Analyzing Synchronous Distributed Algorithms

- Linear Temporal Logic
- Fluents
- Atomic Commitment specification using fluents
- Synchronous Models

Linear Temporal Logic

- **LTL formulas built from:**
  - atomic propositions in $\mathcal{P}$ and standard Boolean operators
  - temporal operators:-

X	next time	
U	strong until	
W	weak until	
F	eventually	<>
G	always	[]

- **Interpreted on infinite words** $w = \langle x_0\, x_1\, x_2 \ldots \rangle$ **over** $2^{\mathcal{P}}$
  - x_i is the set of atomic propositions that hold at time instant i
  - In other words, an interpretation maps to each instant of time a set of propositions that hold at that instant.

Linear Temporal Logic

- $w \models p$ iff $p \in x_0$, for $p \in \mathcal{P}$
- $w \models \neg \varphi$ iff not $w \models \varphi$
- $w \models \varphi \vee \psi$ iff $(w \models \varphi)$ or $(w \models \psi)$
- $w \models \varphi \wedge \psi$ iff $(w \models \varphi)$ and $(w \models \psi)$
- $w \models \mathbf{X}\, \varphi$ iff $w_1 \models \varphi$
- $w \models \varphi\, \mathbf{U}\, \psi$ iff $\exists\, i \geq 0$, such that:
  $w_i \models \psi$ and $\forall\, 0 \leq j < i$, $w_j \models \varphi$

Linear Temporal Logic

Using $\neg, \vee, \wedge, \mathbf{X}, \mathbf{U}$

- $\mathbf{true} \equiv \varphi \vee \neg\varphi$
- $\mathbf{false} \equiv \neg\mathbf{true}$
- $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\mathbf{F}\, \varphi \equiv \mathbf{true}\; \mathbf{U}\, \varphi$ -- eventually
- $\mathbf{G}\, \varphi \equiv \neg\mathbf{F}\neg\varphi$ -- always
- $\varphi\, \mathbf{W}\, \psi \equiv ((\varphi\, \mathbf{U}\, \psi) \vee \mathbf{G}\, \varphi)$ -- weak until

What are the atomic propositions?

Fluents

Fluents (time-varying properties of the world) are true at particular time-points if they have been initiated by an action occurrence at some earlier time-point, and not terminated by another action occurrence in the meantime. Similarly, a fluent is false at a particular time-point if it has been previously terminated and not initiated in the meantime

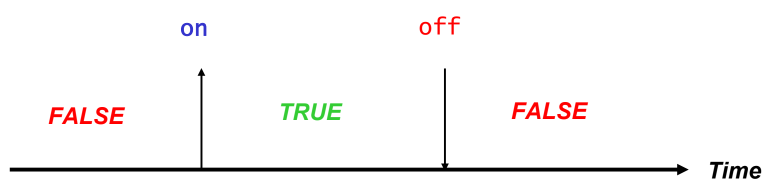
[Sandewall 94]; [Kowalski, Sergot 86]; [Miller, Shanahan 99]

Fluent LTL (FLTL)

- **Set of atomic propositions is set of fluents** Φ
- **We define fluents as follows:** $Fl \equiv \langle I_{Fl}, T_{Fl} \rangle$
  - I_{Fl}, T_{Fl} are sets of initiating and terminating actions accordingly, such that $I_{Fl} \cap T_{Fl} = \varnothing$
- **A fluent *Fl* may initially be true or false at time zero as denoted by the attribute *InitiallyFl***
- **For LTS *M*, an action *a* defines implicitly:**
  - $Fluent(a) \equiv \langle \{a\}, \ \alpha M - \{a\} \rangle$ $\quad$ $Initially_a = false$

Fluent Propositions

Defined in terms of sets actions




```
fluent
  LIGHT = <{on},{power_cut,off}> initially False
```


Atomic Commitment

Alphabet

```
range ID  = 0..N-1
vote[i:ID].yes      //  process i votes yes
vote[i:ID].no       //  process i votes no
decide[i:ID].yes    //  process i decides yes
decide[i:ID].no     //  process i decides no
fail[i:ID]          //  process i fails
```

Fluents – default is initially false

```
fluent VOTE[i:ID][v:{yes,no}] = <vote[i][v],never>
fluent DECIDED[i:ID]          = <decide[i].{yes,no},never>
fluent COMMIT [i:ID]          = <decide[i].yes, never>
fluent ABORT  [i:ID]          = <decide[i].no, never>
```

The declaration `ABORT[i:ID]` above is simply declaring a set of fluents, `ABORT[0]`, `ABORT[1]`, `ABORT[2]` and `ABORT[3]`, `for N = 4.`

FLTL syntax

Unary operators (*unop*):	Binary operators(*binop*):
`[]` always **(G)**	`U` strong until
`<>` eventually **(F)**	`&&` logical AND
`X` next time	`||` logical OR
`!` logical negation	`->` implication
	`<->`equivalence

FLTL formula Φ := `True` | `False` | *prop* | (Φ) | *unop* Φ | Φ *binop* Φ,
where *prop* is a fluent, action or set of actions.

```
exists[i:1..N] Φ(i)      ≡  Φ(1) || …|| Φ(N)
-- short form  FL[1..N] ≡  FL[1] || …|| FL[N]
forall[i:1..N] Φ[i]      ≡  Φ(1) && …&& Φ(N)
```

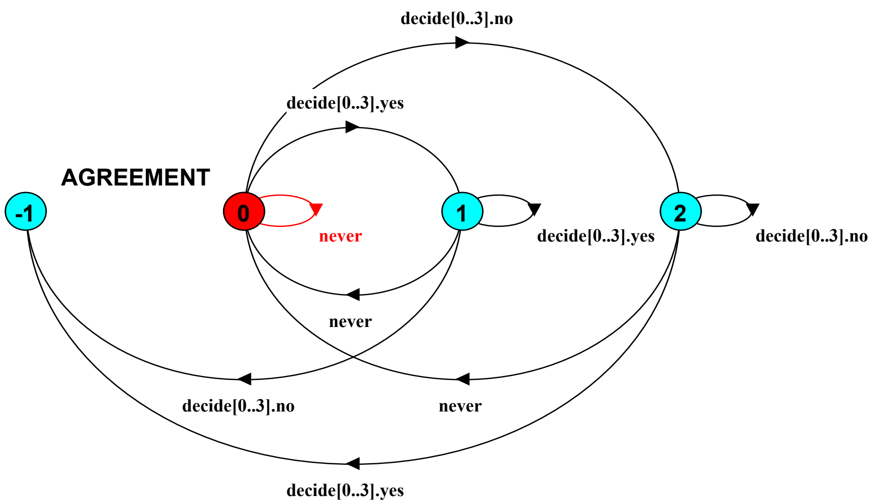

Atomic Commitment Properties - safety

Agreement: No two processes (whether correct or crashed) should decide on different values.

```
assert AGREEMENT =  []!(COMMIT[ID] && ABORT[ID])
```

-- it is always not (never) the case that one of the processes 0..N-1 can have committed and also one of these processes can have aborted i.e. have decided on different values.

AGREEMENT property LTS



Atomic Commitment Properties - safety

Validity - 1:

If *any* process votes *no*, then *no* is the only possible decision value.

```
assert VALID_1 = [](VOTE[ID]['no] -> !COMMIT[ID])
```

The reading of this formula is that it is always the case that if one of the processes 0..N..1 has voted no then it can not be the case that one of these processes has committed i.e. decided other than no.

Atomic Commitment Properties - safety

Validity – 2:
If *all* processes vote *yes*, and there are no failures, then *yes* is the only possible decision value.

```
assert VALID_2 = [](forall[i:ID]
                    (VOTE[i]['yes] && !CRASHED[i])
                     -> !ABORT[ID])
```

This reads that it is always the case that if all processes vote *yes* and are not crashed (i.e. correct) then it cannot be the case that one of the processes is aborted (i.e. decides other than *yes*). We use not aborted rather than directly using the committed fluent since commitment is not true initially and may never be true due to failure.

Atomic Commitment Properties - liveness

Strong Termination:

All correct processes eventually decide.

```
assert
STRONGTERM = <>(forall[i:ID]
                  (!CRASHED[i]-> DECIDED[i])
               )
```

This reads: it is eventually the case that if a process has not crashed then it will decide.

Atomic Commitment Properties - liveness

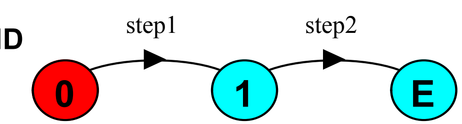
- **Weak Termination**: If there are no failures, then all processes eventually decide.

```
assert
WEAKTERM = ([]forall[i:ID] !CRASHED[i]
               -> <>forall[i:ID] DECIDED[i])
```

This reads: if its always the case that all processes do not crash then eventually all processes reach the decided state.

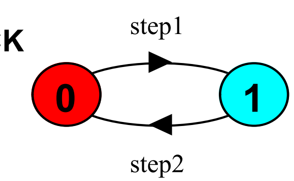
Synchronous model

ROUND




```
ROUND = (step1 -> step2 -> END).
```

CLOCK




```
CLOCK = ROUND;CLOCK+{never}.
```

Network

```
CHAN(From=0, To =1)
   =(chan[From][To].send[m:Msg] -> CHAN[m]
    | step1 -> CHAN
    | step2 -> CHAN['null]
    ),
CHAN[m:Msg]
   =(chan[From][To].recv[m] -> CHAN
    | step1 -> CHAN
    | step2 -> CHAN[m]
    ).

||NETWORK = (forall[i:0..N-1][j:0..N-1]
               if (i!=j) then CHAN(i,j)
             ||CLOCK
             ).
```


SEND_ALL

```
SEND_ALL(From=0,M='null)
   = if ((N-1)>From)
     then TX[N-1-From]
     else (step2->END),
TX[n:0..N-1-From]
   = (when (n>0)
        chan[From][From+1..N-1].send[M] -> TX[n-1]
     |when (n>0)
        fail[From] -> ENDED
     |when (n==0)
        step2 -> END
     ),
ENDED
   = ({step1,step2}->ENDED).
```

SEND_ALL sends a message from a process numbered Id to all processes numbered Id+1..N-1, thus for Id=1 and N=4, the process sends a message to the processes numbered 2 and 3.

Two-phase commit - participant

```
const    N        = 4
set      Msg      = {yes,no, null}

DECIDE(Id=0, D='null)
 = if (D=='yes || D=='no) then (decide[Id][D]->END) else END.

PARTICIPANT(Id=1) = ROUND1,
ROUND1
  = (vote[Id][v:{yes,no}]->step1->SEND[v]),
SEND[v:Msg]
  = (chan[Id][0].send[v] -> step2 ->
       if (v=='no) then DECIDE(Id,v);ENDED else ROUND;ROUND2
    |fail[Id] -> ENDED),
ROUND2
  = (chan[0][Id].recv[m:Msg] -> DECIDE(Id,m);ENDED
    |fail[Id] -> ENDED),
ENDED
  = ({step1,step2}->ENDED)
    +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg]}.
```


Two-phase commit - coordinator

```
COORDINATOR(Id=0)
  = (vote[Id][v:{yes,no}]->ROUND;ROUND1[v]),
ROUND1[v:{yes,no}]
  = (chan[Id+1..N-1][Id].recv[m:Msg]
     -> if (v =='no) then ROUND1['no]
        else if (m == 'no || m == 'null) then ROUND1['no]
        else if (m == 'yes && v == 'yes) then ROUND1['yes]
    |step1 -> ROUND2[v]
    ),
ROUND2[v:{yes,no}]
   = DECIDE(Id,v);SEND_ALL(Id,v);ENDED,
ENDED
   = ({step1,step2}->ENDED)
       +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg]}.
```

Two-phase commit - system

```
//constrain number of failures
FCONSTRAINT(F=0) = FAIL[0],
FAIL[f:0..F]     = (when (f<F) fail[0..N-1] -> FAIL[f+1]
                   )+{fail[0..N-1]}.

||SYS = (  COORDINATOR(0)
        || forall[i:1..N-1] PARTICIPANT(i)
        || NETWORK
        || FCONSTRAINT(2)
        )>>{step1,step2}.
```

Making step1 & step 2 low priority forces all communication to occur within rounds.

Analysis

- Check to see if all properties hold
  - Strong Termination?
- For properties that do hold, example or *witness* executions can be obtained by checking the negation of a property:

```
assert WITNESS_AGREEMENT = !AGREEMENT
```

- Check the same properties for three-phase commit.