

CS6110/6116 Lecture 39

We will look at a simple distributed computing model based on Nancy Lynch's [IO Automata](#), and then define a natural [event logic](#) used for reasoning about distributed protocols and systems based on these IO Automata (IOA).

Nuprl uses a more abstract notion of process as the distributed computing model in the logic of events -- our [general process model](#).

Outline of Topics

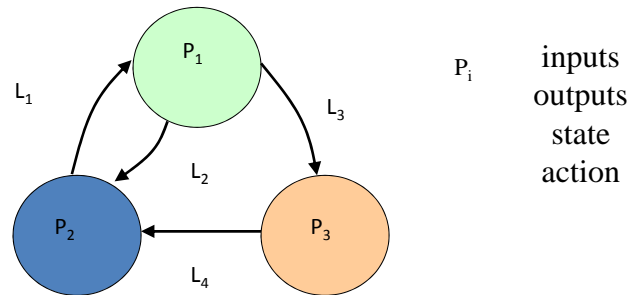
Asynchronous message-passing model of distributed computing

[locations](#) (agents, state machines)
connected by communication [links](#)

[Events occur](#) during computation and are causally ordered *a la* Lamport

Simple example: leader election in a ring

Computations with state: asynchronous, distributed



P_i inputs
 outputs
 state
 action

P_i processes
 L_i communications channels

Computational Rules for Processes

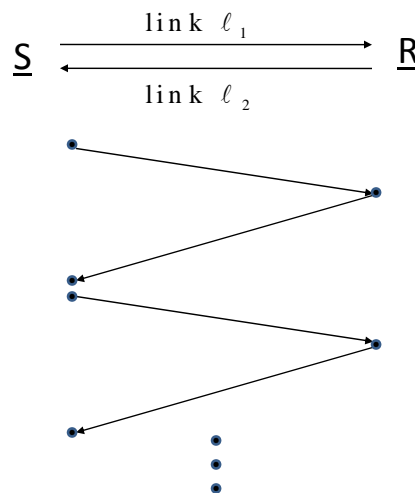
The reduction rules for IOA or the Message Automata used for event logic are not deterministic because other processes change the communication links, and a scheduler picks the actions taken at a location.

Our semantics assumes a **scheduler** for each process and allows for an unbounded number of outcomes in one reduction step of a system.

Message Automata Clauses

- $@i\ x : T$ **initially** $= v$
- $@i$ **effect** $k(v : t)$ **on** x
 $x := f\ state\ v$
- $@i$ **precondition** $a(v:t)$ **is** $P\ state\ v$
- $k(v : T)$ **sends on link l**
 $[tg_1, f_1\ state\ v; \dots; tg_n, f_n\ state\ v]$
- $@i$ **only** $[k_1, \dots, k_n]$ **sends on link l with tag tg**
- $A \oplus B$, where A, B are message automata

Message Sequence Diagrams



Event Structures – the computation model

Our semantics is abstracted from the standard textbook **asynchronous message passing model**, e.g. Attiya & Welch 2004. At each process (location) i there is a **state** $s(i)$, **message** queues (in coming, outgoing) $m(i)$, and an **action** $a(i)$. There is a schedule which determines which action is taken if any.

Event Structures – the events

The events e at a location i are transitions

$$s(i), m(i), a(i) \rightarrow s'(i), m'(i), a'(i)$$

at which an action takes place. At a location, the events are sequential, linearly ordered s_0, s_1, s_2, \dots and unbounded.

Executions of Distributed Systems

Executions of distributed systems are event structures in a natural way. The **execution** of a process (Message Automaton) is the set of all event structures consistent with it.

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.

9

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.

10

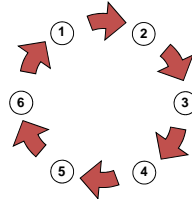
Specification for Leader Election in a Ring

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 $n(i) = \text{dst}(\text{out}(i))$, 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 $i \neq p(j) \Rightarrow d(i, p(j)) = d(i, j) - 1$.

11

Specification, continued

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

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

12

Specification, continued

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

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

13

Decomposing the Leader Election Task

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

1. $\exists e. \text{kind}(e)=\text{rcv}(\text{out}(i), \langle \text{vote}, \text{uid}(i) \rangle)$
2. $\forall e'. \text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, u \rangle) \Rightarrow (u > \text{uid}(i) \Rightarrow \exists e'. \text{kind}(e')=\text{rcv}(\text{out}(i), \langle \text{vote}, u \rangle))$
3. $\forall e'. [(\text{kind}(e')=\text{rcv}(\text{out}(i), \langle \text{vote}, \text{uid}(i) \rangle)) \vee \exists e. (\text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, u \rangle) \ \& \ (e < e' \ \& \ u > \text{uid}(i)))]$
4. $\forall e@i. \text{kind}(e)=\text{rcv}(\text{in}(i), \text{uid}(i)). \exists e'@i. \text{kind}(e')=\text{leader}$
5. $\forall e@i. \text{kind}(e)=\text{leader}. \exists e@i. \text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle)$

14

Realizing Leader Election

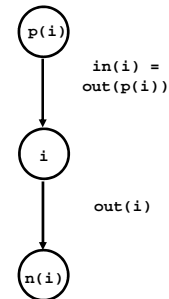
Theorem $\forall R: \text{List}(\text{Loc}) . \text{Ring}(R)$
 $\exists D: \text{Dsys}(R) . \text{Feasible}(D) .$
 $\forall es: \text{Consistent}(D, es) . (\text{LE}(R, es) \Rightarrow \text{Leader}(R, es))$

Proof: Let $m = \max \{ \text{uid}(i) \mid i \in R \}$, then $\text{ldr} = \text{uid}^{-1}(m)$.
 We prove that $\text{ldr} = \text{uid}^{-1}(m)$ using three simple lemmas.

15

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 $u > \text{uid}(i)$ it sends it on.
3. Every rcv is either vote of predecessor $\text{rcv}_{\text{in}(i)}$ for itself or a vote larger than process id before.
4. If a process gets a vote for itself, it declares itself ldr.
5. If a processor declares ldr it got a vote for itself.



16

Lemmas

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

By **induction on distance of i to ldr** .

Lemma 2. $\forall i, j : R. \forall e @ i. \text{kind}(e) = \text{rcv}(\text{in}(i), \langle \text{vote}, j \rangle) .$
 $(j = \text{ldr} \vee d(\text{ldr}, j) < d(\text{ldr}, i))$

By **induction on causal order of rcv events**.

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

If $\text{kind}(e') = \text{leader}$, then by property 5, $\exists v @ i. \text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle) .$

Hence, by Lemma 2 $i = \text{ldr} \vee (d(\text{ldr}, i) < d(\text{ldr}, i))$

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

$\exists e. \text{kind}(e) = \text{rcv}(\text{in}(\text{ldr}), \langle \text{vote}, \text{uid}(\text{ldr}) \rangle)$

which follows from Lemma 1.

QED

17

Leader Election Message Automaton

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

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 $x := \text{if } me = v \text{ then } true \text{ else } x$

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*

18

Consensus is a Good Example

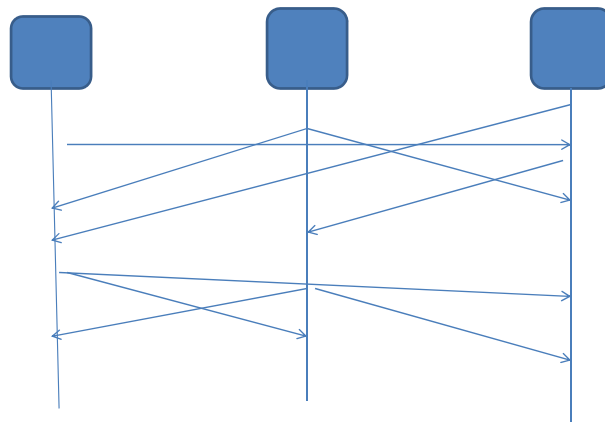
In modern distributed systems, e.g. the Google file system, clouds, etc., reliability against **faults** (crashes, attacks) is achieved **by replication**.



Consensus is used to coordinate write actions to keep the replicas identical. It is a **critical protocol** in modern systems used by IBM, Google, Microsoft, Amazon, EMC, etc.

Requirements of Consensus Task

Use **asynchronous** message passing to decide on a value.



Logical Properties of Consensus

P1: If all inputs are **unanimous** with value v , then any decision must have value v .

All $v:T$. (If All $e:E(\text{Input})$. $\text{Input}(e) = v$ then
All $e:E(\text{Decide})$. $\text{Decide}(e) = v$)

Input and **Decide** are **event classes** that effectively partition the events and assign values to them. The **events** are points in abstract space/time at which “information flows.” More about this just below.

Logical Properties continued

P2: All decided values are input values.

All $e:E(\text{Decide})$. Exists $e':E(\text{Input})$.
 $e' < e$ & $\text{Decide}(e) = \text{Input}(e')$

We can see that P2 will imply P1, so we take P2 as part of the requirements.

Event Classes

If X is an **event class**, then $E(X)$ are the events in that class. Note $E(X)$ **effectively** partitions all events E into $E(X)$ and $E - E(X)$, its complement.

Every event in $E(X)$ has a value of some type T which is denoted **$X(e)$** . In the case of $E(\text{Input})$ the value is the typed input, and for $E(\text{Decide})$ the value is the one decided.

Events

Formally the type E of events is defined relative to the computation model which includes a definition of **processes**.

The events are the **points of space/time** at which information is exchanged. The information at an event e is **$\text{info}(e)$** .

Further Requirements for Consensus

The key **safety property** of consensus is that all decisions agree.

P3: Any two decisions have the same value.
This is called **agreement**.

All e_1, e_2 : $E(\text{Decide})$. $\text{Decide}(e_1) = \text{Decide}(e_2)$.

Specific Approaches to Consensus

Many consensus protocols proceed in **rounds**, **voting on values**, trying to reach agreement. We have synthesized two families of consensus protocols, the **2/3 Protocol** and the **Paxos** Protocol families.

We structure specifications around **events during the voting process**, defining $E(\text{Vote})$ whose values are pairs $\langle n, v \rangle$, a **ballot number**, n , and a **value**, v .

Properties of Voting

Suppose a group G of n processes, P_i , decide by voting. If each P_i collects all n votes into a list L , and applies some **deterministic function $f(L)$** , such as majority value or maximum value, etc., then **consensus is trivial in one step**, and the value is known at each process in the first round – possibly at very different times.

The problem is much harder because of **possible failures**.

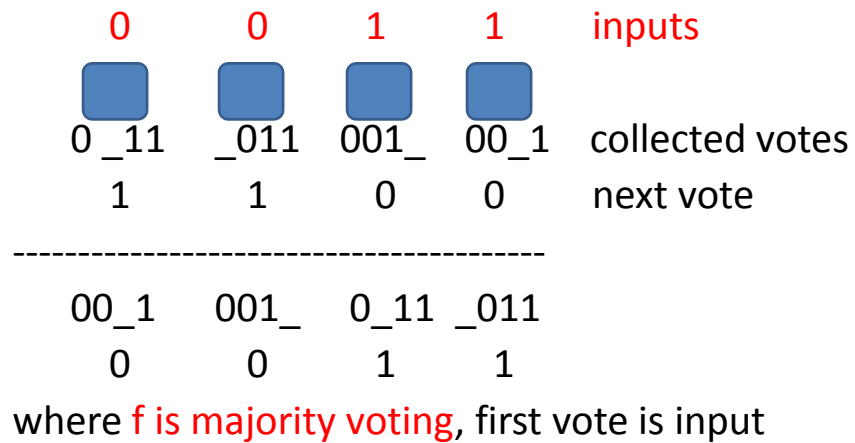
Fault Tolerance

Replication is used to ensure system availability in the presence of **faults**. Suppose that we assume that up to f processes in a group G of n might fail, then how do the processes reach consensus?

The **TwoThirds method** of consensus is to take $n = 3f + 1$ and **collect only $2f + 1$** votes on each round, assuming that f processes might have failed.

Example for $f = 1, n = 4$

Here is a sample of voting in the case $T = \{0,1\}$.



Specifying the 2/3 Method

We can specify the fault tolerant 2/3 method by introducing further event classes.

$E(\text{Vote})$, $E(\text{Collect})$, $E(\text{Decide})$

$E(\text{Vote})$: the initial vote is the $\langle 0, \text{input value} \rangle$,
subsequent votes are $\langle n, f(L) \rangle$

$E(\text{Collect})$: collect $2f+1$ values from G into list L

$E(\text{Decide})$: decide v if all collected values are v

The Hard Bits

The small example shows what can go wrong with $2/3$. It can **waffle forever** between 0 and 1, thus never decide.

Clearly if there is a decide event, the values agree and that unique value is an input.





Can we say anything about eventually deciding, e.g. **liveness**?

Liveness

If f processes eventually fail, then our design will work because if f **have all failed** by round r , then at round $r+1$, all alive processes will see the same $2f+1$ values in the list L , and thus they will all vote for $v' = f(L)$, so in round $r+2$ the values will be unanimous which will trigger a decide event.

Example for $f = 1, n = 4$

Here is a sample of voting in the case $T = \{0,1\}$.

0	0	1	1	inputs
				
0 01_	001_	001_	_011	collected votes
0	0	0	1	next vote

000_	00_1	0_01	_001
0	0	0	0

where f is majority voting, first vote is input, round numbers omitted.

Safety Example

We can see in the $f = 1$ example that once a process P_i receives $2/3$ unanimous values, say 0, it is not possible for another process to overturn the majority decision.

Indeed this is a general property of a $2/3$ majority, the remaining $1/3$ cannot overturn it even if they band together on every vote.

Safety Continued

In the general case when voting is not by majority but using $f(L)$ and the type of values is discrete, we know that **if any process P_i sees unanimous value v in L , then any other process P_j seeing a unanimous value v' will see the same value, i.e. $v = v'$** because the two lists, L_i and L_j at round r must share a value, that is they intersect.

Synthesizing the 2/3 Protocol from a Proof of Design

We can formally prove the safety and liveness conditions from the event logic specification given earlier.

From this **formal proof of design, pf**, we can automatically extract a protocol, first as an abstract process, then by verified compilation, a program in Java or Erlang.

The Synthesized 2/3 Protocol

Begin r :Nat, decided_ i , vote_ i : Bool,
 $r = 0$, decided_ $i = \text{false}$, $v_i = \text{input to } P_i$; vote_ $i = v_i$

Until decided_ i **do**:

1. $r := r + 1$
2. **Broadcast** vote $\langle r, \text{vote}_i \rangle$ to group G
3. **Collect** $2f + 1$ round r votes in list L
4. vote_ $i := \text{majority}(L)$
5. **If** unanimous(L) **then** decided_ $i := \text{true}$

End

Role of the Environment

All distributed computing models must have a component that determines when messages between processes are delivered. We call this the **environment**. It introduces **uncertainty** into the model and determines the **schedule** of events.

A Fundamental Theorem of about the Environment

The **Fischer/Lynch/Paterson** theorem (**FLP85**) about the computing environment says:

it is not possible to guarantee consensus among n processes when one of them might fail.

We have seen the possibility of this with the 2/3 Protocol which could waffle between choosing 0 or 1. The environment can act as an adversary to consensus by managing message delivery.

The Environment as Adversary

In the setting of synthesizing protocols, I have shown that the FLP result can be made constructive (**CFLP**). This means that there is an algorithm, env , which given a potential consensus protocol P and a proof pf that it is **nonblocking** can create message ordering and a computation based on it, $\text{env}(P, \text{pf})$, in which P runs forever, failing to achieve consensus.

Perfect Attacker

The algorithm $\text{env}(P, pf)$ is the perfect “denial of service attacker” against any consensus protocol P that is sensible (won’t block).

Note, $2/3$ will **block** if it waits for n replies or if it refuses to change votes as rounds progress.