tag, A, da, din))
\end{aligned}
$$

Message-Automata A and B have the same signature if their X, A, Tag, Lnk, In, and Out components are equal. The subtype of $MsgA$ with given signature $\langle X, A, Tag, Lnk, In, Out\rangle$ is

$$MsgA(X, A, Tag, Lnk, In, Out) \equiv$$

$$\{a : MsgA \mid a_X = X \ \wedge \ a_A = A \ \wedge \ a_{Tag} = Tag \ \wedge \\ a_{Lnk} = Lnk \ \wedge \ a_{In} = In \ \wedge \ a_{Out} = Out\}$$

Message-Automata A and B are compatible ($A \ \| \ B$) or satisfy the relation $A \ \subseteq \ B$ if they have the same signature and the ten finite partial functions, ds, da, din, $dout$, $init$, pre, ef, snd, $frame$, and $sframe$ of A and B are compatible or are related by $\subseteq$. And we define $A \ \oplus \ B$ by applying the $\oplus$ operation to each of the ten components.

lemma

$$\forall A, B : MsgA.\ A \ \| \ B \ \Rightarrow \ A \ \subseteq \ A \ \oplus \ B \ \wedge \ B \ \subseteq \ A \ \oplus \ B$$

5.3 Distributed Systems

A network is represented by a graph $\langle Loc, Lnk, src, dst\rangle$. The incoming and outgoing edges at a vertex i are defined by

$$\begin{aligned} In(dst, i) &= \{l : Lnk \mid dst(l) = i\} \\ Out(src, i) &= \{l : Lnk \mid src(l) = i\} \end{aligned}$$

A distributed system is a network graph, name spaces X, A, and Tag, and an assigmnent of a message-automaton to each location.

$$\begin{aligned} &Dsys \ \equiv \\ &\quad Loc : \mathbb{D} \ \times \ Lnk : \mathbb{D} \ \times \ src : Lnk \rightarrow Loc \ \times \ dst : Lnk \rightarrow Loc \\ &\quad \times \ X : \mathbb{D} \ \times \ A : \mathbb{D} \ \times \ Tag : \mathbb{D} \\ &\quad \times \ m : i : Loc \rightarrow MsgA(X, A, Tag, Lnk, In(dst, i), Out(src, i)) \end{aligned}$$

If $D \in Dsys$ is a distributed system then we abbreviate $D_m(i)$ by $D(i)$. Distributed systems D and E have the same signature if their Loc, Lnk, src, dst, X, A, and Tag components are equal. Distributed systems D and E are compatible ($D \ \| \ E$) or satisfy the relation $D \ \subseteq \ E$ if they have the same signature and, for every $i \in D_{Loc}$, the message-automata , $D(i)$ and $E(i)$ are compatible or satisfy the relation $D(i) \ \subseteq \ E(i)$. And we define $D \ \oplus \ E$ by applying the $\oplus$ operation to each location.

5.4 Semantics of Distributed Systems and Message-Automata

The semantics of a distributed system D is the set of possible worlds w that are consistent with it. To be consistent, w must have the same signature as D, be a fair-fifo world, and respect the meanings of the six components $init$, pre, ef, $send$, $frame$, and $sframe$ of the message-automata at each location.

$$\begin{aligned} &Init(X, M, s) \ \equiv \\ &\quad \forall x : X.\ P \ =! \ M.init(x) \ \Rightarrow \ P(s.x) \end{aligned}$$

$$\begin{aligned} &MStep(X, A, Tag, Lnk, M, s, a, s', m) \ \equiv \\ &\quad \forall x : X.\ E \ =! \ M.ef(\langle kind(a), x\rangle) \ \Rightarrow \\ &\qquad s'.x \ = \ E(s, val(a) \ \in \ M.ds(x) \end{aligned}$$

$$
\begin{aligned}
&\wedge \\
&\forall x : X.\ F\ =!\ M.send(\langle kind(a), x\rangle)\ \Rightarrow \\
&\quad m\ =\ F(s, val(a)\ \in\ List(Message(Lnk, Tag, M.dout)) \\
&\wedge \\
&\forall x : X.\ L\ =!\ M.frame(x)\ \Rightarrow \\
&\quad kind(a) \notin L\ \Rightarrow\ s'\ =\ s\ \in\ State(X, M.ds)) \\
&\wedge \\
&\forall l : Lnk. \forall tg : Tag.\ L\ =!\ M.sframe(\langle l, tg\rangle)\ \Rightarrow \\
&\quad kind(a) \notin L\ \Rightarrow\ onlinktagged(l, tg, m) =\ \textbf{nil} \\
&\wedge \\
&\forall a : A.\ P\ =!\ M.pre(a)\ \Rightarrow\ P(s, a)
\end{aligned}
$$

$$
\begin{aligned}
&FairPre(D, w)\ \equiv \\
&\quad \forall i : w_{Loc}. \forall a : w_A.\ P\ =!\ D(i).pre(a)\ \Rightarrow \\
&\quad\ \exists^\infty t : \mathbb{N}.\ (\neg isnull(w_a(i,t))\ \wedge\ kind(w_a(i,t)) = local(a))\ \vee \\
&\qquad\quad \neg(\exists v : D(i).da(a).\ P(w_s(i,t), v))
\end{aligned}
$$

$$
\begin{aligned}
&PossibleWorld(D, w)\ \equiv \\
&\quad FairFifo(w) \\
&\quad \wedge\, w_X = D_X\ \wedge\ w_A = D_A\ \wedge\ w_{Tag} = D_{Tag} \\
&\quad \wedge\, w_{Loc} = D_{Loc}\ \wedge\ w_{Lnk} = D_{Lnk}\ \wedge\ w_{src} = D_{src}\ \wedge\ w_{dst} = D_{dst} \\
&\quad \wedge\, \forall i : w_{Loc}.\ Init(w_X, D(i), w_s(i, 0)) \\
&\quad \wedge\, \forall i : w_{Loc}. \forall t : \mathbb{N}.\ \neg isnull(w_a(i,t))\ \Rightarrow \\
&\qquad MStep(w_X, w_A, w_{Tag}, w_{Lnk}, D(i), w, w_s(i,t), w_a(i,t), w_s(i, t+1), w_m(i,t)) \\
&\quad \wedge\, FairPre(D, w)
\end{aligned}
$$

lemma

$$
\begin{aligned}
&\forall D_1, D_2 : Dsys. \\
&\quad D_1\ \subseteq\ D_2\ \Rightarrow \\
&\quad \forall w : World.\ PossibleWorld(D_2, w)\ \Rightarrow\ PossibleWorld(D_1, w)
\end{aligned}
$$

proof: D_1, D_2, and w all have the same signature, X, A, Tag, Loc, Lnk, src, and dst. For every $i \in Loc$, $M_1 = D_1(i)\ \subseteq\ M_2 = D_2(i)$. The definition of $PossibleWorld$ uses the automata $M \in \{M_1, M_2\}$ only in the context of conditional application of the finite partial functions, $M.init$, $M.pre$, $M.ef$, $M.send$, $M.frame$, and $M.sframe$, and also in some equality propositions over types $State(X, M.ds)$, $M.ds(x)$, and $List(Message(Lnk, Tag, M.dout))$. The conditional applications all occur positively, and so the statement for M_2 implies the statement for M_1, by the definition of $M_1\ \subseteq\ M_2$ and the lemma on conditional application of finite partial functions.

The equalities also occur positively, and, so the equality for M_2 implies the equality for M_1 because $State(X, M_2.ds)$ is a subtype of $State(X, M_1.ds)$, and similarly, $M_2.ds(x)$ is a subtype of $M_1.ds(x)$ and $List(Message(Lnk, Tag, M_2.dout))$ is a subtype of $List(Message(Lnk, Tag, M_1.dout))$.

5.5 Rules for Message-Automata

The message-automata in a distributed system put constraints on the possible worlds that can be executions of the system. We can state these constraints as rules on the event systems that come from the possible worlds. A rule of the form $@i\ M : \psi$ means that $\forall D : Dsys.\ \forall w : World.$

$$PossibleWorld(D, w)\ \wedge\ i \in D_{Loc}\ \wedge\ M\ \subseteq\ D(i)\ \Rightarrow\ Ev(w) \models \psi$$

It says that the event system of any possible world of any distributed system with at least M at location i will satisfy ψ.

5.5.1 Rule for initial clauses

$$@i \qquad \textbf{state}\ x : T;\ \textbf{initially}\ p(x) : p(x\ \textbf{initially}\ i)$$

proof: Let $i \in D_{Loc}$ and let $M = D(i)$ where

$$\text{state x:T; initially p(x)}\ \subseteq\ M$$

and let w be a possible world such that $PossibleWorld(D, w)$. Then $M.init(x)$ is defined and equal to $p(x)$, so by the $Init$ clause of $PossibleWorld$,

$$p(w_s(i, 0).x)$$

and this is, by definition of **initially**,

$$p(x\ \textbf{initially}\ i)$$

□

5.5.2 Rule for frame clauses

$$\begin{array}{ll} @i \qquad & \textbf{only}\ L\ \textbf{affects}\ x : \\ & \forall e@i.\ kind(e) \notin L\ \Rightarrow\ \neg(x\ \Delta\ e)\ \wedge \\ & \qquad (x\ \Delta\ e)\ \Rightarrow\ kind(e) \in L \end{array}$$

proof: Let $i \in D_{Loc}$ and let $M = D(i)$ where

$$\text{state x:T; only L affect x}\ \subseteq\ M$$

and let w be a possible world such that $PossibleWorld(D, w)$. Let $e = \langle i, t\rangle$ be an event in $Ev(w)$, then $\overline{a} = w_a(i, t)$ is not null. Let $k = kind(\overline{a})$ and suppose that $k \notin L$. Then $M.frame(x)$ is defined and equal to L, so by the definition of $PossibleWorld$, $s'.x = s.x$, where $s', s = w_s(i, t), w_s(i, t+1)$, and hence, by defintion of **when** and **after**, x **when** $e = x$ **after** e, so $\neg(x\ \Delta\ e)$. The second clause is the contrapositive of the first, just proved. In general, the contrapositive isn't constructively equivalent, but in this case, since the proposition $kind(e) \in L$ is decidable, it is.

□

5.5.3 Rule for effect clauses

$$
\begin{array}{ll}
@i & \textbf{state } x : T1;\ \textbf{action } k : T2; \\
 & k(v)\ \textbf{effect } x := f(s, v) : \\
 & \forall e@i.\ kind(e) = k \ \Rightarrow\ x \textbf{ after } e \ =\ f(s \textbf{ when } e, val(e))
\end{array}
$$

proof: Let $i \in D_{Loc}$ and let $M = D(i)$ where

state x:T1; action k:T2; effect k(v): x:= f(s,v) $\subseteq\ M$

and let w be a possible world such that $PossibleWorld(D, w)$. Let $e = \langle i, t\rangle$ be an event in $Ev(w)$, then $\overline{a} = w_a(i, t)$ is not null. Suppose that $kind(\overline{a}) = k$. Then $M.ef(\langle k, x\rangle)$ is defined and equal to $f(s, v)$, so by the definition of $PossibleWorld$, $s'.x = f(s, val(\overline{a}))$, where $s', s = w_s(i, t), w_s(i, t + 1)$, and hence, by defintion of **when**, **after**, and val, x **after** e = $f(s$ **when** $e, val(e))$

□

5.5.4 Rule for send clauses

$$
\begin{array}{ll}
@i & \textbf{action } k : T; \\
 & k(v)\ \textbf{sends } f(s, v) : \\
 & \forall e@i = k(v).\ \forall l : Lnk.\ sends(l, e) \ =\ onlink(l, f(s \textbf{ when } e, v))
\end{array}
$$

proof: Let $i \in D_{Loc}$ and let $M = D(i)$ where

action k:T; k(v): sends f(s,v) $\subseteq\ M$

and let w be a possible world such that $PossibleWorld(D, w)$. Let $e = \langle i, t\rangle$ be an event in $Ev(w)$, then $\overline{a} = w_a(i, t)$ is not null. Suppose that $kind(\overline{a}) = k$. Then $M.send(\langle k, x\rangle)$ is defined and equal to $f(s, v)$, so by the definition of $PossibleWorld$, $w_m(i, t) = f(w_s(i, t), val(\overline{a}))$, and hence, by defintion of **when**, $sends$, and val, $sends(e) = onlink(l, f(s$ **when** $e, val(e)))$

□

5.5.5 Rule for send frame clauses

$$
\begin{array}{ll}
@i & \textbf{only } L \textbf{ sends } \langle l, tg\rangle : \\
 & i = D_{src}(l) \ \Rightarrow\ \forall e'.\ kind(e') = rcv_l(tg) \ \Rightarrow\ kind(sender(e')) \in L
\end{array}
$$

proof: Let $l \in D_{Lnk}$ and let $M = D_{src}(l)$ where

only L sends ⟨l,tg⟩ $\subseteq\ M$

and let w be a possible world such that $PossibleWorld(D, w)$. Let e' satisfy $kind(e') = rcv_l(tg)$. Because $PossibleWorld$ implies $FairFifo$, we have $msg(l, tg, val(e')) = emsg(e') \in sends(sender(e'))$, where $sender(e') = \langle i, t\rangle$ is an event in $Ev(w)$, with $i = D_{src}(l)$. Then $sends(sender(e')) = w_m(i, t)$ and $M.sframe(\langle l, tg\rangle)$ is defined and equal to L, so by the definition of $PossibleWorld$, $kind(w_a(i, t)) \in L$. Thus $kind(sender(e')) \in L$.

□

5.5.6 Rule for precondition clauses

$@i$ **action** $k : T$;
$k(v)$ **precondition** $p(s, v)$:
$\forall e@i.\ kind(e) = k \ \Rightarrow\ p(s \textbf{ when } e, val(e))$
$\wedge\ (\exists e@i.\ (kind(e) = k)\ \vee\ (\exists e@i.\ \forall v : T.\ \neg p(s \textbf{ after } e, v))\ \vee\ \forall v : T.\ \neg p(s \textbf{ initially } i, v)$
$\wedge\ \forall e@i.\ (\exists e' \geq_{loc} e.\ (kind(e')) = k)\ \vee\ (\exists e' \geq_{loc} e.\ \forall v : T.\ \neg p(s \textbf{ after } e', v))$

proof: Let $i \in D_{Loc}$ and let $M = D(i)$ where

$$\text{action k:T; precondition k(v): p(s,v)} \ \subseteq\ M$$

and let w be a possible world such that $PossibleWorld(D, w)$. If $e = \langle i, t\rangle$ is an event in $Ev(w)$ then $\overline{a} = w_a(i, t)$ is not null. If $kind(\overline{a}) = k$ then $M.pre(k)$ is defined and equal to $p(s, v)$, so by the definition of $PossibleWorld$, we have

$$p(w_s(i, t), value(\overline{a}))$$

and this is the same as

$$p(s \textbf{ when } e, val(e))$$

So we have proved the first clause of the rule. Instantiating $FairPre(D, w)$ with k, we may choose $t' > t$ such that

$$(\neg isnull(w_a(i, t'))\ \wedge\ kind(w_a(i, t')) = local(a))\ \vee\ \neg(\exists v : M.da(k).\ p(w_s(i, t'), v))$$

In the first case, we let $e' = \langle i, t'\rangle$ and since $kind(w_a(i, t')) = k$ and k is not $null$, e' is an event in $Ev(w)$ and $e\ <_{loc}\ e'$ and $kind(e') = k$, so,

$$\exists e' \geq_{loc} e.\ kind(e') = k$$

In the second case,

$$\neg(\exists v : M.da(k).\ p(w_s(i, t'), v))$$

Find the least $t'' \leq t'$ such that for all t''' in the interval $(t'', t']$ the action $w_a(i, t''')$ is null. Then $t \leq t''$, since $w_a(i, t)$ is not null, and hence $w_a(i, t'')$ is not null, so we may choose e' to be $\langle i, t''\rangle$ and e' is an event in $Ev(w)$ and $e\ \leq_{loc}\ e'$. The state $w_s(i, t'' + 1)$ is the same as the state $w_s(i, t')$ because all the actions $w_a(i, t''')$ for t''' in the interval $(t'', t']$ are null and so, by the definition of $FairFifo$ the states are equal. Thus $w_s(i, t') = s$ **after** e' and we have

$$\neg(\exists v : M.da(k).\ s \textbf{ after } e', v))$$

Therefore, since $M.da(k)$ is defined and equal to T,

$$\exists e' \geq_{loc} e.\ \forall v : T.\ \neg p(s \textbf{ after } e', v)$$

Thus we have proved the third clause of the rule. The proof of the second clause is similar to the proof of the third clause, but since we are not starting with an event we also have to consider the possibility that no events occur at all at location i. By the same fairness clause we still get a t' such that

$$(\neg isnull(w_a(i, t'))\ \wedge\ kind(w_a(i, t')) = local(a))\ \vee\ \neg(\exists v : M.da(k).\ p(w_s(i, t'), v))$$

The first disjunct implies, as before, an e such that $kind(e) = k$. In the second case, we proceed as before to find the least $t'' \leq t'$ such that for all t''' in the interval $(t'', t']$ the action $w_a(i, t''')$ is null. If $w_a(i, t'')$ is not null, we proceed as before to produce an e' such that $\forall v : T. \neg p(s$ **after** $e', v)$. The new case is that $w_a(i, t'')$ might also be null. In this case $t'' = 0$, so all the actions $w_a(i, t)$ for $t \leq t'$ are null. In this case $w_s(i, t')$ is the same as $w_s(i, 0)$, so we conclude that $\forall v : T. \neg p(s$ **initially** $i, v)$, and that proves the second clause of the rule.

$\square$

6 Derivation Lemmas

For any label x we can constrain it to take a constant value at any location. **Constant Lemma**

$$\forall x : Lbl. \forall i : Loc. \forall T : \mathbb{U}. \forall v : T. \forall e@i.\ x \textbf{ when } e = v$$

proof: Use the rules for the frame clause and initial clause

$$@i \qquad \textbf{only } [] \textbf{ affects } x$$
$$@i \qquad \textbf{state } x : T;\ \textbf{initially } x = v$$

to get

$$\forall e@i.\ kind(e) \notin [] \Rightarrow \neg(x \,\Delta\, e) \wedge x \textbf{ initially } i = v$$

This implies

$$\forall e@i.\ \neg(x \,\Delta\, e) \wedge x \textbf{ initially } i = v$$

which implies

$$\forall e@i.\ \Delta(x, e) = 0 \wedge x \textbf{ initially } i = v$$

By lemma (29), this implies

$$\forall e@i.\ x \textbf{ when } e = v$$

$\square$

For any label k we can make a local action k that occurs exactly once at any location.
Once Lemma

$$\forall k : Lbl. \forall i : Loc. (\exists e@i.\ kind(e) = k) \wedge (\forall e@i <_{loc} e'. \neg(kind(e) = k \wedge kind(e') = k))$$

proof: Use the second clause of the rule for the precondition clause

$$@i \qquad \textbf{action } k : Unit;$$
$$k(v) \textbf{ precondition } \neg done$$

to get

$$(\exists e@i.\ (kind(e) = k) \vee (\exists e@i.\ \neg\neg done \textbf{ after } e) \vee \neg\neg done \textbf{ initially } i$$

Use the rule for the initial clause

$$@i \qquad \textbf{state } done : \mathbb{B};\ \textbf{initially } done = false$$

to get

$$\neg done \textbf{ initially } i$$

so we have

$$(\exists e@i.\ (kind(e) = k)\ \vee\ (\exists e@i.\ done \textbf{ after } e)$$

From this we first establish the first clause, $\exists e@i.\ (kind(e) = k$. The first case is what we are trying to prove. In the second case we have an e such that $loc(e) = i$ and $done$ **after** e but also $\neg done$ **initially** i. From this we can conclude, by lemma (30), that

$$\exists e'@i.\ done\ \Delta\ e'$$

Using the rule for the frame clause

$$@i \qquad \textbf{only } [k] \textbf{ affects } done$$

we get

$$\forall e@i.\ done\ \Delta\ e\ \Rightarrow\ kind(e) = k$$

From this we conclude $kind(e') = k$ which finishes the first claim.
To prove the second clause we use the rule for the effect clause

$$\begin{array}{ll} @i & \textbf{state } done : \mathbb{B};\ \textbf{action } k : Unit; \\ & k(v) \textbf{ effect } done := true \end{array}$$

to get

$$\forall e@i.\ kind(e) = k\ \Rightarrow\ done \textbf{ after } e$$

and the first clause of the rule for the precondition clause already introduced gives

$$\forall e@i.\ kind(e) = k\ \Rightarrow\ \neg done \textbf{ when } e$$

We can then prove by induction that

$$\forall e@i <_{loc} e'.\ \neg(kind(e) = k\ \wedge\ kind(e') = k)$$

If e' has no predecessors, then the statement is true. If $e_1 = pred(e')$ then $done$ **when** $e' = done$ **after** e_1 and if $e <_{loc} e'$ and $kind(e) = k\ \wedge\ kind(e') = k$ then $done$ **after** e and $\neg done$ **when** e', so we have $e \leq_{loc} e_1$ and $done$ **after** $e \neq done$ **after** e_1 This implies that there is an e_2 such that $e <_{loc} e_2\ \wedge\ e_2 \leq_{loc} e_1$ such that $done$ **when** $e_2 \neq done$ **after** e_2 and by the frame clause already introduced, this implies $kind(e_2) = k$. But then we have $e <_{loc} e_2$ and $e_2 <_{loc} e'$ and both e and e_2 have kind k, contradicting the induction hypothesis.

$\square$

For any tag tg, location i, and function f, we can cause a message with the tag tg containing the value $f(s)$ to be received on any link l with source i.

Send once Lemma

$$\forall tg : Lbl.\, \forall i : Loc.\, \forall f : State(i) \rightarrow T.\, \forall l : Lnk.\, src(l) = i \;\Rightarrow$$
$$(\exists e, e'.\, e \prec e' \;\wedge\; kind(e') = rcv_l(tg) \;\wedge\; val(e') = f(s \textbf{ when } e))$$
$$\wedge\, \forall e_1 @ i = tg.\, sends(e_1) = [msg(l, tg, f(s \textbf{ when } e))]$$

proof: Using the Once Lemma, we get

$$\forall i : Loc.\, (\exists e@i.\, kind(e) = tg) \;\wedge\; (\forall e@i <_{loc} e'.\, \neg(kind(e) = tg \;\wedge\; kind(e') = tg))$$

Using the rule for the sends clause

$$\begin{array}{ll} @i & \textbf{action } tg : Unit; \\ & tg(v) \textbf{ sends } [msg(l, tg, f(s))] \end{array}$$

we get

$$\forall e@i.\, kind(e) = tg \;\Rightarrow\; sends(e) = [msg(l, tg, f(s \textbf{ when } e))]$$

From these we can conclude that there is an event e at location i with kind tg and $msg(l, tg, f(s \textbf{ when } e)) \in sends(e)$. By lemma (25), we then conclude that

$$\exists e' >_{loc} e.\, kind(e') = rcv_l(tg) \;\wedge\; val(e') = f(s \textbf{ when } e)$$

$$\Box$$

Recognizer Lemma

$$\forall k : Lbl.\, \forall i : Loc.\, \forall p : State(i) \rightarrow V(k, i) \rightarrow \mathbb{P}.$$
$$\forall e'@i.\, x \textbf{ when } e' \;\Leftrightarrow\; \exists e <_{loc} e'.\, kind(e) = k \;\wedge\; p(s \textbf{ when } e, val(e))$$

proof: From the clause

$$@i \qquad \textbf{state } x : \mathbb{B};\; \textbf{initially } x = false$$

get x **initially** $i = false$. So from lemma (30),

$$\forall e'@i.\, x \textbf{ when } e' \;\Rightarrow\; \exists e <_{loc} e'.\, x \,\Delta\, e$$

From the frame clause

$$@i \qquad \textbf{only } [k] \textbf{ affects } x$$

we get

$$\forall e@i.\, x \,\Delta\, e \;\Rightarrow\; kind(e) = k$$

From the effect clause

$$\begin{array}{ll} @i & \textbf{state } x : \mathbb{B};\; \textbf{action } k : T; \\ & k(v) \textbf{ effect } x := \textbf{if } p(s, v) \textbf{ then } true \textbf{ else } x \end{array}$$

we get

$$\forall e@i.\ kind(e) = k \ \Rightarrow\ x \textbf{ after } e \ =\ p(s \textbf{ when } e, val(e)) \ \vee\ x \textbf{ when } e$$

This gives us,

$$\forall e'@i.\ x \textbf{ when } e' \ \Rightarrow\ \exists e <_{loc} e'.\ kind(e) = k \ \wedge\ p(s \textbf{ when } e, val(e))$$

To prove the other direction of the iff, we see that the effect clause gives

$$\forall e@i.\ kind(e) = k \ \wedge\ p(s \textbf{ when } e, val(e)) \ \Rightarrow\ x \textbf{ after } e$$

So it suffices to show that

$$\forall e'@i.\ \forall e <_{loc} e'.\ x \textbf{ after } e \ \Rightarrow\ x \textbf{ when } e'$$

This follows by induction from the frame clause and the effect clause since only action k can change x and can only change x from false to true.

□

Trigger Lemma

$$\begin{aligned}&\forall k, k' : Lbl.\ \forall i : Loc.\ \forall p : State(i) \rightarrow V(k,i) \rightarrow \mathbb{P}.\\ &(\forall e'@i = k'.\ \exists e <_{loc} e'.\ kind(e) = k \ \wedge\ p(s \textbf{ when } e, val(e)))\\ &\wedge (\forall e@i = k.\ p(s \textbf{ when } e, val(e)) \ \Rightarrow\ \exists e'.\ kind(e') = k')\end{aligned}$$

proof: Use the Recognizer Lemma to get a recognizer state variable x such that

$$\forall e'@i.\ x \textbf{ when } e' \ \Leftrightarrow\ \exists e <_{loc} e'.\ kind(e) = k \ \wedge\ p(s \textbf{ when } e, val(e))$$

Then add the precondition clause

$$\begin{array}{ll} @i & \textbf{action } k' : Unit;\\ & k'(v) \textbf{ precondition } x = true \end{array}$$

$$\begin{aligned}&\forall e@i.\ kind(e) = k' \ \Rightarrow\ x \textbf{ when } e\\ &\wedge (\exists e@i.\ (kind(e) = k') \ \vee\ (\exists e@i.\ \neg x \textbf{ after } e) \ \vee\ \neg x \textbf{ initially } i\\ &\wedge \forall e@i.\ (\exists e' \geq_{loc} e.\ (kind(e') = k') \ \vee\ (\exists e' \geq_{loc} e.\ \neg x \textbf{ after } e')\end{aligned}$$

The first clause $\forall e@i.\ kind(e) = k' \ \Rightarrow\ x \textbf{ when } e$ and the recognizer easily imply the first clause of the trigger. To show the second clause of the trigger, suppose $kind(e_i) = k$ and $p(s \textbf{ when } e, val(e))$. Then for any $e \ <_{loc}\ e'$ we will have $x \textbf{ when } e'$. From the third clause of the precondition rule we have

$$(\exists e' \geq_{loc} e.\ (kind(e') = k') \ \vee\ (\exists e' \geq_{loc} e.\ \neg x \textbf{ after } e')$$

But the second case contradicts what we have just shown, so we have

$$(\exists e' \geq_{loc} e.\ (kind(e') = k')$$

□

7 Leader Election in a Ring

7.1 Specification of Leader Election

A *flow* is a subset $F \subseteq Loc$ and a function $out : F \to Lnk$ such that

$$\forall i : F.\ src(out(i)) = i \ \wedge\ dst(out(i)) \in F$$

We define the function $n : F \to F$ by $n(i) = dst(out(i))$. If n is one-to-one and connected,

$$\forall i, j : F.\ n(i) = n(j) \ \Rightarrow\ i = j$$
$$\forall i, j : F.\ \exists k : \mathbb{N}.\ n^k(i) = j$$

Then the flow F is a ring R, and n is onto R so we may define functions p and in by $p(i) = n^{-1}(i)$ and $in(i) = out(p(i))$. We also define a distance $d(i, j) = \mu k \geq 1.\ n^k(i) = j$. Then,

$$i \neq p(j) \ \Rightarrow\ d(i, p(j)) = d(i, j) - 1$$

The *leader election problem* is to have exactly one member of a group announce that it is the leader. If we choose to have the announcement be the occurence of the action ”leader” at a location, then the specification of the leader election for a group R is the following

$$Leader(R) \quad \equiv\ \exists ldr : R. \quad (\exists e@ldr = leader.\ ) \wedge (\forall i : R.\ \forall e@i = leader.\ i = ldr)$$

7.2 Simple Leader Election

If R is a ring, and we have a one-to-one function, $uid : R \to \mathbb{N}$, then we claim that the following specification is derivable and refines $Leader(R)$.

$$
\begin{array}{ll}
 & LE(R, uid, in, out) \ \equiv\ \forall i \in R. \\
(1) & \exists e = rcv_{out(i)}(vote)(uid(i)). \\
(2) & \forall e = rcv_{in(i)}(vote)(v).\ v > uid(i) \ \Rightarrow\ \exists e' = rcv_{out(i)}(vote)(v). \\
(3) & \forall e' = rcv_{out(i)}(vote)(v).\ v = uid(i) \ \vee \\
 & \qquad\qquad \exists e = rcv_{in(i)}(vote)(v).\ e \prec e' \ \wedge\ v > uid(i) \\
(4) & \forall e = rcv_{in(i)}(vote)(uid(i)).\ \exists e'@i = leader. \\
(5) & \forall e'@i = leader.\ \exists e = rcv_{in(i)}(vote)(uid(i)).\ e \prec e'
\end{array}
$$

Theorem1 If (R, in, out, n, p) is a ring and $uid : R \to \mathbb{N}$ is 1-1, then

$$LE(R, uid, in, out) \ \Rightarrow\ Leader(R)$$

proof: Assuming the hypotheses, we let $m = max\{uid(i) \mid i \in R\}$ and let $ldr = uid^{-1}(m)$. Then the conclusion, $Leader(R)$ follows from the following four lemmas.

$\square$

Lemma1 $\forall i : R.\ \exists e = rcv_{in(i)}(vote)(uid(ldr))$.

proof: By induction on $d(ldr, i)$. If $d = 1$ then $in(i) = out(ldr)$, so by (1)

$$\exists e = rcv_{in(i)}(vote)(uid(ldr)).$$

If $d > 1$ then $p(i) \neq ldr$ and $d(ldr, i) < d(ldr, p(i))$, so by induction

$$\exists e = rcv_{in(p(i))}(vote)(uid(ldr)).$$

Then by (2), since $uid(ldr) > uid(p(i))$,

$$\exists e = rcv_{out(p(i))}(vote)(uid(ldr)).$$

and $out(p(i)) = in(i)$.

□

Lemma2 $\forall i, j : R.\ \forall e = rcv_{in(i)}(vote)(uid(j)).\ j = ldr \ \vee \ d(ldr, j) < d(ldr, i)$

proof: By induction on $\prec$. If $e = rcv_{in(i)}(vote)(uid(j))$ then by (3)

$$uid(j) = uid(p(i)) \ \vee \ \exists e = rcv_{in(p(i))}(vote)(uid(j)).\ e \prec e' \ \wedge \ uid(j) > uid(p(i))$$

In the first case, we have $j = p(i)$ and this implies $j = ldr \ \vee \ d(ldr, j) < d(ldr, i)$. In the second case, $uid(j) > uid(p(i))$ so $p(i) \neq ldr$ and, by induction, we have

$$j = ldr \ \vee \ d(ldr, j) < d(ldr, p(i))$$

But $d(ldr, p(i)) < d(ldr, i)$, since $p(i) \neq ldr$

□

Lemma3 $\forall i : R.\ \forall e'@i = leader.\ i = ldr$

proof: If $e' = leader_i$ then by (5)

$$\exists e = rcv_{in(i)}(vote)(uid(i)).\ e \prec e'$$

Then, by Lemma2, $i = ldr \ \vee \ d(ldr, i) < d(ldr, i)$. The second case is impossible, so $i = ldr$

□

Lemma4 $\exists e'@ldr = leader.$

proof: By (4), it is enough to show $\exists e = rcv_{in(ldr)}(vote)(uid(ldr))$. But this follows from Lemma1.

$\square$

Theorem2 $LE(R, uid, in, out)$

proof: We have to "implement" each of the five clauses, by deriving them from the rules for message-automata and event systems. Instantiate the Constant Lemma to get a state variable "me" such that

$$\forall e@i.\ me \textbf{ when } e = uid(i)$$

Instantiate the Send Once Lemma using $tg = vote, f(s) = s.me, l = out(i)$. This gives

$$\exists e, e'.\ kind(e') = rcv_{out(i)}(vote)\ \wedge\ val(e) = me \textbf{ when } e$$

and also

$$\forall e_1@i = vote.\ sends(e_1) = [msg(out(i), vote, me \textbf{ when } e_1)]$$

which implies

$$\exists e'.\ kind(e') = rcv_{out(i)}(vote)\ \wedge\ val(e) = uid(i)$$

which is clause (1) of $LE(R, uid, in, out)$, and also

$$\forall e_1@i = vote.\ sends(e_1) = [msg(out(i), vote, uid(i))]$$

Instantiate the Trigger lemma with $k = rcv_{in(i)}(vote),\ k' = leader,\ p(s, v) = (me = v)$ to get

$$\begin{array}{l}\forall i : Loc. \\ (\forall e'@i = leader.\ \exists e <_{loc} e'.\ kind(e) = rcv_{in(i)}(vote)\ \wedge\ uid(i) = val(e)) \\ \wedge\ (\forall e@ = rcv_{in(i)}(vote).\ uid(i) = val(e)\ \Rightarrow\ \exists e'.\ kind(e') = leader)\end{array}$$

This gives us clauses (4) and (5) of $LE(R, uid, in, out)$.
The rule for the sends clause

$$\begin{array}{ll}@i & \textbf{action } rcv_{in(i)}(vote) : \mathbb{N}; \\ & rcv_{in(i)}(vote)(v) \textbf{ sends if } v > me \textbf{ then } [msg(out(i), vote, v)] \textbf{ else } []\end{array}$$

gives, (since $(me \textbf{ when } e_i) = uid(i)$)

$$\forall e@ = rcv_{in(i)}(vote)(v).\ sends(e)\ =\ \textbf{if } v > uid(i) \textbf{ then } [msg(out(i), vote, v)] \textbf{ else } []$$

So

$$\forall e@ = rcv_{in(i)}(vote)(v).\ v > uid(i)\ \Rightarrow\ msg(out(i), vote, v) \in sends(e)$$

By lemma (25), this implies clause (2)

$$\forall e = rcv_{in(i)}(vote)(v).\ v > uid(i)\ \Rightarrow\ \exists e' = rcv_{out(i)}(vote)(v).$$

Finally, to derive clause (3) we need a send frame clause to constrain the actions that can send vote messages. In what we have derived so far, the only actions that send vote messages are the

$rcv_{in(i)}(vote)$ action and also the action $vote$ from the Send once Lemma. So we use the rule for the send frame clause

$$@i \quad \textbf{only } [rcv_{in(i)}(vote); vote] \textbf{ sends } \langle out(i), vote\rangle :$$
$$\forall e'.\ kind(e') = rcv_{out(i)}(vote) \ \Rightarrow\ kind(sender(e')) = rcv_{in(i)}(vote) \ \vee\ kind(sender(e')) = vote$$

From this we can prove clause (3) since if $e' = rcv_{out(i)}(vote)(v)$ then $emsg(e') = msg(out(i), vote, v) \in sends(out(i), sender(e'))$. Then either $kind(sender(e')) = vote$, in which case $sends(sender(e')) = [msg(out(i), vote, uid(i))]$ so $v = uid(i)$, or, for some v, $sender(e') = rcv_{in(i)}(vote)(v)$, in which case

$$sends(sender(e')) \ = \ \textbf{if } v > uid(i) \textbf{ then } [msg(out(i), vote, v)] \textbf{ else } []$$

so we must have $v > uid(i)$.

□

At this point we have proved the leader election specification, so we can extract from our proof a distributed system as an assignment of message-automata to locations. From this proof we get the following clauses for each $i \in R$:

$$\begin{array}{ll}
@i & \textbf{state } me : \mathbb{N};\ \textbf{initially } me = uid(i) \\
@i & \textbf{state } done : \mathbb{B};\ \textbf{initially } done = false \\
@i & \textbf{state } x : \mathbb{B};\ \textbf{initially } x = false \\
@i & \textbf{action } vote : Unit; \\
 & vote(v) \textbf{ precondition } \neg done \\
@i & \textbf{state } done : \mathbb{B};\ \textbf{action } vote : Unit; \\
 & vote(v) \textbf{ effect } done := true \\
@i & \textbf{action } vote : Unit; \\
 & vote(v) \textbf{ sends } [msg(out(i), vote, me)] \\
@i & \textbf{action } rcv_{in(i)}(vote) : \mathbb{N}; \\
 & rcv_{in(i)}(vote)(v) \textbf{ sends if } v > me \textbf{ then } [msg(out(i), vote, v)] \textbf{ else } [] \\
@i & \textbf{state } x : \mathbb{B};\ \textbf{action } rcv_{in(i)}(vote) : T; \\
 & rcv_{in(i)}(vote)(v) \textbf{ effect } x := \textbf{if } me = v \textbf{ then } true \textbf{ else } x \\
@i & \textbf{action } leader : Unit; \\
 & leader(v) \textbf{ precondition } x = true \\
@i & \textbf{only } [rcv_{in(i)}(vote); vote] \textbf{ sends } \langle out(i), vote\rangle \\
@i & \textbf{only } [] \textbf{ affects } me \\
@i & \textbf{only } [vote] \textbf{ affects } done \\
@i & \textbf{only } [rcv_{in(i)}(vote)] \textbf{ affects } x
\end{array}$$

7.3 Peterson Leader Election

$$Init(e, i) \quad \equiv \quad sends(e) = \textbf{if } sent \textbf{ when } e \textbf{ then } [] \textbf{ else } [msg(out(i), vote, uid(i))]$$

$$
\begin{aligned}
Forward(e,i,v) &\equiv sends(e) = \textbf{if } sent \textbf{ when } e \textbf{ then } [msg(out(i), vote, v)] \textbf{ else} \\
&\qquad\qquad [msg(out(i), vote, uid(i)); msg(out(i), vote, v)] \\
P(e,i,v) &\equiv last \textbf{ when } e > uid(i) \ \wedge\ last \textbf{ when } e > v \\
L(e,i,v) &\equiv active \textbf{ when } e \ \wedge\ start \textbf{ when } e \ \wedge\ v = uid(i)
\end{aligned}
$$

$PLE(R, uid, in, out) \ \equiv\ \forall i \in R.$

(1) $sent$ **initially** $i = false$

(2) $\forall e@i.\ sent\ \Delta\ e \ \Rightarrow\ kind(e) = init \ \vee\ kind(e) = rcv_{in(i)}(vote)$

(3) $(\exists e@i = init.\,) \ \wedge\ (\forall e@i = init.\ Init(e,i) \ \wedge\ sent$ **after** $e)$

(4) $start$ **initially** $i = true$

(5) $\forall e@i.\ start\ \Delta\ e \ \Leftrightarrow\ kind(e) = rcv_{in(i)}(vote)$

(6) $active$ **initially** $i = true$

(7) $\forall e@i.\ active\ \Delta\ e \ \Rightarrow\ kind(e) = rcv_{in(i)}(vote)$

(8) $\forall e@i.\ last\ \Delta\ e \ \Rightarrow\ kind(e) = rcv_{in(i)}(vote)$

(9) $\forall e = rcv_{in(i)}(vote)(v).\ last$ **after** $e = v$

(10) $\forall e = rcv_{in(i)}(vote)(v).\ sent$ **after** $e = true$

(11) $\forall e = rcv_{in(i)}(vote)(v).\ (\neg(active \textbf{ when } e) \ \vee\ start \textbf{ when } e) \ \Rightarrow$
$\neg(active\ \Delta\ e) \ \wedge\ Forward(e,i,v)$

(12) $\forall e = rcv_{in(i)}(vote)(v).\ (active \textbf{ when } e \ \wedge\ \neg(start \textbf{ when } e)) \ \Rightarrow$
$P(e,i,v) \ \Rightarrow\ \neg(active\ \Delta\ e) \ \wedge\ sends(e) = [msg(out(i), vote, uid(i))]$
$\wedge$
$\neg P(e,i,v) \ \Rightarrow\ active \textbf{ after } e = false \ \wedge\ sends(e) = []$

(13) $\forall e' = rcv_{out(i)}(vote)(v).\ \forall e.\ e \ \mapsto\ e' \ \Rightarrow\ kind(e) = rcv_{in(i)}(vote) \ \vee\ kind(e) = init$

(14) $\forall e = rcv_{in(i)}(vote)(v).\ L(e,i,v) \ \Rightarrow\ \exists e'@i = leader.$

(15) $\forall e'@i = leader.\ \exists e = rcv_{in(i)}(vote)(v).\ e \ <_{loc}\ e' \ \wedge\ L(e,i,v)$

Definition

$$
\begin{aligned}
L(e_1, e_2) &\equiv active \textbf{ when } e_1 \ \wedge\ e_1 \overset{+}{\mapsto} e_2 \ \wedge\ kind(e_2) = rcv_{in(loc(e_2))}(vote) \ \wedge \\
&\qquad \forall e.\ e_1 \overset{+}{\mapsto} e \ \wedge\ e \overset{+}{\mapsto} e_2 \ \Rightarrow\ \neg(active \textbf{ when } e) \\
LA(e_1, e_2) &\equiv active \textbf{ when } e_2 \ \wedge\ L(e_1, e_2) \\
R(e) &\equiv \|rcvs(in(loc(e)), vote, \textbf{before}(e))\| \\
S(e, m) &\equiv \|snds(out(loc(e)), vote, \textbf{before}(e, m))\| \\
S(e) &\equiv \|snds(out(loc(e)), vote, \textbf{before}(e))\|
\end{aligned}
$$

Lemma A Assuming $PLE(R, uid, in, out)$,

$$\forall e@i.\ S(e) = R(e) + \textbf{if } active \textbf{ when } e \ \wedge\ sent \textbf{ when } e \textbf{ then } 1 \textbf{ else } 0$$

proof: This is an example of the proof of an invariant. We prove invariants by induction on $<_{loc}$. If $first(e)$ then both sides of the equation are 0. Only $init$ and $rcv_{in(i)}(vote)$ events can affect the invariant. Each of them preserve it, since every receive causes one send, except for the case of a $Forward$ where $sent$ was false, which sends two, but also changes $sent$ from false to true, and the case of a receive that sends nothing (second case in clause (12)), which also changes $active$ from true to false. The $init$ event also preserves the invariant because it either sends nothing and leaves $active$ and $sent$ unchanged, or else it sends one message and changes $sent$ from false to true.

$\square$

Lemma B Assuming $PLE(R, uid, in, out)$,

$$\begin{array}{ll}
(a) & \forall e@i.\ start\ \textbf{when}\ e\ \Leftrightarrow\ R(e) \text{ is even} \\
(b) & \forall e@i <_{loc} e'.\ sent\ \textbf{after}\ e\ \Rightarrow\ sent\ \textbf{when}\ e' \\
(c) & \forall e@i <_{loc} e'.\ active\ \textbf{when}\ e'\ \Rightarrow\ active\ \textbf{after}\ e \\
(d) & \forall e@i.\ \neg(sent\ \textbf{when}\ e)\ \Rightarrow\ active\ \textbf{when}\ e
\end{array}$$

proof: (a) follows from (4) and (5). (b) follows from (2), (3), and (10), since the only events that can change $sent$ set it to $true$. (c) follows from (7), (11), and (12), since the only events that can change $active$ set it to $false$ or leave it unchanged. (d) follows by induction since the only event that can make $active$ false is a $rcv(vote)$ where $start = false$, but this must be preceded by a $rcv(vote)$ where $start = true$, and this, by (10), set $sent$ to $true$, and by (b) it will stay true.

$\square$

Lemma C Assuming $PLE(R, uid, in, out)$,

$$\forall e_2@ = rcv(vote).\ \exists e_1.\ L(e_1, e_2)$$

proof: By induction on $\prec$. For some e we have $e \mapsto e_2$. Let $i = loc(e)$. By (12) $kind(e)$ is either $init$ or $rcv_{in(i)}(vote)$. If it is $init$ then by (d) of Lemma B, $active$ **when** e, so $L(e, e_2)$. If $kind(e) = rcv_{in(i)}(vote)$, then if $active$ **when** e, we have $L(e, e_2)$. Otherwise, by induction, we have e_1 such that $L(e_1, e)$ and $e \mapsto e_2$ and $\neg(active$ **when** $e)$, so $L(e_1, e_2)$.

$\square$

Lemma D Assuming $PLE(R, uid, in, out)$,

$$\begin{array}{l}
kind(e_2) = rcv_{in(loc(e_2))}(vote)\ \wedge\ e \mapsto e_2\ \wedge\ \neg(active\ \textbf{when}\ e)\ \Rightarrow \\
(val(e_2) = val(e)\ \wedge\ R(e) = R(e_2))
\end{array}$$

proof: For some $m, e, m \mapsto e_2$. Then, $lnk(kind(e_2)) = in(loc(e_2)) = out(loc(e))$ and, by the corallary of the Fifo Lemma,

$$S(e, m) = R(e_2)$$

Since $\neg(active$ **when** $e)$ we must have $sent$ **when** e, so $m = 1$ and, by Lemma A

$$S(e) = R(e) + \textbf{if}\ active\ \textbf{when}\ e\ \wedge\ sent\ \textbf{when}\ e\ \textbf{then}\ 1\ \textbf{else}\ 0$$

Also, $val(e_2) = val(e)$, since e will $Forward$ and $sent$ **when** e is true.

□

Lemma E Assuming $PLE(R, uid, in, out)$,

$$L(e_1, e_2) \wedge R(e_2) = n \Rightarrow$$
$$R(e_1) = n - 1 \vee (n = 0 \wedge val(e_2) = uid(loc(e_1)))$$

proof: By induction on the length of the chain $e_1 \overset{+}{\mapsto} e_2$. Let

$$C(e, e') = R(e) = n - 1 \vee (n = 0 \wedge val(e') = uid(loc(e)))$$

For some e, m we have $e, m \mapsto e_2$. Then, $lnk(kind(e_2)) = in(loc(e_2)) = out(loc(e))$ and, by the corallary of the Fifo Lemma,

$$S(e, m) = R(e_2)$$

If $sent$ **when** e, then $m = 1$ and, by Lemma A

$$S(e) = R(e) + \textbf{if } active \textbf{ when } e \wedge sent \textbf{ when } e \textbf{ then } 1 \textbf{ else } 0$$

So, if $\neg(active$ **when** $e)$ then $val(e_2) = val(e)$ (since e will $Forward$ and $sent$ is true) and

$$R(e) = n$$

and $L(e_1, e)$ so by induction, $C(e_1, e)$ and hence, $C(e_1, e_2)$. And, if $active$ **when** e then $e_1 = e$ and

$$R(e) = n - 1$$

So $C(e_1, e_2)$. This leaves the case, $\neg(sent$ **when** $e)$. In this case, by (d) of Lemma B, we have $active$ **when** e, so $e_1 = e$. $kind(e)$ is either $init$ or $rcv_{in(loc(e))}(vote)$. If it is $init$, then $m = 1$ (since e sends only one message), and so

$$\|snds(out(loc(e)), vote, \textbf{before}(e))\| = n$$

But, this implies that $n = 0$ since when $\neg(sent$ **when** $e)$ then there have been no sends before e. Also, for the $init$ case, $val(e_2) = uid(loc(e))$ since that is what e sends. Hence $C(e_1, e_2)$. If e is a receive, then e sends two messages, first its own uid, and then the forwarded value. So $m = 1$ or $m = 2$. If $m = 1$ then as before, $n = 0$ and $val(e_2) = uid(loc(e_1))$ so $C(e_1, e_2)$. If $m = 2$, then $n = 1$ and $R(e_1) = 0 = n - 1$, and so $C(e_1, e_2)$ in this case as well.

□

Lemma F Assuming $PLE(R, uid, in, out)$,

$$\forall e_1.\ active \textbf{ when } e_1 \wedge \exists v : \mathbb{N}.\ msg(out(loc(e_1)), vote, v) \in sends(e_1) \Rightarrow$$
$$\exists e_2.\ LA(e_1, e_2)$$

proof: (Sketch) For any e, if e sends $msg(out(loc(e)), vote, v)$ then the message is received, so there is a receive event $e' = rcv_{in(loc(e'))}(vote)(v)$. If $active$ **when** e' then we are done. Otherwise e' will forward v. If this keeps happening, then the message will eventually come around the ring back to $loc(e_1)$ at some event e_2. We need to show that $active$ **when** e_2. We would have $L(e_1, e_2)$ so by Lemma E,

$$R(e_1) = R(e_2) - 1 \ \lor \ (R(e_2) = 0 \ \land \ val(e_2) = uid(loc(e_1)))$$

In the first case, e_1 is the receive just prior to e_2. Since e_1 sent something, it did not change $active$. Only receive events can change $active$, so $active$ is still true when e_2. In the second case, e_2 is the first receive, so $active$ must still be true when e_2.

□

Lemma G Assuming $PLE(R, uid, in, out)$,

$$L(e_1, e_2) \ \land \ R(e_2) \text{ is even } \Rightarrow \ val(e_2) = uid(loc(e_1))$$

proof: The inactive e in the chain from e_1 to e_2 are all receives and all forward the value they receive. By Lemma E, $R(e_1)$ is odd or $R(e_2) = 0 \ \land \ val(e_2) = uid(loc(e_1))$. In the second case we are done, and in the first, by (a) of Lemma B, $\neg(start$ **when** $e_1)$ so by clause (12) e_1 sends $uid(loc(e_1))$.

□

Lemma H Assuming $PLE(R, uid, in, out)$,

$$L(e_1, e_2) \ \land \ R(e_2) \text{ is odd } \Rightarrow \ val(e_2) = val(e_1)$$

proof: The inactive e in the chain from e_1 to e_2 are all receives and all forward the value they receive. By Lemma E, $R(e_1)$ is even so, by (a) of Lemma B, $(start$ **when** $e_1)$ so by clause (11) e_1 forwards. If $sent$ **when** e_1 then $val(e_2) = val(e_1)$. If $\neg(sent$ **when** $e_1)$ then $R(e_1) = 0$ and $R(e_2) = 1$, so we must have $e_1, 2 \ \mapsto \ e_2$ and so $val(e_2) = val(e_1)$.

□

Definition

$$A(n, i) \quad \equiv \quad \exists e@i = rcv_{in(i)}(vote).\ active \textbf{ when } e \ \land \ R(e) = 2n$$

Lemma I Assuming $PLE(R, uid, in, out)$,

$$\forall i \in R.\ \forall n : \mathbb{N}.\ A(n+1, i) \ \Rightarrow \ A(n, i)$$

proof: Follows easily from the definitions and (c) of the earlier lemma

□

Lemma J Assuming $PLE(R, uid, in, out)$,

$$\forall i : R.\ A(0, i)$$

proof: It's enough to show that every i receives at least one vote, since on the receipt of the first vote, $active$ will be true (active can only be changed to false on even numbered receives). To show that every i receives at least on vote, we note that $init$ must occur and it sends one unless $sent$ is true, but $sent$ is true only if $init$ did send one, or a receive occured. Every receive sends at least one vote, except when the second case of clause (12) happens, but this can only happen on even numbered receives, so at least one send must have occurred.

□

Lemma K Assuming $PLE(R, uid, in, out)$,

$$\forall n : \mathbb{N}.\ \exists i \in R.\ A(n, i)$$

proof: By induction on n. The base case is Lemma J. Suppose $\exists i \in R.\ A(n, i)$. We show that $\exists i \in R.\ A(n+1, i)$. Let m be the location with the maximum uid of all i such that $A(n, i)$. Then there is an e with $loc(e) = m$ and $active$ **when** e and $R(e) = 2n$. By clause (11), $msg(out(m), vote, val(e)) \in sends(e)$, so by Lemma G, $\exists e_2.\ LA(e, e_2)$. By lemma E, $R(e_2) = 2n + 1$, so there was a prior receive at e_1 for which $R(e_1) = 2n$ (and e_1 active since e_2 is active). So by lemma C there is an e_0 such that $L(e_0, e_1)$. By lemma G, $val(e_1) = uid(loc(e_0))$. We claim that $loc(e_0) = m$. (why?) If so, then, at e_2 we have $last$ **when** $e_2 = uid(m)$, hence, from clause (12), e_2 sends something. By Lemma F, there is an e_3 such that $LA(e_2, e_3)$, so we must have $A(n + 1, loc(e_3)$.

□

Location $i, j \in R$ are consecutive locations satisfying predicate P if

$$Conseq(P, i, j) \ \equiv\ i \neq j\ \wedge\ P(i)\ \wedge\ P(j)\ \wedge\ \forall k : \mathbb{N}.\ 0 < k < d(i, j)\ \Rightarrow\ \neg P(n^k(i))$$

Lemma L Assuming $PLE(R, uid, in, out)$,

$$\forall i, j \in R. \forall n : \mathbb{N}.\ Conseq(A(n), i, j)\ \Rightarrow\ A(n + 1, j)\ \Rightarrow\ \neg A(n + 1, i)$$

proof: Suppose $Conseq(A(n), i, j)$ and $A(n + 1, j)$. Then there is are $e'_1 \ <_{loc}\ e'_2\ <_{loc}\ e'_3$ at location j such that $active$ **when** e'_3 and $R(e'_3) = 2n_2$, $R(e'_2) = 2n + 1$, and $R(e'_1) = 2n$. Then there are $e_1\ <_{loc}\ e_2$ such that $L(e_1, e'_1)$ and $L(e_2, e'_2)$. Then $val(e'_1) = uid(loc(e_1))$ and $val(e'_2) = val(e_2)$, and hence, by (12) and because e'_3 is still active, $uid(loc(e_1)) > val(e_2)$. But we claim that both e_1 and e_2 have location i (why?). Also, $R(e_2) = 2n$ so if e_3 is the next receive at i, then by (12), $\neg(active$ **when** $e_3)$ so $\neg A(n + 1, i)$.

□

Lemma M Assuming $PLE(R, uid, in, out)$,

$$\forall i \in R. \exists e@i = leader.\ \ \Leftrightarrow\ \exists n : \mathbb{N}.\ A(n, i)\ \wedge\ \forall j \in R.\ j \neq i\ \Rightarrow\ \neg A(n, j)$$

proof: ($\Leftarrow$) If $A(n, i)$, then there is an e with $loc(e) = i$ and $active$ **when** e and $R(e) = 2n$. By clause (11), $msg(out(m), vote, val(e)) \in sends(e)$, so by Lemma G, $\exists e_2.\ LA(e, e_2)$. By lemma E, $R(e_2) = 2n + 1$, so there was a prior receive at e_1 for which $R(e_1) = 2n$ (and e_1 active since e_2 is active). So by lemma C there is an e_0 such that $L(e_0, e_1)$. By lemma G, $val(e_1) = uid(loc(e_0))$. We claim that $loc(e_0) = i$. (why?) But, because of e_1 we also have $A(n, loc(e_1))$ so $loc(e_1) = i$. Thus at e_1 location i received its own uid, and $R(e_1) = 2n$. So by clause (14) $\exists e@i = leader$.
($\Rightarrow$) By (15) there exist e where $active$ **when** e and $R(e)$ is even, and e is a receive of its own

uid. Then, by Lemma C, for some e_1, $L(e_1, e)$ and, by Lemma G, $val(e) = uid(loc(e_1))$, but this implies that $uid(loc(e_1)) = uid(loc(e))$, so $loc(e_1) = loc(e)$. All the intermediate e' on the chain from e_1 to e are inactive and have $R(e') = 2n$, so all have $\neg A(n, loc(e'))$, and they must include all locations other that $loc(e)$.

$\square$

Lemma N Assuming $PLE(R, uid, in, out)$,

$$Leader(R)$$

proof: Let $N(n) = \|\{i \in R \mid A(n,i)\}\|$. Then claim

$$\forall m : \mathbb{N}.\ N(m) = 1 \ \vee \ lessN(m+1)N(m)$$

The claim follows from Lemmas I, K, and L. Then claim

$$\forall n : \mathbb{N}.\ \exists m : \mathbb{N}.\ N(m) = n \ \Rightarrow \ \exists m : \mathbb{N}.\ N(m) = 1$$

This claim is proved by induction on n, using Lemma L and the previous claim.
Now, if $N(m) = 1$ then $\exists ldr \in R.\ A(m, ldr) \ \wedge \ \forall j \in R.\ j \neq ldr \ \Rightarrow \ \neg A(m, j)$, so by Lemma M, $\exists e@ldr = leader$. And, for any $i \in R$, if $kind(e_i) = leader$ then, by Lemma M, $\exists n : \mathbb{N}.\ A(n,i) \ \wedge \ \forall j \in R.\ j \neq i \ \Rightarrow \ \neg A(n, j)$. By considering the cases $n \leq m$ and $m \leq n$ and using (1), we see that $i = ldr$.

$\square$

8 View Synchrony

A *view* v is a pair $\langle v.id, v.set \rangle$ of a *view identifier* and a set of locations. There must be a transtitive, anti-reflexive ordering $<$ on the view identifiers. A view represents a named, ordered, guess about the current members of a group. Suppose P is a set of locations and $Iview$ is a function of type $p : P \rightarrow \{v : view \mid p \in v.set\}$ that assigns an initial view to each location p in P. Suppose that we have links from the locations in P to and from a service; that is, we have have functions $to : p : P \rightarrow \{l : Lnk \mid dst(l) = p\}$ and $from : p : P \rightarrow \{l : Lnk \mid src(l) = p\}$. Then the service provides *view synchrony* if

$$\begin{aligned}
Gpsnd(e, m, p) &\equiv e = rcv_{from(p)}(gpsnd)(m) \\
Grcv(e, m, q) &\equiv e = rcv_{to(loc(e))}(gprcv)(\langle m, q \rangle) \\
Safe(e, m, q) &\equiv e = rcv_{to(loc(e))}(safe)(\langle m, q \rangle) \\
Newview(e, v) &\equiv e = rcv_{to(loc(e))}(newview)(v) \\
View(e) &\equiv \textbf{if } first(e) \textbf{ then } Iview(loc(e)) \textbf{ else} \\
& \ \textbf{if } Newview(pred(e), v) \textbf{ then } v \textbf{ else} \\
& \ View(pred(e))
\end{aligned}$$

$$\begin{aligned}
VS(M, SM) \quad &\equiv \\
(1) \quad & \quad M(e_1, e_2) \ \Rightarrow \ e_1 \prec e_2 \ \wedge \ View(e_1) = View(e_2) \ \wedge
\end{aligned}$$

$$
\begin{array}{rcl}
 & & \exists m : Msg.\, Gpsnd(e_1, m) \ \wedge \ Gprcv(e_1, m, loc(e_1)) \\
(2) & & \forall e', m, q.\, Gprcv(e', m, q) \ \Rightarrow \ \exists e.\, M(e, e') \\
(3) & & M(e_1, e_2) \ \Rightarrow \ M(e_1', e_2) \ \Rightarrow \ e_1 = e_1' \\
(4) & & SM(e_1, e_2) \ \Rightarrow \ e_1 \prec e_2 \ \wedge \ View(e_1) = View(e_2) \ \wedge \\
 & & \exists m : Msg.\, Gpsnd(e_1, m) \ \wedge \ Safe(e_1, m, loc(e_1)) \\
(5) & & \forall e', m, q.\, Safe(e', m, q) \ \Rightarrow \ \exists e.\, SM(e, e') \\
(6) & & SM(e_1, e_2) \ \Rightarrow \ SM(e_1', e_2) \ \Rightarrow \ e_1 = e_1' \\
(7) & & loc(e) \in View(e) \\
(8) & & e_1 \ <_{loc} \ e_2 \ \Rightarrow \ View(e_1).id \leq View(e_2).id \\
(9) & & View(e_1).id = View(e_2).id \ \Rightarrow \ View(e_1) = View(e_2) \\
(10) & & SM(e_1, e_2) \ \Rightarrow \ \forall p : View(e_1).set.\, \exists e@p.\, e \prec e_2 \ \wedge \ M(e, e_2) \\
(11) & & M(a, a_1) \ \wedge \ M(a, a_2) \ \wedge \ M(b, b_1) \ \wedge \ M(b, b_2) \\
 & & \wedge\, loc(a_1) = loc(b_1) \ \wedge \ loc(a_2) = loc(b_2) \ \Rightarrow \\
 & & a_1 \ <_{loc} \ b_1 \ \Leftrightarrow \ a_2 \ <_{loc} \ b_2 \\
VS & \equiv & \exists M, SM : E \to E \to \mathbb{P}.\, VS(M, SM)
\end{array}
$$

$$
\begin{array}{rcl}
Quorum(v, P) & \equiv & 2 * \|v.set \ \cap \ P\| > \|P\| \\
Complete(P, L) & \equiv & \forall p : P.\, p \in [snd(x) \mid x \in L] \\
GpRcvd(e) & \equiv & [val(e') \mid e' \in rcvs(to(loc(e)), gprcv, \textbf{before}(e)) \ \wedge \ View(e') = View(e)] \\
Safe(e) & \equiv & [val(e') \mid e' \in rcvs(to(loc(e)), safe, \textbf{before}(e)) \ \wedge \ View(e') = View(e)]
\end{array}
$$

9 Consensus using View Synchrony

Here is an algorithm for consensus using VS:

$$\forall e = rcv_{in(i)}(propose)(c).\, vote \ \textbf{when} \ e = \perp \Rightarrow vote \ \textbf{after} \ e = \langle c, weak, \perp \rangle \tag{30}$$

$$\forall e = rcv_{to(i)}(gprcv)(\langle v, q \rangle).\, vote \ \textbf{when} \ e = \perp \Rightarrow vote \ \textbf{after} \ e = v \tag{31}$$

$$\forall e@i.\, \exists e' >_{loc} e.\, MayVote(e') \ \Rightarrow \ DoVote(e') \tag{32}$$

$$\forall e@i.\, CE(e) \ \Rightarrow \ SetWinner(e) \tag{33}$$

$$\forall e@i.\, DE(e) \ \Rightarrow \ elected \ \textbf{after} \ e = true \tag{34}$$

$$\forall e@i = rcv_{to(i)}(newview).\, voted \ \textbf{after} \ e = false \tag{35}$$

where

$$
\begin{array}{rcl}
MayVote(e) & \equiv & vote \ \textbf{when} \ e \neq \perp \ \wedge voted \ \textbf{when} \ e = false \ \wedge \\
 & & Quorum(View(e), P)
\end{array}
$$

$$
\begin{array}{rcl}
DoVote(e) & \equiv & sends(e) = [msg(from(i), gpsnd, vote \textbf{ when } e)] \ \wedge \\
 & & voted \textbf{ after } e = true \\
CE(e) & \equiv & Complete(P, GpRcvd(e)) \\
DE(e) & \equiv & Complete(P, Safe(e)) \\
MaxG(L) & \equiv & max([g \mid \langle\langle c, s, g\rangle, q\rangle \in L \ \wedge \ s = strong]) \\
StrongVotes(L) & \equiv & [c \mid \langle\langle c, s, g\rangle, q\rangle \in L \ \wedge \ s = strong \ \wedge \ g = MaxG(L)] \\
AllVotes(L) & \equiv & [c \mid \langle\langle c, s, g\rangle, q\rangle \in L] \\
Winner(L) & \equiv & \textbf{if } StrongVotes(L) \neq nil \textbf{ then } head(StrongVotes(L)) \textbf{ else} \\
 & & head(AllVotes(L)) \\
SetWinner(e) & \equiv & vote \textbf{ when } e = \langle Winner(RcvdVotes(e)), strong, View(e).id\rangle
\end{array}
$$

Here is a summary of the instructions to each location participating in the algorithm:

1. A vote is a triple $\langle c, s, g\rangle$ of a candidate, strength, and view identifier.
2. The current view is $primary$ if it contains a quorum.
3. If you receive a proposal for c and have no vote, then set vote=$\langle c, weak, \perp\rangle$.
4. If you receive a vote for $\langle c, s, g\rangle$ and have no vote, then set vote=$\langle c, s, g\rangle$.
5. If you have a vote $\langle c, s, g\rangle$ and have not yet voted, then, if $primary$, do $Gpsnd(\langle c, s, g\rangle)$ and set $voted$ to $true$.
6. If you have received votes from all members of the group, then select the winner w by
   (a) If any of the votes had strength $strong$ then choose the candidate from the earliest of those with the highest id.
   (b) Otherwise choose the candidate from the earliest vote.
   (c) Set vote=$\langle w, strong, current.id\rangle$.
7. If you have received confirmations (safe messages) of the votes from all members of the group then set elected=$true$
8. If you receive a new view, then set voted = $false$ and run the election again, but do not change your vote.

Definition (agrees)

$$
\begin{array}{rcl}
Agree(e, e') & \equiv & \exists g : viewid.\ g \geq View(e).id \ \wedge \\
 & & vote \textbf{ when } e' = \langle candidate(vote \textbf{ when } e), strong, g\rangle
\end{array}
$$

Lemma 1

$$
\begin{array}{ll}
\forall e. & DE(e) \ \Rightarrow \ \forall p : View(e).set. \\
 & \exists e'@p.\ e' \prec e \ \wedge \ View(e') = View(e) \ \wedge \ CE(e') \ \wedge \ Agree(e, e')
\end{array}
$$

proof: Suppose $DE(e)$, and let $p = loc(e)$ and $v = View(e)$. By definition of $DE(e)$, safe messages must have been received at p for votes from every member of the group, $v.set$. By view synchrony, all members of the group must have received all the votes, and received them in the same order. Thus, they all have had events e' satisfying $CE(e')$, and, by clause (6), they had all set their vote, $vote$ **when** e' to the same $\langle c, strong, v.id\rangle$. This includes location p, and since e has the same view as e', no new view has been received between e' and e so p has not changed its vote, and hence $candidate(vote$ **when** $e) = c$, and hence, $Agree(e, e')$.

□

Lemma 2

$$\forall e.\, DE(e) \;\Rightarrow\; \forall e'.\, CE(e') \;\wedge\; View(e').id \geq View(e).id \;\Rightarrow\; Agree(e, e')$$

proof: Given $DE(e)$, let $v = View(e)$. We prove, by induction on $\prec$,

$$\forall e'.\, CE(e') \;\wedge\; View(e').id \geq v.id \;\Rightarrow\; Agree(e, e')$$

Suppose $CE(e')$ and $w = View(e')$ and $w.id \geq v.id$. Let $q = loc(e')$. If $w.id = v.id$ then we use Lemma 1. By Lemma 1, there is an event e_q with $loc(e_q) = q$, and $View(e_q) = v$, such that

$$CE(e_q) \;\wedge\; Agree(e, e_q)$$

But there can only be one event e' per location and view for which $CE(e')$, so $e_q = e'$ and we have $Agree(e, e')$.
So assume $w.id > v.id$. Views v and w are both quorums, since otherwise no votes are sent in them. Thus there exists a location $p \in v.set \;\cap\; w.set$. Let $c = candidate(vote$ **when** $e)$. By Lemma 1, there is an event e_p such that $vote$ **when** $e_p = \langle c, strong, v.id\rangle$. Let e_{wp} be the event when new view w is received at location p. Location p will only change its vote by clause (6), because of a completed election, so by induction, $vote$ **when** $e_{wp} = \langle c, strong, g\rangle$ for some $g \geq v.id$. Thus, in the election in view w that is completed (for location q) at e', p has voted for $\langle c, strong, g\rangle$. Only a vote of the form $\langle d, strong, g'\rangle$ with $g' \geq g$ could beat p's vote. Such a vote would have to come from some location in $w.set$ that had set its vote to $\langle d, strong, g'\rangle$ at some event $e_d \prec e'$ (because $e_d <_{loc}$ voting in $w \prec$ vote received at $q <_{loc} e'$). The only way to set a vote to $\langle d, strong, g'\rangle$ is by a completed election in a view with identifier g'. Since $g' \geq g \geq v.id$, by the induction hypothesis, this implies $d = c$. Thus the winner of the election completed at e' is $\langle c, strong, g'\rangle$ for some $g' \geq v.id$ and we have $Agree(e, e')$.

□

Lemma (Consensus)

$$\begin{array}{l}\forall e_1, e_2.\, DE(e_1) \;\wedge\; DE(e_2) \;\Rightarrow \\ candidate(vote \textbf{ when } e_1) = candidate(vote \textbf{ when } e_2)\end{array}$$

proof: Let $v = View(e_1)$ and $w = View(e_2)$. Let $c_1 = candidate(vote$ **when** $e_1)$ and $c_2 = candidate(vote$ **when** $e_2)$. Assume, without loss of generality, $v.id \leq w.id$. Let $p = loc(e_2)$. By Lemma 1 , for some $e_p \prec e_2$,

$$CE(e_p) \;\wedge\; View(e_p) = w \;\wedge\; vote \textbf{ when } e_p = \langle c_2, strong, w.id\rangle$$

By Lemma 2,

$$\exists g : viewid.\; g \geq v.id \;\wedge\; vote \textbf{ when } e_p = \langle c_1, strong, g\rangle$$

Thus, $c_1 = c_2$.

□

10 Conclusion

10.1 Related Work

Winskel considered event systems in his 1980 Ph.D. thesis [60] and in other publications [61], inspired in part by Lamport [38]. He considered relationships to Petri nets and to domain theory and established the generality of event system, but he did not consider process extraction from proofs.

Hoare [33] and Milner [48] created extremely influential process calculi and their work is the basis for exploring process realizability of logical formulas [7, 49, 50], but they do not take up the issue of extraction from proofs either.

One of the most direct approaches to using proofs as processes is the work of Abramsky [4, 5] directed toward linear logic. These results are of considerable theoretical interest, but they have not been connected to practical verification.

Verification based on IO Automata [42] has been directly modeled in Nuprl [11] and PVS [6] and it is subsumed here as the special case where we reason directly about message automata. See also Vardi[59], Clarke and Emerson [17], Manna and Wolper [46], and Leonard and Heitmeyer [40] for different notions of sythesis that reference the meaning we intend.

Many logics used for practical reasoning and formal verification are based on programming logics [62, 55] or on *temporal logic* [44, 45], especially Unity [16] and $TLA+$ [39]. We look at the relationship between $TLA+$ and our work in the next section. Temporal logic has a limited role in synthesis [23]. Results on *knowledge in multi-agent systems* [24, 25, 26, 27, 28, 30] uses models with some of the properties of our worlds.

Abraham [1, 2, 3] uses classical multi-sorted first order logic to model processes whose state transitions are events. He also linearly orders events at a process and assumes a causal order on events generated by the local orders, capturing insights from Lamport [38, 38]. Our approach is related to his in that we use a higher-order constructive logic to define the models. His logic and ours deal explicitly with collections of events and with functions on these collections — another feature missing from temporal logic.

10.2 Relationship to *TLA+*

Lamport's $TLA+$ is a classical temporal logic of actions. He does not treat the issue of finding a constructive sublanguage from whose proofs it might be possible to extract distributed systems. Our work shows how this can be done. For a start, using the methods of Howe [34], the underlying logic of $TLA+$ can be embedded in a constructive logic such as Nuprl. Secondly, the temporal logic can be reduced to our event logic, as we now sketch.

Essentially the $TLA+$ process model arises by collapsing all locations to a single point and uniting all states into one "global state." Communication links are considered as state variables as well. The logic is based on describing the next state relation in a computation viewed as a single sequence of global states. This embedding would let us prove a result of the form:

> $TLA+$ is a sublanguage of the classical Event Logic obtained by adding the axiom
>
> $$\forall P : Prop_i.\ P \vee \neg P$$
>
> to the Event Logic defined in this paper.

10.3 Spaces of Events

The Event Logic formalism allows us to discuss classes and structured spaces of events. For example, *Strand spaces*[56] consist of sequences of send and receive messages at a process and sequences of send and receive messages of a penetrator process trying to break security. Thus strands are locations in event structures, and the ordering on elements is the same as our ordering on nonlocal events. These spaces model limitations on penetrators, and are used in specifying correctness criteria of encryption protocols [56, 29]. The methods of argument appear natural in our Logic of Events, and we can use inductive methods similar to those employed by Paulson in Isabelle [53].

References

[1] U. Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149:257–298, 1995.

[2] U. Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[3] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-stabilizing ℓ-exclusion. *Theoretical Computer Science*, 266:653–692, 2001.

[4] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

[5] S. Abramsky. Process realizability. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation: Proceedings of the 1999 Marktoberdorf Summer School*, pages 167–180. IOS Press, 2000.

[6] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of i/o automata with tame. *Automated Software Engineering*, 9(3):201–232, 2002.

[7] A. Barber, P. Gardner, M. Hasegawa, and G. D. Plotkin. From action calculi to linear logic. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 1998.

[8] D. A. Basin. An environment for automated reasoning about partial functions. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 101–110. Springer-Verlag, NY, 1988.

[9] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[10] G. Berry and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[11] M. Bickford and J. J. Hickey. Predicate transformers for infinite-state automata in Nuprl type theory. In *Proceedings of* 3^{rd} *Irish Workshop in Formal Methods*, 1999.

[12] M. Bickford, C. Kreitz, R. van Renesse, and X. Liu. Proving hybrid protocols correct. In R. Boulton and P. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag.

[13] K. Birman, R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. van Renesse, O. Rodeh, and W. Vogels. The horus and ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, 2000.

[14] K. Birman, R. Constable, M. Hayden, J. J. Hickey, C. Kreitz, R. van Renesse, O. Rodeh, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161. IEEE, 2000.

[15] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. MIT Press, Boston, 1993.

[16] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[17] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science (LNCS)*, pages 52–71. Springer–Verlag, 1982.

[18] W. R. Cleaveland, editor. *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[19] R. L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[20] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[21] R. L. Constable and J. Hickey. Nuprl's class theory and its applications. In F. L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[22] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[23] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[24] K. Engelhardt, R. van der Meyden, and Y. Moses. A refinement theory that supports reasoning about knowledge and time for synchronous agents. In R. Nieuwenhuis and A. Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *LNAI*, pages 125–141. Springer-Verlag, December 2001.

[25] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[26] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[27] J. Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

[28] J. Y. Halpern and R. Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[29] J. Y. Halpern and R. Pucella. On the relationship between strand spaces and multi-agent systems. In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 106–115, 2001.

[30] J. Y. Halpern and R. A. Shore. Reasoning about common knowledge with infinitely many agents. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1999.

[31] M. Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[32] J. J. Hickey, N. Lynch, and R. V. Renesse. Specifications and proofs for Ensemble layers. In Cleaveland [18], pages 119–133.

[33] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[34] D. J. Howe. Semantic foundations for embedding HOL in Nuprl. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[35] D. A. Karr, C. Rodrigues, J. Loyall, R. E. Schantz, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt. Application of the QuO quality-of-service framework to a distributed video application. In *International Symposium on Distributed Objects and Applications*, 2001.

[36] C. Kreitz. Automated fast-track reconfiguration of group communication systems. In Cleaveland [18], pages 104–118.

[37] C. Kreitz, M. Hayden, and J. J. Hickey. A proof environment for the development of group communications systems. In *Fifteen International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 317–332. Springer, 1998.

[38] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[39] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003.

[40] E. I. Leonard and C. L. Heitmeyer. Program synthesis from formal requirements specifications using apts. *Higher-Order and Symbolic Computation*. To appear.

[41] X. Liu, C. Kreitz, R. van Renesse, J. J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles*, December 1999.

[42] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[43] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

[44] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[45] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, 1995.

[46] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. and Syst.*, 6(1):68–93, 1984.

[47] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[48] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[49] R. Milner. Action structures and the π-calculus. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[50] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[51] C. Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Symposium on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[52] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[53] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society Press, 1997.

[54] R. V. Renesse, M. Hicks, M. Bickford, R. Constable, C. Kreitz, and L. Lorigo. Aspect-oriented programming of continuous media networks. Cornell, 2001.

[55] F. B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[56] F. J. Thayer, J. H. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[57] L. Théry. A machine-checked implementation of Buchberger's algorithm. *Journal of Automated Reasoning*, 26(2):107–137, February 2001.

[58] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburg, and D. Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[59] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, 1995.

[60] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[61] G. Winskel. An introduction to event structures. In J. W. de Bakker et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 345 in Lecture Notes in Computer Science, pages 364–397. Springer, 1989.

[62] J. Zwiers, W. P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *ICALP 1985*, pages 509–519, 1985.