1 Introduction

Last time we defined the very important *causal order* relation, $e_1 < e_2$. It is the transitive closure of $\texttt{Pred}(x, y)$. Now we will show that this is a decidable relation, i.e.

$$\forall x : E.\forall y : E.((x < y) \vee \sim (x < y)).$$

We first look closely at the possible structure of this ordering and review its definition. We use message sequence diagrams (fig 1).

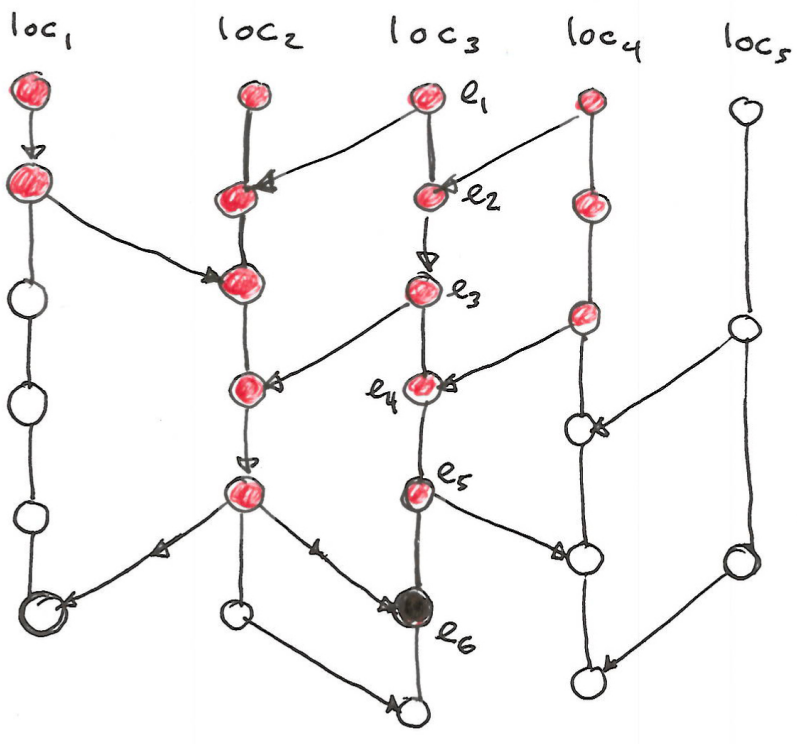

Figure 1: message sequence diagram

The red events show those which are causally before e_6 at $\texttt{loc}_3$. The arrows ($\rightarrow$) indicate messages are sent, so the target of the arrow is a receive event. The lines without arrows (—), say from e_5 to e_6 at $\texttt{loc}_3$ shows a simple predecessor relation, event e_5 might change the state at $\texttt{loc}_3$ and can affect what happens after e_6.

We can define the predecessor events at a location as a list, say at $\texttt{loc}_3$ the predecessor events of e_6 are the *list*

$\texttt{before}(e_6) = [e_5, e_4, e_3, e_2, e_1]$. The events causally before an event form a tree which we call the *cone* of the event.

Definition

$\texttt{before}(e)$ = if $\texttt{first}(e)$ then $[e, \text{nil}]$
else $\texttt{cons}(e, \texttt{before}(\texttt{pred}(e)))$

where $\texttt{cons}(e), [e_1, \ldots, e_n, \text{nil}]) =$
$[e, e_1, \ldots, e_n, \text{nil}]$.

Definition

$\texttt{cone}\ (e) =$

if $\texttt{rcv?}(e)$ then if $\texttt{first}(e)$
then $\langle e, \texttt{loc}(e), \texttt{cone}(\texttt{sender}(e))\rangle$
else $\langle e, \texttt{cone}(\texttt{pred}(e)), \texttt{cone}(\texttt{sender}(e))\rangle$
else if $\texttt{first}(e)$
then $\langle e, \texttt{loc}(e), nil\rangle$
else $\langle e, \texttt{cone}(\texttt{pred}(e)), nil\rangle$

One way to decide whether $e_1 < e_2$ is to form $\texttt{cone}(e_2)$ and then check to see if $e_1 \in \texttt{cone}(e_2)$ using the fact that equality of events is decidable.

Another way is to prove it by induction of causal order.

Theorem
$\forall x, y : E.(x < y) \vee \sim (x < y)$ - causal order is decidable.

The *induction principle for causal order* is this

Axiom
$\forall x : E.(\forall y : E.(y < x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x : E.P(x)$

This is like complete induction on $\mathbb{N}$, recall
$\forall n : ((\forall y.(y < n \Rightarrow P(y)) \Rightarrow P(n)) \Rightarrow \forall x.P(x)$

See the supplemental notes for Tue Nov. 15 for an argument that causal order is strongly well founded, given in the technical report: *A Causal Logic of Events in Formalized Computational Type Theory* .

In that article we do not discuss explicitly and fully the idea of a *choice function* sequence that correlates events to natural numbers $\mathbb{N}$. Thus that account does not take into consideration the kinds of *nondeterminism* that arise in real world systems. Such an account depends on some mechanism to introduce *choice* or *uncertainty* in the model. This choice arises from not knowing in advance the arrival order of messages destined for a location. We can conduct a "tour" of event orderings only after the fact. We do not have time in this course to discuss the mechanisms for non determinism in detail. We will however examine the notion in more detail when we study consensus protocols next week. We might at that time sketch the idea of *choice sequences*.

Next we examine the notion of *state* at a location. We imagine that processes at $\texttt{loc}_i$ have access to local state accessible by identifiers, another basic sort of the theory denoted $\texttt{Id}(x)$. The state stores data or values. We say $\forall i, x.(\texttt{Loc}(i)\ \&\ \texttt{Id}(x) \Rightarrow \exists v.\texttt{Value}(v)\ \&\ \texttt{St}(i, x, v))$

We assume only finitely many identifiers for the examples we consider, denoted $x_1, x_2 \ldots x_n$. We can specify the initial values x `initially` i is the initial value of identifier x at location i.

We also introduce the *temporal operators*
x `when` e
x `after` e

Axiom
$\forall e : E.\neg\texttt{first}(e) \Rightarrow (x\ \texttt{when}\ e) = (x\ \texttt{after}\ \texttt{pred}(e))$

Definition
$x \triangle e$ `iff` x `after` $e \neq x$ `when` e
This is the *change operator.*

2 Message Automata Realizers for event statements

1. $@i\ p(x\ \texttt{initially}\ i)$ `realizes` $\forall e@i.\ \texttt{first}(e) \Rightarrow p(x\ \texttt{when}\ e)$

2. $@i\ \texttt{rcv}_i(v)$ `event` $x := f(s, v)$ for s the state at i *realizes*
   $\forall x : \texttt{Value}.\forall l : \texttt{Link}.\forall e@\ i.e = \texttt{rcv}_i(v) \Rightarrow (x\ \texttt{after}\ e) = f(s, v)$

3. $@i\ \texttt{send}_l(v)$ *realizes*
   $\forall e@i.(\texttt{kind}(e) = \texttt{send}_e(v)) \Rightarrow \exists e'@\ \texttt{dest}(e).\ \texttt{kind}(e') = \texttt{rcv}_e(v)$
   $\&\ \texttt{sender}(e') = e$

4. $@i$ only L affects x for L a list of actions *realizes*
   $\forall e@i.\ \texttt{kind}(e) \notin L \Rightarrow \neg(x \triangle e)\ \&\ (x \triangle e \Rightarrow \texttt{kind}(e) \in L)$
   This is called a *frame condition*

5. $@i$ only L sends $\langle v, tag\rangle$ *realizes*
   $\forall e@\ \texttt{dest}(l).\texttt{kind}(e) = \texttt{rcv}_l(\langle v, tag\rangle) \Rightarrow \texttt{kind}(\texttt{sender}(e)) \in L.$
   This is called a sends *frame condition.*

Now we are prepared to treat the *acknowledgement protocol* from Nov. 10 in more detail. Recall the context.

As an example, suppose we want a system of processes P with the property that two of its processes, say S and R connected by link l_1 from S to R and l_2 from R to S should operate using explicit acknowledgement. So when S sends to R on l_1 with tag tg, it will not send again on l_1 with this tag until receiving an acknowledgement tag, ack, on l_2. The specification can be stated as a theorem about event structures.

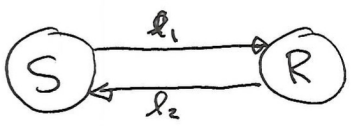

Figure 2: S and R are processes linked by reliable FIFO communication channels $l_1,\ l_2$.

Theorem 1 *For any distributed system P with two designated processes S and R linked by $S \overrightarrow{l_1} R$ and $R \overrightarrow{l_2} S$ with two new tags, tg and ack, we can construct an extension P' of P such that the following* ***specification*** *holds:* $\forall e_1, e_2 : E.\texttt{loc}(e_1) = \texttt{loc}(e_2) = S \ \&\ \texttt{kind}(e_1) = \texttt{kind}(e_2) = \texttt{send}(l_1, tg).\ e_1 < e_2 \Rightarrow \exists r : E.\texttt{loc}(r) = S \ \&\ \texttt{kind}(r) = \texttt{rcv}(l_2, ack).e_1 < r < e_2$.

This theorem is true because we know how to add clauses to processes S and R to achieve the specification, which means that the *specification is constructively achievable*. We can prove the theorem constructively and in the process define the extension P' implicitly. Here is how.

Proof: What would be required of P' to meet the specification? Suppose in P' we have $e_1 < e_2$ as described in the theorem. We need to know more than the obvious fact that two send events occurred namely $\langle tg, m_1\rangle, \langle tg, m_2\rangle$ were sent to R. One way to have more information is to remember the first event in the state. Suppose we use a new Boolean state variable of S, called `rdy`, and we require that a send on l_1 with tag tg happens only if `rdy` = `true` and that after the send, `rdy` = `false`. Suppose we also stipulate in a frame condition that *only a receive on* l_2 sets ready to true, then we know that `rdy when` e_1 = `true`, `rdy after` e_1 = `false` and `rdy when` e_2 = `true`. So between e_1 and e_2, some event $3'$ must happen at S that sets `rdy` to `true`. But since only a $\texttt{rcv}(l_2, ack)$ can do so, then e' must be the receive required by the specification.

This argument proves constructively that P' exists, and it is clear that the proof shows how to extend process S namely add these clauses:

a : `if rdy = true then`
`send`$(l_1, \langle tg, m\rangle)$; `rdy := false`
r : `rcv`(l_2, ack) `effect rdy := true`
`only`$[a, r]$ `affect rdy`

QED

We could add a liveness condition that a send will occur by initializing `rdy` to true. If we want a live dialogue we would need to extend R by

`rcv`$(l_1, \langle tg, m\rangle)$ `effect send` (l_2, ack)

but our theorem did not require liveness.