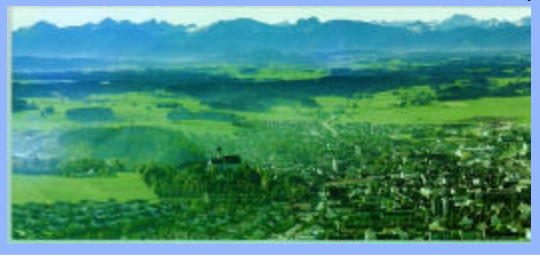
Lecture 4

Event Systems

This lecture is based on work done with Mark Bickford.

Marktoberdorf Summer School, 2003



Formal Methods

One of the major research challenges faced by computer science is providing a technology for building highly reliable software systems that perform well and evolve economically to ever-higher reliability and better performance.

One approach is using formal methods, tools based on logic. They are not widely adopted. There are several approaches to improving them:

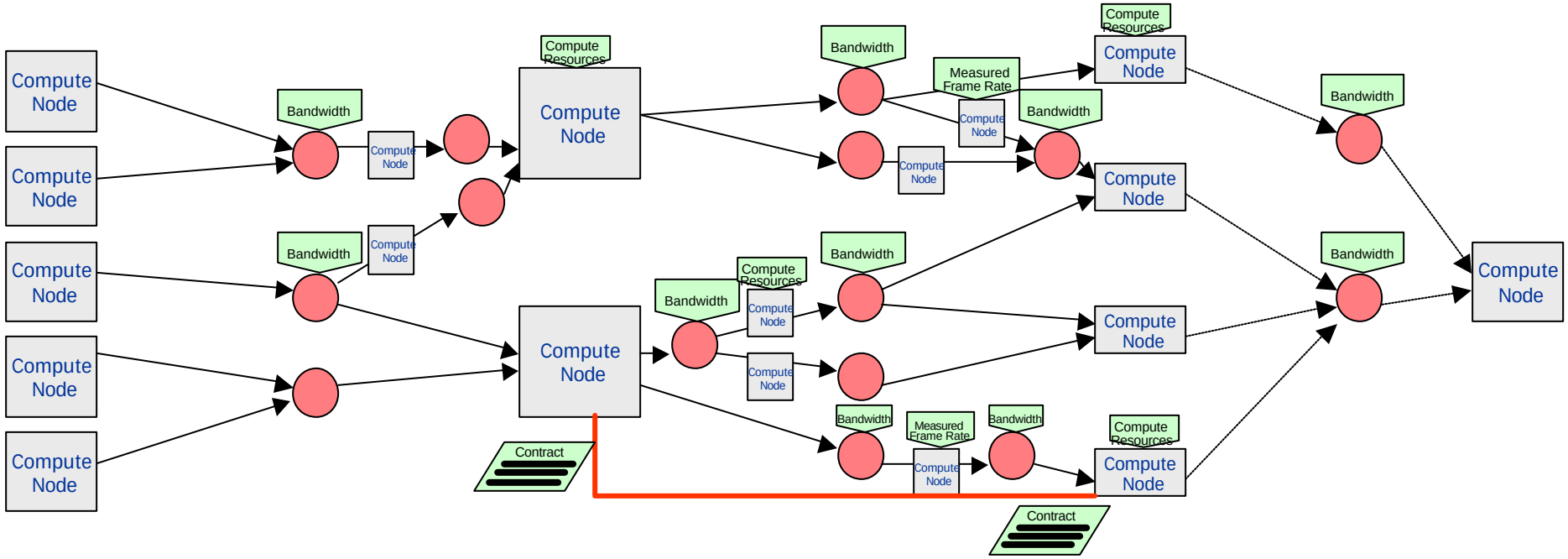
- More fully automated tools – type checkers, model checkers, decision procedures
- More effective interactive tools – provers

Our Experience

We have worked on distributed system verification for a decade, with Ken Birman's group at Cornell.

We discovered that we have accumulated sufficient knowledge that we could participate in protocol design and development at the speed of the designers and programmers.

Abstract Network Model – Media Net



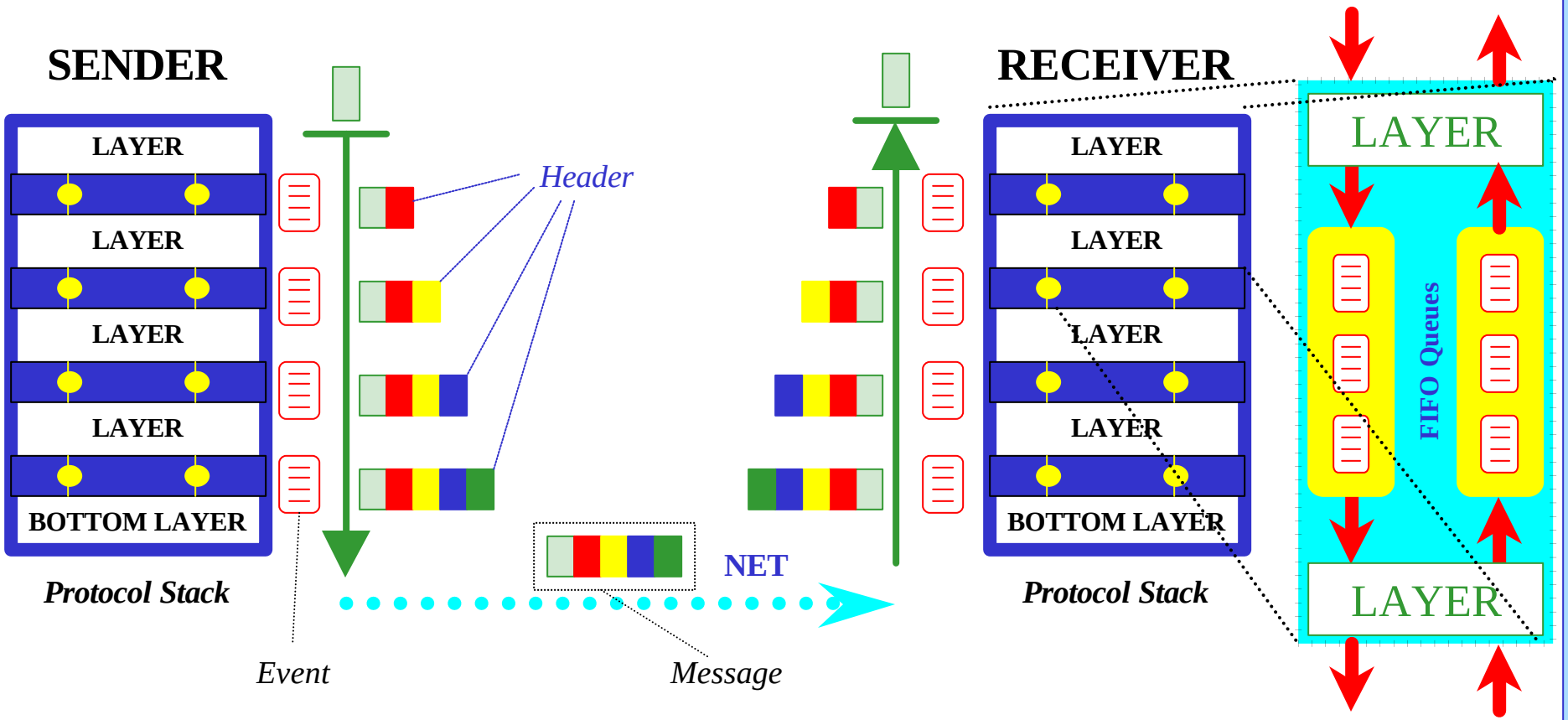
Concurrency Models

There is an enormous literature providing models for concurrent computation and distributed systems, e.g. Petri nets, process calculi, operational semantics, domain theories, machine models (IO Automata, message automata), model theory, etc.

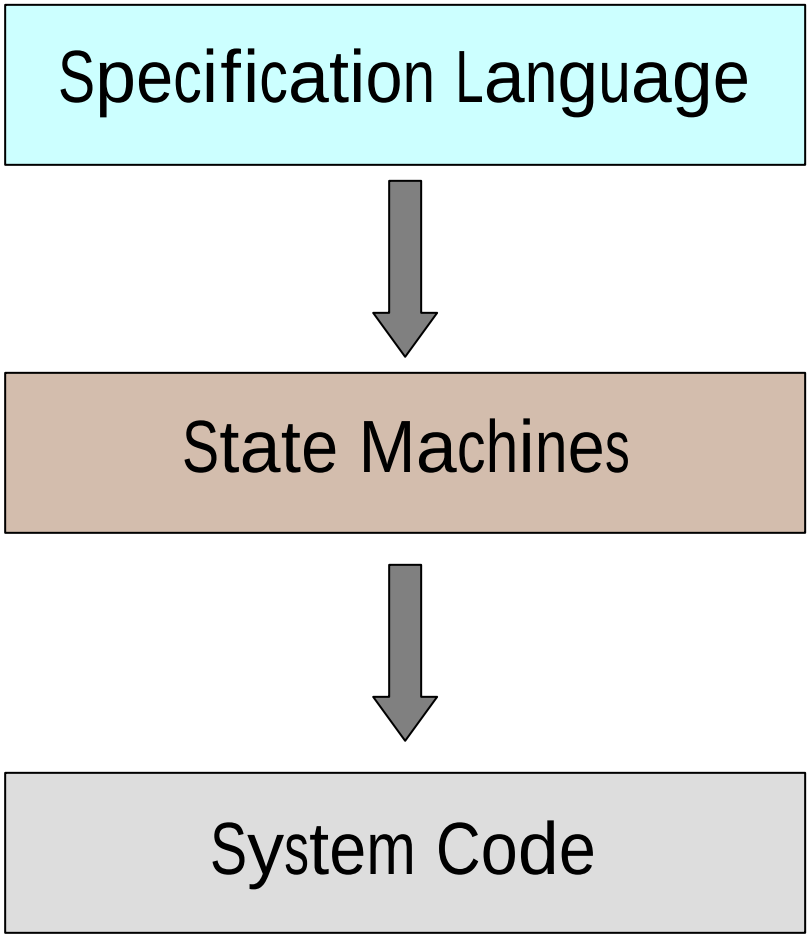
These models support several logics of processes, e.g. temporal logics, programming logics, etc.

Our work has been based on variants of Nancy Lynch's IO Automata. We presented some results in this summer school in 1999, with Jason Hickey.

Stack Model


SENDER
LAYER
LAYER
LAYER
LAYER
BOTTOM LAYER
Protocol Stack
Header
Event
NET
Message
RECEIVER
LAYER
LAYER
LAYER
LAYER
BOTTOM LAYER
Protocol Stack
LAYER
FIFO Queues
LAYER

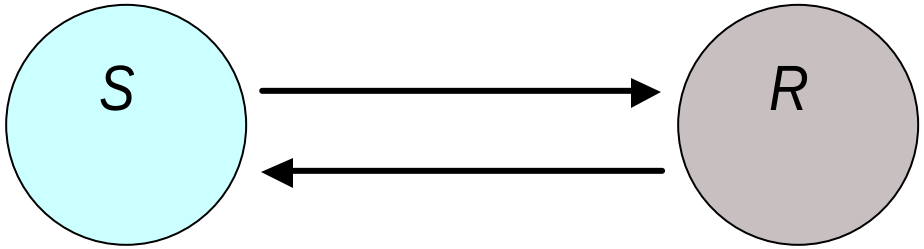
Event Systems Abstraction



Event Systems – Key Concepts

- Type *E* of events – abstract
- Type *Loc* of loci of events or process identifiers
- Type *Lnk* of communication links
- There are two kinds of events: (1) **local** actions and (2) **receives**; each can also send on links
- Each receive event has a unique sender
- Events at any locus are totally ordered, and causal order relates events among loci

Example – Two-Phase Handshake

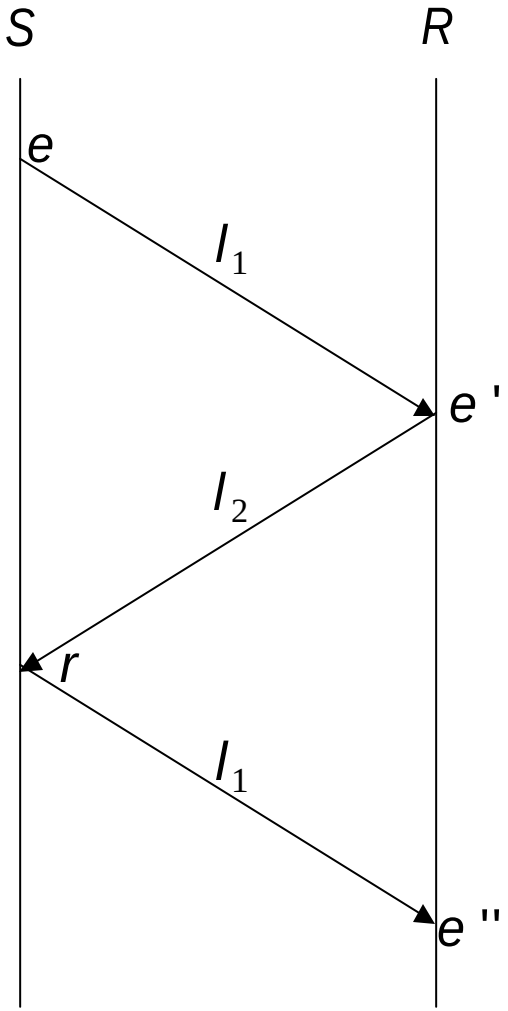


$$E_p = \{e : E \mid loc(e) = p\}$$

$$Snd_{p,l} = \{e : E_p \mid sends(e, l) \neq nul\}$$

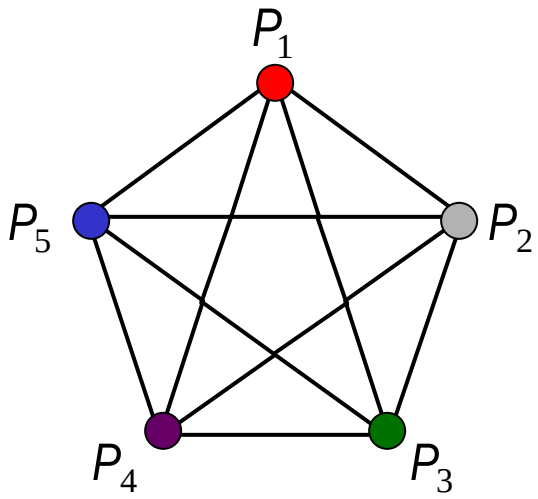
$$Rcv_{p,l} = \{e : E_p \mid kind(e)\ is\ receive\ on\ l\}$$

Communication Diagram



Distributed Systems

A distributed system *D* will be a graph *G* whose nodes are processes and whose links are communication channels.



We will treat processes abstractly in these lectures. In our verification work we use message automata and IO Automata at the nodes. We can also imagine other abstractions – e.g. active objects – as well as CML or Java programs.

Executions of Distributed Systems

Executions of distributed systems are event systems in a natural way. Executions are typically indexed by time, and that can be discrete, say $t \in \mathbb{N}$.

At each moment of time, a process at i is in a state, $s(i,t)$, and the links are lists of tagged messages, $m(l,t)$. At each locus i and time t, there is an action, $a(i,t)$, taken. The action can be null, i.e. no state change, no receives, hence no sends.

Fair-Fifo Executions

We assume executions are fair: channels are loss-less; and fifo: messages are received in the order sent.

1. Only the process at i can send messages on links originating at i.
2. A receive action at i must be on a link whose destination is i and whose message is at the head of the queue on that link.
3. There can be null actions that leave a state unchanged between t and $t+1$.
4. Every queue is examined infinitely often, and if it is nonempty, a message is delivered.
5. The precondition of every local action is examined infinitely often and if true the action is taken.

Event Systems of Fair-Fifo Executions

If w is a fair-fifo execution of a distributed system D, we can define an event system from it, $Ev(w)$.

The types $Loc, Lnk, Kind, Tag, Id$ are inherited from D.

The events E are the points $< i, t >$ locus i and time t, at which a non-null action, local or receive, occurred in w.

Specifying Protocols and Systems

Function specification:

$$\vdash ?x : A.\ ?y : B.\ R(x, y)\ \ ext\ \ f : A \rightarrow B$$

Protocol specification is part of a system specification.

$$\overline{x} : System \vdash ProcessSpec(x_i)\ \ ext\ \ P_i$$

We will focus on the *ProcessSpec.*

Axioms

Axiom 1: For every event e which sends a list of messages on link l, we can find an event e' at the destination of l at which all messages sent are received.

$$?e:E.\ ?l:Lnk.\ ?e':E.\ ?e'':E.$$
$$receive(e'') \Rightarrow e = sender(e'') \Rightarrow$$
$$link(e'') = l \Rightarrow (e' = e'' \vee loc(e') = dst(l))$$

Axiom 2: The predecessor function at each locus is one-to-one.

$$?e,e':E.\ loc(e) = loc(e') \Rightarrow$$
$$(pred(e) = pred(e') \Rightarrow e = e')$$

Axioms

Axiom 3 : The causal order $<$ is strongly well-founded.

$$\exists f : E \to \mathbb{N}.\ \forall e, e' : E.\ \left(e < e' \Rightarrow f\ (e) < f\ (e')\right).$$

Axiom 4 : If e is not the first event at a locus, then its predecessor is at the same location.

$$?e : E.\ \neg first\ (e) \Rightarrow loc\ \left(pred\ (e)\right) = loc\ (e).$$

Axiom 5 : If e is a *receive* event, then the locus of the sender is the source of the link of e.

$$?e : E.\ receive\ (e) \Rightarrow loc\ \left(sender\ (e)\right) = src\ \left(link\ (e)\right).$$

Axioms

Axiom 6 : Messages on a link are received in the order sent.

$$?e, e' : E.\ \big(rcv\ (e) \Rightarrow rcv\ (e')\big) \Rightarrow \big(link\ (e) = link\ (e')\big) \Rightarrow \big(sender\ (e) < sender\ (e')\big) \Rightarrow e < e'.$$

Axiom 7 : By convention, for any event except the first, to say that an observable x has a value v when event e happens at the locus of e is to say that x after the predecessor of e is v.

$$?e : E.\ \neg first\ (e) \Rightarrow ?x : Id.\ \big(x\ when\ e = x\ after\ pred\ (e)\big)$$

Deriving Algorithms and Protocols

Sequential program derivation from a specification:

$$\vdash \exists p : State \rightarrow State.\ ?s : State.\ R(s, p(s))$$

Nuprl supports extraction.

$$\vdash ?s : State.\ ?s' : State.\ R(s, s')\ ext\ p$$

Refinements for Programs

$\vdash ?x : A . ?y : B . R(x, y)$ *ext* $l\,x . cut\,(x, z . g(x, z); l(x))$

$x : A \vdash ?y : B . R(x, y)$ *ext* $cut\,(x, z . g(x, z); l(x))$

by cut L

1. $x : A, z : L \vdash ?y : B . R(x, y)$ *ext* $g(x, z)$

   by D o $\ulcorner g(x, z) \urcorner$

   $x : A, z : L \vdash R(x, g(x, z))$

2. $x : A \vdash L$ *ext* $l(x)$

by ____

Refinements for Systems

$\vdash ?D : System.\ ?es : ES(D).$

$R(es)\ ext\ \ Comp(pf_1(D_1, es_1), pf_2(D_2, es_2))$

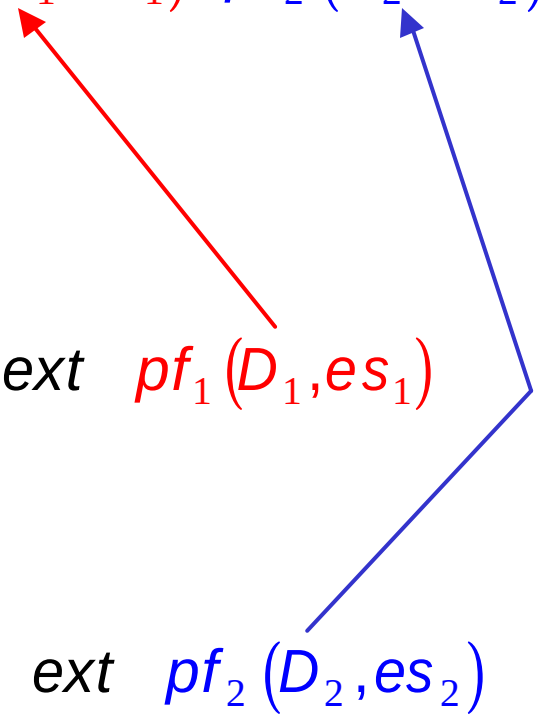
by Comp

1. $D_1 : System(G, Loc, Lnk)$

   $es_1 : ES(D_1) \vdash R_1(es_1)\ ext\ \ pf_1(D_1, es_1)$

2. $D_2 : System(G, Loc, Lnk)$

   $es_2 : ES(D_2) \vdash R_2(es_2)\ ext\ \ pf_2(D_2, es_2)$



Event Systems *es* of Distributed System *D*

For an (abstract) distributed system D, say that es is an event system of D, $es \in ES(D)$, if es is the event system $Ev(w)$ of an execution w of D.

Deriving the Two-Phase Handshake

We illustrate this process by deriving a protocol for the two-phase handshake from a proof that its specification is realizable.

(1) $\forall e_1, e_2 : Snd_{S,I_1}.\ ?r : Rcv_{S,I_2}.\ (e_1 < e_2 \Rightarrow e_1 < r < e_2)$

(2) $\forall e_1, e_2 : Snd_{R,I_2}.\ ?r_1, r_2 : Rcv_{R,I_1}.\ (e_1 < e_2 \Rightarrow r_1 < e_1 < r_2 < e_2)$

Deriving Handshake – 1

One way to achieve alternation of sends and receives at S is to introduce a boolean variable *rdy* and stipulate that S sends only when *rdy* is true. When a send on l_1 occurs, *rdy* is set to false.

Lemma 1: For any two processes S, R, and links l_1, l_2,

$$src(l_1) = S \quad \& \quad dst(l_1) = R, src(l_2) = R$$

$$\& \quad dst(l_2) = S, \quad \& \quad rdy \text{ in } Id,$$

we can realize an event system with the property that initially $rdy = true$ at S.

Call the realizer for Lemma 1 ES_1.

Deriving Handshake – 2

Lemma 2 : We can find a realizer that extends ES_1 such that for any non-empty type T ,

$$\forall e : E_S . (rdy \ when \ e = true) \Rightarrow$$
$$?e' : E_S . \ e < e' \wedge (?v : T . (rdy \ after \ e' = false \wedge$$
$$sends(e', I) = [I_1, val, v]) \vee rdy \ when \ e' = false).$$

Proof :

Infinitely often the process at S will examine its precondition $rdy = true$. If rdy has not changed, then the send will occur, and rdy will be set to false. Otherwise, rdy must be false at t , so some event e' after e set it to false.

Qed

Call the realizer for Lemma 2 ES_2.

Deriving Handshake – 3

Lemma 3 : We can find a realizer that extends ES_2 such that

$$\forall e : E_S .\ sends\ (e, I_1) \neq nil \Rightarrow rdy\ after\ e = false .$$

Proof :

We constrain the realizer of Lemma 2 to satisfy the frame condition that there are no actions that send on I_1 with tag *val* except for the one specified in Lemma 2.

Qed

Call the realizer for Lemma 3 ES_3.

Deriving Handshake – 3

Lemma 4 : We can find a realizer that extends ES_3 such that

$$\forall e : E_S .\ rcv\,(e, I_2) \Rightarrow rdy\ after\ e = true .$$

Proof :

We add to the process at S a response to any receive action on I_2, namely this receive will set *rdy* to true.

Qed

Call this realizer ES_4.

Deriving Handshake – 4

Lemma 5 : We can find realizers that extend ES_5 such that

$$\forall e : E_S .\ rcv(e, I_2) \Rightarrow ?e' : E_S .\ e < e' \wedge sends(e, I_1) \neq nil .$$

Proof :

We add the frame condition that only a receive on I_2 or reset of *rdy* to *false* can affect *rdy* .

Note, *rdy* after e=*true* by Lemma 4. Execution is fair, thus the precondition of the send will be checked infinitely often, hence after e. Only a reset can change *rdy* to *false* by the frame condition, and this happens only if the send on I_1 is executed. Thus there will be an event e' after e such that the send is executed.

Qed

Call this realizer ES_5.

Deriving Handshake – 4

Theorem 1: Any realizer extending ES_5 satisfies

$$\forall e_1, e_2 : Snd_{S,l_1}.\ \left(e_1 < e_2 \Rightarrow \exists r : Rcv_{S,l_2}.\ e_1 < r < e_2\right)$$

Proof :

Let e_1, e_2 be send events at S on link l_1. Suppose $e_1 < e_2$. By Lemma 3, *rdy* after $e_1 = false$. The only event that can set *rdy* to *true* is a receive event. the only way $e_1 < e_2$ is possible is that *rdy* when $e_2 = true$. Thus before e_2 and after e_1 there must be a receive event.

Qed

Lamport's TLA+ Specification – 2003

EXTENDS *Naturals* CONSTANT *Data*

VARIABLES *val* , *rdy* , *ack*

$$TypeInvariant \triangleq \land val \in Data \land rdy \in \{0,1\} \land ack \in \{0,1\}$$

$$Init \triangleq val \in Data \land rdy \in \{0,1\} \land ack = rdy$$

$$\begin{aligned} Send \triangleq\ & \land rdy = ack \\ & \land val' \in Data \\ & \land rdy' = 1 - rdy \\ & \land \text{UNCHANGED } ack \end{aligned}$$

$$\begin{aligned} Rcv \triangleq\ & \land rdy \neq ack \\ & \land ack' = 1 - ack \\ & \land \text{UNCHANGED } \langle val , rdy \rangle \end{aligned}$$

$$Next \triangleq Send \lor Rcv$$

$$Spec \triangleq Init \land \Box [Next]_{\langle val , rdy , ack \rangle}$$

THEOREM $Spec \Rightarrow \Box TypeInvariant$