Process Extraction in a Logic of Events

Robert L. Constable
Cornell University

Based on joint work with Mark Bickford

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proofs

Introduction

State of Semantics

research – thinking 20 years out

technology – very promising (Intel, Microsoft, AMD, …)

Next Steps

needs in distributed (network) computing

implementing a logic of events

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proofs

Intro to Event Systems

Design of the Logic of Events was strongly informed by the systems group and partners in several research teams.

We captured the conceptual level at which the protocol designers worked.

- event based analysis
- high level atomicity

We abstracted abstracted many high level distributed system concepts into an accessible logic with practical value:

many details can be automatically added by the extractors

- to MA
- to Java

Intro to Event Systems, continued

At the level of abstraction we use, one can see many interpretations, “good science stories,” as well as good “technology stories.”

The general setting is observables in time/space.

Space: locations at which “things happen”

Time: events happen at locations, but no global clock, only causal order on events

Two kind of events

- local action
- receive a message

Every event send messages (on links *l* with tag *t* value *v*)

Logic of Events, circa 2005

What distinguishes our event structures approach?

-based directly on Leslie Lamport's insights;

type theory captures them naturally

-used by distributed computing researchers, matches their intuitions

-integrated into LPE, hence into procedural programming

-completely formalized

-supports proofs-as-processes of synthesis and programming

-widely applicable: verification, optimization, documentation, security, performance

-organizes a fundamental set of concepts

Events with Order (EOrder)

Here is the signature of events with order.
We need the large type $\mathbb{D}$ of discrete small types (those with decidable equality).
Let $\mathbb{D}$ be this large type.

EOrder

$$E : \mathbb{D}$$

$$Loc : \mathbb{D}$$

$$Pred? : E \to (E + Loc)$$

$$Sender? : E \to (E + Unit)$$

EOrder Axioms

Axiom 1: For any event e that emits a signal, we can find an event e' by which e is received.

Axiom 2: The predecessor function, pred?, is injective.

Axiom 3: The predecessor relation, x L y, is strongly well founded, where x L y iff for y not the first event

x = pred?(y) or x = sender?(y). Namely, there is a function f from E to Nat such that x L y implies f(x) < f(y).

Properties of EOrder

We can squeeze a lot of information from these functions.

loc: E → Loc

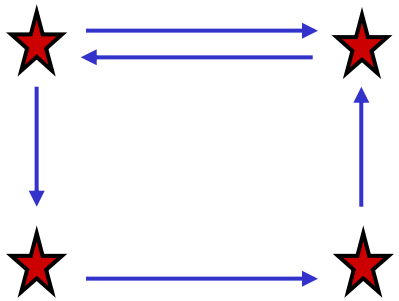
first: E → Bool

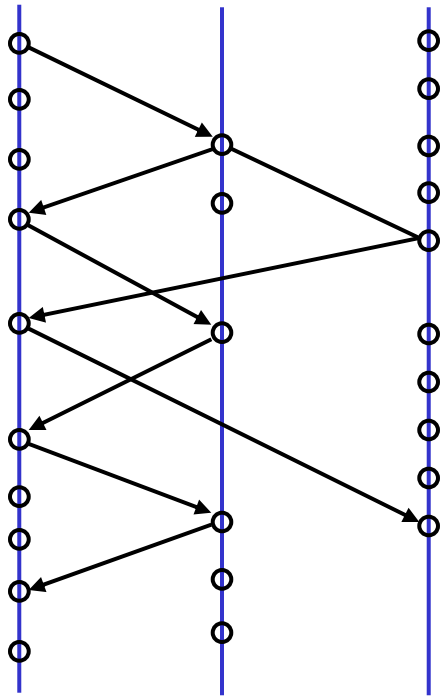
rcv: E → Bool

$<$ causal order (E → E → Prop)

$<_{loc}$ local linear order (E → E → Prop)

Events in Space/Time





Progression of Event Structures

We progressively define the following richer structures

EOrder – events with causal order

EValue – events have values, val(e)

EState – locations have state, temporal operators

initially, when, after

ECom – communication topology is given by a graph

Etime – real time is added

ETrans – transition function

Event Structures (with state)

Discrete types: Events (E), Loc, Actions (Act), Tag, Link (L)

$$loc: E \to Loc$$

$$kind: E \to KND$$

$$first: E \to B$$

$$pred: e:\{x:E \mid \neg first(x)\} \to \{x:E \mid loc(x)=loc(e)\}$$

$$snder: \{x:E \mid kind(x)=rcv\} \to E$$

$$<: E \to E \to B$$

$$Ty: E \to Type$$

$$val: x:E \to Ty(x)$$

$$\textbf{when} : x : Id \to e : E \to Ty(loc(e), x)$$

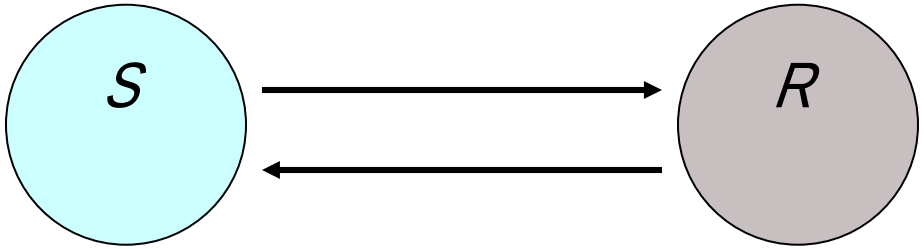
$$\textbf{after} : x : Id \to e : E \to Ty(loc(e), x)$$

$$\textbf{initial} : x : Id \to i : Loc \to Ty(i, x)$$

Outline

1. Introduction
2. Event Structures
   - Signatures and axioms
   - Sample specification
   - Computation model
3. Process Extraction from Proofs

Example – Two-Phase Handshake



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

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

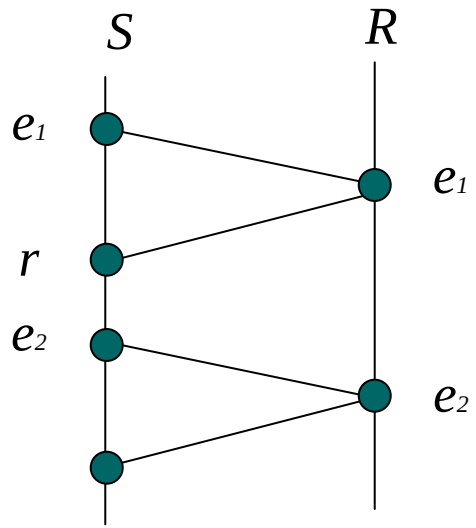
$$Rcv_{p,l} = \{e : E_p \mid kind(e)\ is\ receive\ on\ l\}$$

Deriving the Two-Phase Handshake

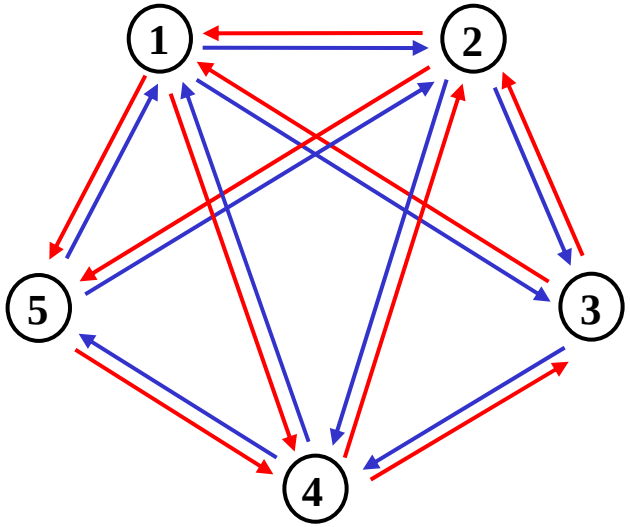
We illustrate this process by deriving a protocol for the two-phase handshake from a proof that its specification is realizable.

(1) $\forall e_1, e_2 : \mathtt{Snd}_{S,l_1}.\quad \exists r : \mathtt{Rcv}_{S,l_2}.\quad (e_1 < e_2 \Rightarrow e_1 < r < e_2)$

(2) $\forall e_1, e_2 : \mathtt{Snd}_{R,l_2}.\quad \exists r_1, r_2 : \mathtt{Rcv}_{R,l_1}.\quad (e_1 < e_2 \Rightarrow r_1 < e_1 < r_2 < e_2)$



Picture of a Computation



loc(1) loc(2) loc(3) loc(4) loc(5)

$\langle s_1, a_1, m_1 \rangle @ t$ $\langle s_2, a_2, m_2 \rangle @ t$ $\langle s_3, a_3, m_3 \rangle @ t$ $\langle s_4, a_4, m_4 \rangle @ t$ $\langle s_5, a_5, m_5 \rangle @ t$

$\langle s_1', a_1', m_1' \rangle @ t+1$ $\langle s_2', a_2', m_2' \rangle @ t+1$ $\langle s_3', a_3', m_3' \rangle @ t+1$ $\langle s_4', a_4', m_4' \rangle @ t+1$ $\langle s_5', a_5', m_5' \rangle @ t+1$

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 is a natural number, t in Nat.

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

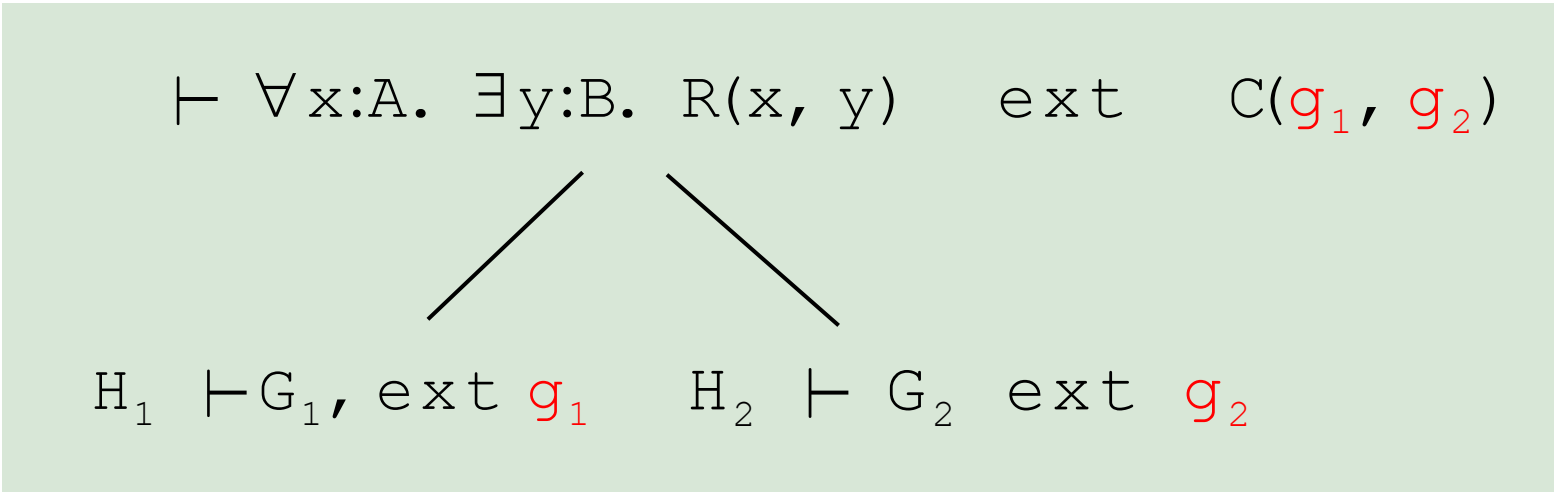
We assume executions are fair: channels are loss-less; and fifo: messages are received in the order sent.

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.

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proofs

Recall Functional Program Synthesis



Refinements for Programs

$\vdash \forall x : A.\ \exists y : B.\ R(x, y)$ ext $\lambda x.\ \mathtt{cut}(x, z.g(x, z); l(x))$

$x : A \vdash \exists y : B.\ R(x, y)$ ext $\mathtt{cut}(x, z.g(x, z); l(x))$

by cut L

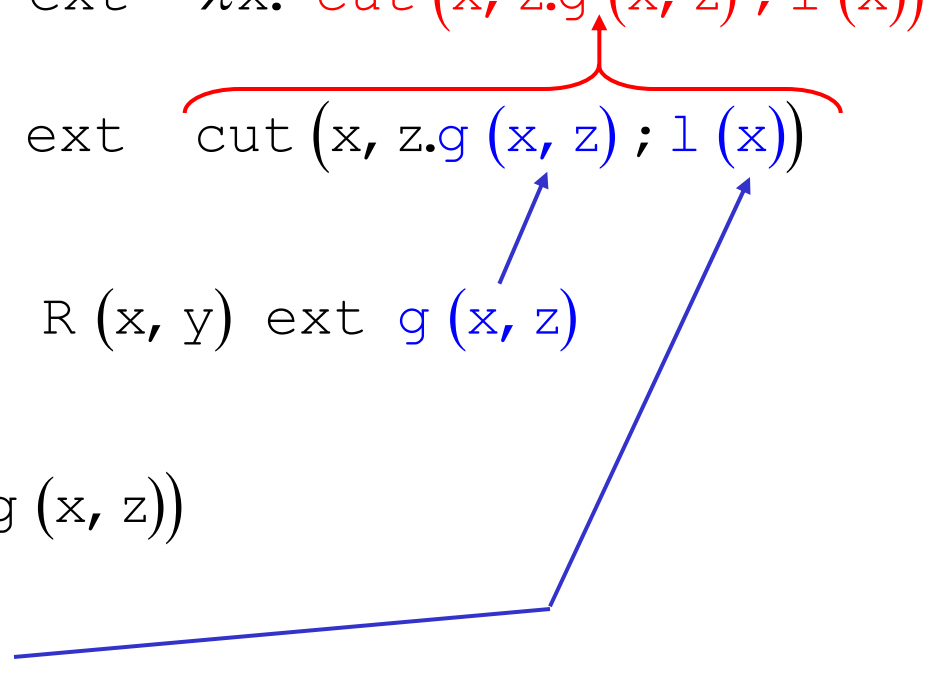
1. $x : A, z : L \vdash \exists y : B.\ R(x, y)$ ext $g(x, z)$

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

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

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

by ____



Refinements for Systems

$\vdash \exists D : \mathtt{System}.\ \forall es : \mathtt{ES}(D).$

$\mathtt{R}(es)\ \mathtt{ext}\ \ \mathtt{Comp}(\mathtt{pf}_1(D_1, es_1), \mathtt{pf}_2(D_2, es_2))$

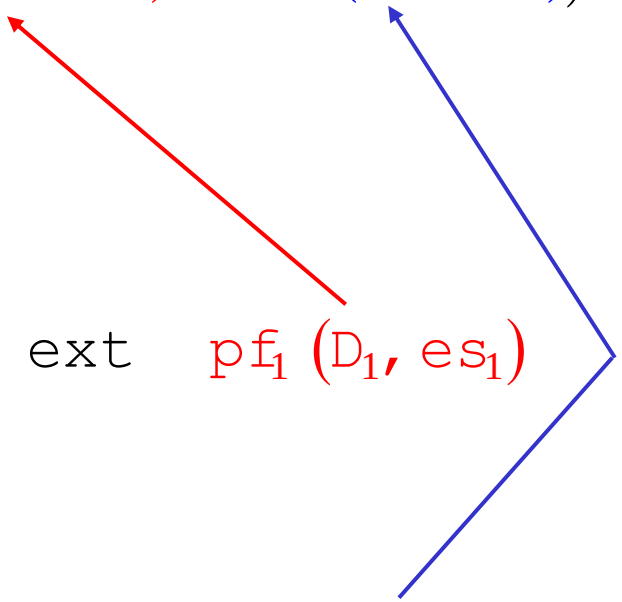
by Comp

1. $D_1 : \mathtt{System}(G, Loc, Lnk)$

$es_1 : \mathtt{ES}(D_1) \vdash \mathtt{R}_1(es_1)\ \ \mathtt{ext}\ \ \mathtt{pf}_1(D_1, es_1)$

2. $D_2 : \mathtt{System}(G, Loc, Lnk)$

$es_2 : \mathtt{ES}(D_2) \vdash \mathtt{R}_2(es_2)\ \ \mathtt{ext}\ \ \mathtt{pf}_2(D_2, es_2)$



Two-Phase Handshake Theorem

Theorem:

$\forall e_1, e_2 : Snd_s . e_1 < e_2 \Rightarrow \exists r : Rcv_s . e_1 < r < e_2$

What are the consequences of $e_1 < e_2$?

S sent two messages

Can we infer further consequences?

Relate send events to knowledge, create a boolean variable *rdy*

- Require *rdy* to be true when a send event occurs
- Require *rdy* to be false after a send event e

Two-Phase Handshake

Theorem:

$\forall e_1, e_2 : \mathrm{Snd}_s . e_1 < e_2 \Rightarrow \exists r : \mathrm{Rcv}_s . e_1 < r < e_2$

How to establish this by reasoning?

Suppose $e_1 < e_2$ are sends on link ℓ from S,

Then by L_1 (rdy after e_1) = false.

Since e_2 is a send, rdy must be true at e_2 by L_2.

Therefore some event e' before e_2 and after e_1

must set rdy to true.

By L_3 the event e' must be received from R on link ℓ'.

Let r be this e'.

If we have the lemmas, then the theorem is true.

Lemmas

L1. If S sends on link ℓ then it waits

$\forall$e:$\mathrm{Snd}_{s,\ell}$ (rdy after e) = false

L2. S sends on link ℓ only when rdy = true

$\forall$e: $\mathrm{Snd}_{s,\ell}$ (rdy when e) = true

L3. After S on link ℓ sends it is ready only after an acknowledgement on link ℓ'

$\forall$e : $\mathrm{Snd}_{s,\ell}$ e changes rdy to true $\Rightarrow$

e is a receive from R on link ℓ'.

Realizing the Lemmas - 1

L1. $\forall$ e : $\mathrm{Snds}_{,\ell}$. (rdy after e) = false

We can require that the sends on ℓ are caused by the action of setting rdy to false.

a: rdy:= false; send (ℓ, <tag, m >)

We also require that only action a can send on ℓ.

This is a frame condition.

L2. $\forall$e:$\mathrm{Snds}_{,\ell}$. (rdy when e) = true

We require the precondition on the action a that rdy = true

a: pre(rdy = true) $\Rightarrow$ rdy:= false; send (ℓ, < tag, m >)

Realizing the Lemmas - 2

L3: $\forall$ e : $\mathtt{Snd}_{s,\ell}$. (e Δ rdy) = true) $\Rightarrow$

kind(e) = rcv on ℓ'

We stipulate that a receive sets rdy to true

and that only a rcv or a can change rdy.

b: rcv(ℓ', m) $\Rightarrow$ rdy:= false

only [a,b] affect rdy (frame condition)

Handshake Message Automation

action local (a) sends on ℓ_1 < `tag, v` >

only [a] sends on ℓ_1

state `rdy` : $\mathbb{B}$; initially rdy = true

precondition a is rdy = true

effect local (a) rdy := false

action $\texttt{rcv}_{\ell_2}$ < `ack` > : `Atom`

effect rdy := true

only [local (a), $\texttt{rcv}_{\ell_2}$ <ack>] affect rdy

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proof

   Examples – two phase handshake

   -- leader election in a ring

   Realizers

   Realizability Theorems

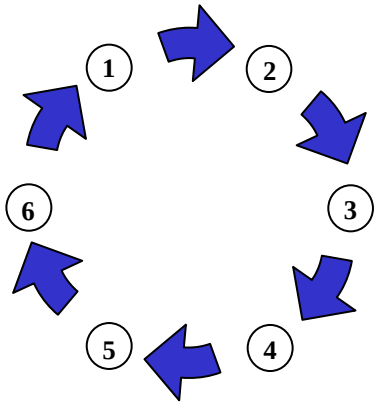
Specification for Leader Election in a Ring

Leader Election

In a Ring R of Processes with Unique Identifiers (uid's)

Specification

Let R be a non-empty list of locations linked in a ring



Let $n(i) = dst(out(i))$, the next location

Let $p(i) = n^{-1}(i)$, the predecessor location

Let $d(i,j)$-$\mu k \geq 1$. $n^k(i) = j$, the distance from i to j

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

Specification, continued

$$\text{Leader}(R,es) == \exists\, ldr: R.\ \big(\exists e@ldr.\ kind(e)=leader\big)\ \&\ \big(\forall i:R.\ \forall e@i.\ kind(e)=leader\ \Rightarrow\ i=ldr\big)$$

Theorem $\forall$ R:List(Loc). Ring(R)
$\exists$ D:Dsys(R). Feasible(D) &
$\forall$es: ES. Consistent(D,es). Leader(R,es)

Logically Decomposing the Leader Election Task

Let LE(R,es) == $\forall$i:R.

1. $\exists$e. kind(e)=rcv(out(i), <vote,uid(i)>)

2. $\forall$e. kind(e)=rcv (in(i), <vote,u>) $\Rightarrow$

   $\left(\text{u>uid(i)} \Rightarrow \exists e'.\text{kind}(e')=\text{rcv}\left(\text{out(i)},\text{<vote,u>}\right)\right)$

3. $\forall e'.$ [$\left(\text{kind}(e')=\text{rcv(out(i), <vote,uid(i)>)}\right) \vee$

   $\exists e.\left(\text{kind(e)=rcv(in(i), <vote,u>)} \& \left(e < e' \;\&\; \text{u> uid(i)}\right)\right)$]

4. $\forall$e@i. kind(e)=rcv(in(i),uid(i)). $\exists$e′@i. kind(e′)=leader

5. $\forall$e@i. kind(e)=leader. $\exists$e@i. kind(e)=rcv $\left(\text{in(i), <vote,uid(i)>}\right)$

Realizing Leader Election

Theorem $\forall R{:}\texttt{List}(\texttt{Loc})\,.\,\texttt{Ring}(R)$

$\exists D{:}\texttt{Dsys}(R)\,.\,\texttt{Feasible}(D)\,.$

$\forall es{:}\texttt{Consistent}(D, es)\,.\,\big(\texttt{LE}(R, es) \Rightarrow \texttt{Leader}(R, es)\big)$

Proof: Let $\texttt{m} = \max\{\texttt{uid}(i) \mid i \in R\}$, then $\texttt{ldr} = \texttt{uid}^{-1}(\texttt{m})$.

We prove that $\texttt{ldr} = \texttt{uid}^{-1}(\texttt{m})$ using three simple lemmas.

Intuitive argument that a leader is elected

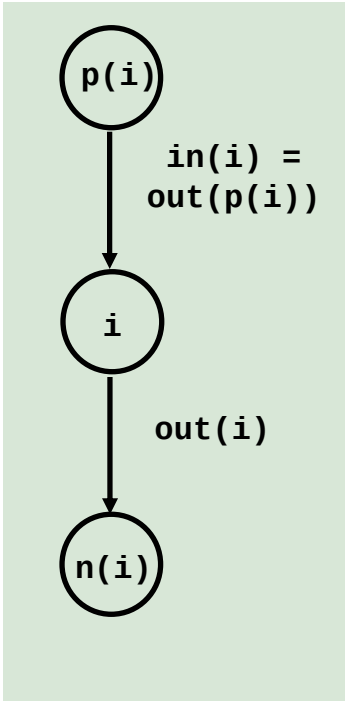
1. Every *i* will get a vote from predecessor for the predecessor.

2. When a process *i* gets a vote *u* from its predecessor with *u* > uid(*i*) it sends it on.

3. Every rcv is either vote of predecessor $rcv_{in(i)}$ for itself or a vote larger than process id before.

4. If a processor sets a vote for itself, it declares itself ldr.

5. If a processor declares ldr it got a vote for itself.



Lemmas

Lemma 1. $\forall i : R.\ \exists e @ i.\ kind(e) = rcv\ (in(i), \langle vote, ldr \rangle)$

By induction on distance of i to ldr.

Lemma 2. $\forall i, j : R.\ \forall e @ i.\ kind(e) = rcv\ (in(i), \langle vote, j \rangle).$

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

By induction on causal order of rcv events.

Lemma 3. $\forall i : R.\ \forall e' @ i.\ (kind(e') = leader \Rightarrow i = ldr)$

If $kind(e') = leader$, then by property 5, $\exists v @ i.rcv\ (in(i), \langle vote, uid(i) \rangle)$.
Hence, by Lemma 2 $i = ldr \vee (d(ldr, i) < d(ldr, i))$
but the right disjunct is impossible.

Finally, from property 4, it is enough to know
$\exists e.kind(e) = rcv\ (in(ldr), \langle vote, uid(ldr) \rangle)$
which follows from Lemma 1.

QED

Realizing the clauses of LE(R,es)

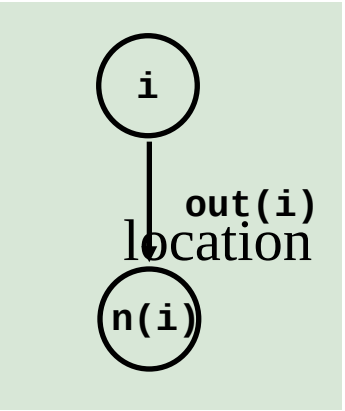
We need to show that each clause of `LE(R,es)` can be implemented by a piece of a distributed system, and then show the pieces are compatible and feasible.

We can accomplish this very logically using these Lemmas:

- Constant Lemma
- Send Once Lemma
- Recognizer Lemma
- Trigger Lemma

Implementing Clause 1 of LE(R,es)

1. ∃e.kind(e) = rcv (out(i), <vote, uid(i)>)



We need to send <vote,uid(i)> from each location i.

The Constant Lemma allows us to create a state variable me at each i with me=uid(i)

∀e @ i. (me **when** e) = uid(i)

The Send Once Lemma lets the process at i send uid(i)

∃e.kind(e) = rcv (out(i), <vote, uid(i)>)

Implementing Clauses 4, 5 of `LE(R,es)`

We can instantiate the Trigger Lemma to obtain

$$
\begin{array}{l}
\forall \texttt{i} : \texttt{Loc}. \\
\quad \forall \texttt{e}' \ @ \ \texttt{i}.\texttt{kind}(\texttt{e}') = \texttt{leader}. \\
\qquad (\exists \texttt{e} \ @ \ \texttt{i}.\ \texttt{e} < \texttt{e}'.\ \texttt{kind}(\texttt{e}) = \texttt{rcv}(\texttt{in}(\texttt{i}), \langle \texttt{vote}, \texttt{uid}(\texttt{i}) \rangle)) \\
\qquad \forall \texttt{e} \ @ \ \texttt{i}.\ \texttt{kind}(\texttt{e}) = \texttt{rcv}(\texttt{in}(\texttt{i}), \langle \texttt{vote}, \texttt{uid}(\texttt{i}) \rangle) \Rightarrow \\
\qquad\quad \exists \texttt{e}' \ @ \ \texttt{i}.\ \texttt{kind}(\texttt{e}') = \texttt{leader}
\end{array}
$$

Leader Election Message Automaton

state $me : \mathbb{N}$; initially $uid(i)$

state $done : B$; initially $false$

state $x : B$; initially $false$

action $vote$; precondition $\neg done$

effect $done := true$

sends $[msg(out(i), vote, me)]$

action $rcv_{in(i)}(vote)(v) : \mathbb{N}$;

sends if $v > me$ then $[msg(out(i), vote, v)]$ else[]

effect $x :=$ if $me = v$ then $true$ else x

action $leader$; precondition $x = true$

only $rcv_{in(i)}(vote)$ affects x

only $vote$ affects $done$

only $\{vote, rcv_{in(i)}(vote)\}$ sends $out\ (i), vote$

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proof

   Examples – two phase handshake

   -- leader election in a ring

   Realizers

   Realizability Theorems

Message Automata Clauses

- `@ i x : T` **initially** `= v`
- `@ i` **effect** `k(v : t)` **on** `x`

  `x := f state f`
- `@ i` **precondition** `a(v:t)` **is** `P state v`
- `k(v : T)` **sends on link l**

  `[tg1, f1 state v; ...; tgn, fn state v]`
- `@ i` **only** `[k1, ..., kn]` **sends on link l with tag** `tg`
- `A ⊕ B`**, where** `A, B` **are message automata**

Realizers, continued

There are many basic formulas of type theory that have atomic (or axiomatic) realizers, e.g., induction.

$$\big(\mathtt{P(0)}\ \&\ \forall \mathtt{x}{:}\mathbb{N}.\ \mathtt{P(x)} \Rightarrow \mathtt{P(x+1)}\big) \Rightarrow \forall \mathtt{x}{:}\mathbb{N}.\ \mathtt{P(x)}$$

realized by $\lambda\,(\mathtt{h}.\,\lambda(\mathtt{x}.\,\mathtt{ind(x;1of(h);u,v.2of(h)\,(u)\,(v)\,)\,)})$

There are six kinds of event specifications that have atomic (or axiomatic) realizers.

Realizable specifications

initial -- gives the value of a variable

effect -- defines a change of state based on an action

frame -- limits actions that can change a variable

pre -- takes an action if a precondition is true

sends -- sends tagged messages on a specified link

sframe -- limits actions that can send

Realizing Primitive Event Specifications

initial (using Message Automata)

@i state x:T; initially p(x)

realizes : p(initial (x,i))

frame

Define $x_{\triangle}e$ as x after e $\neq$ x when e

@i only L affects x

realizes : $\forall e@i.\ (x_{\triangle}e \Rightarrow \text{kind}(e) \in L)$

Effect lemma

effect

@i state x:T_1; action k:T_2;

k(v) effect x: = f(s when e, val (e))

realizes

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

x after e = f(s when e, val (e))

pre

@i action k:T; k(v) precondition p(s,v)

realizes:

```
∀e @ i. (kind(e) = k  ⇒  p(s where, val(e)))
 & ∀e @ i. ∃e' @ i. e ≤ e'  &
          (kind(e') = k  ∨  ∀v : T. ¬p(s after e', v))
 & ∃v : T. p(init(es)(i), v)  ⇒  ∃e : E. loc(e) = i
```

The skolem function in the ∀e ∃e' clause

gives a "schedule" for the action k.

Compound Realizers

Realizers are built by combining the six basic clauses. In the concrete case of Message Automata, the clauses are just joined by union. In the abstract setting, there is a combining operator, $R_1 \oplus R_2$.

es-realizer

Realizer

$\equiv_{\text{def}}$ rec(X.Unit

$+X\times X$

$+\text{Id}\times T{:}\text{Type}\times\text{Id}\times T$

$+\text{Id}\times\text{Type}\times\text{ID}\times(\text{KndList})$

$+\text{IdLnk}\times\text{Id}\times(\text{KndList})$

$+\text{Id}\times ds{:}x{:}\text{Id fp}\to\text{Type}\times\text{Knd}\times T{:}\text{Type}\times x{:}\text{Id}\times\big(\text{State}(ds)\to T\to\text{DeclaredType}(ds;x)\big)$

$+ds{:}x{:}\text{Id fp}\to\text{Type}$

$\times\text{Knd}$

$\times T{:}\text{Type}$

$\times\text{IdLnk}$

$\times dt{:}x{:}\text{Id fp}\to\text{Type}$

$\times\big((tg{:}\text{Id}\times(\text{State}(ds)\to T\to(\text{DeclaredType}(dt;tg)\text{ List})))\text{ List}\big)$

$+\text{Id}\times ds{:}x{:}\text{Id fp}\to\text{Type}\times\text{Id}\times T{:}\text{Type}\times(\text{State}(ds)\to T\to\text{Prop})$

Compatibility

Arbitrary compositions, $R_1 \oplus R_2$ might not be compatible. For example, R_1 might be a frame condition that says

R_1: only action `k` can change `x` and

R_2: action `k'` changes `x` for some $k \neq k'$

Also compatible realizers must have compatible types

R_1: declares `x` to be of type T_1

R_2: declares `x` to be of type T_2

We must have $T_1 \sqsubseteq T_2$ or $T_2 \sqsubseteq T_1$

Compatibility, continued

Compatibility is defined by 15 conditions from the 6 by 6 matrix of possibilities (half minus the diagonal). They are not decidable in theory but are in practice.

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proof

   Examples – two phase handshake

   -- leader election in a ring

   Realizers

   Realizability Theorems

Consistency

If `P` is any event specification, then the type theory expression of the goal is this

$$\vdash \exists D : DSyst.\ Feasible(D)\ \&\ \forall es{:}ES.\ Consistent(D, es) \Rightarrow P(es)$$

We say that `Feasible(D)` if `D` has at least one execution.

We say `Consistent(D,es)` provided `es` is an event system that arises from a possible execution of `D`.

Feasibility

A realizer R is feasible if it has an execution. For this to be possible, the clauses of R must be compatible and the types of variables, event values, and message content must be nonempty.

Computability

One of the main theorems of Bickford's massive library is that if distributed system D is feasible, then we can construct the possible executions, worlds, of it.

Moreover, from a world `W of D`, we can construct event systems for `D, es(D),` consistent with it.

```
Consistent(D,es)
```

This is a constructive proof, as are all in the library. So it defines the computational rules for the realizers given a schedule.

Running Distributed Systems Generate Event Structures

Theorem 1

For all DSys D, Feasible(D) $\Rightarrow$ $\exists$w:World. Possible(D,w)

Theorem 2

For all DSys D and all Possible Worlds w of D,

we can build an EventSystem es(w) Consistent(D, es(w)).