Lecture 3

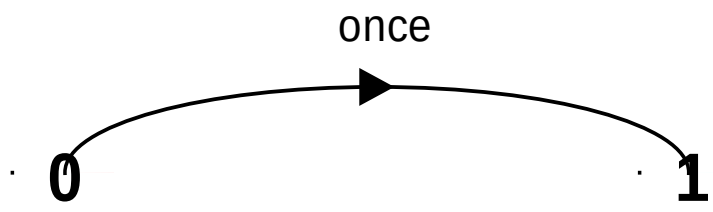
- Administration
  - Piazza
  - Project Update
  - Assignment Update

- Describing distributed systems

FSP - action prefix

If **x** is an action and **P** a process then **(x-> P)** describes a process that initially engages in the action **x** and then behaves exactly as described by **P**.

```
ONESHOT = (once -> STOP).
```



ONESHOT state machine

(terminating process)

Convention: actions begin with lowercase letters

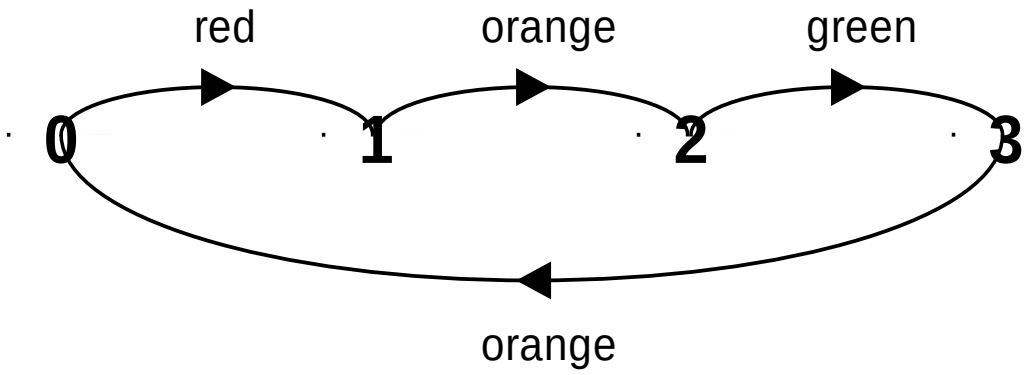
PROCESSES begin with uppercase letters

FSP - action prefix

FSP model of a traffic light :

```
TRAFFICLIGHT = (red->orange->green->orange
                        -> TRAFFICLIGHT).
```

LTS generated using *LTSA*:



Trace:

red→orange→green→orange→red→orange→green …

FSP - choice

If **x** and **y** are actions then **(x-> P | y-> Q)** describes a process which initially engages in either of the actions **x** or **y**. After the first action has occurred, the subsequent behavior is described by **P** if the first action was **x** and **Q** if the first action was **y**.

Who or what makes the choice?

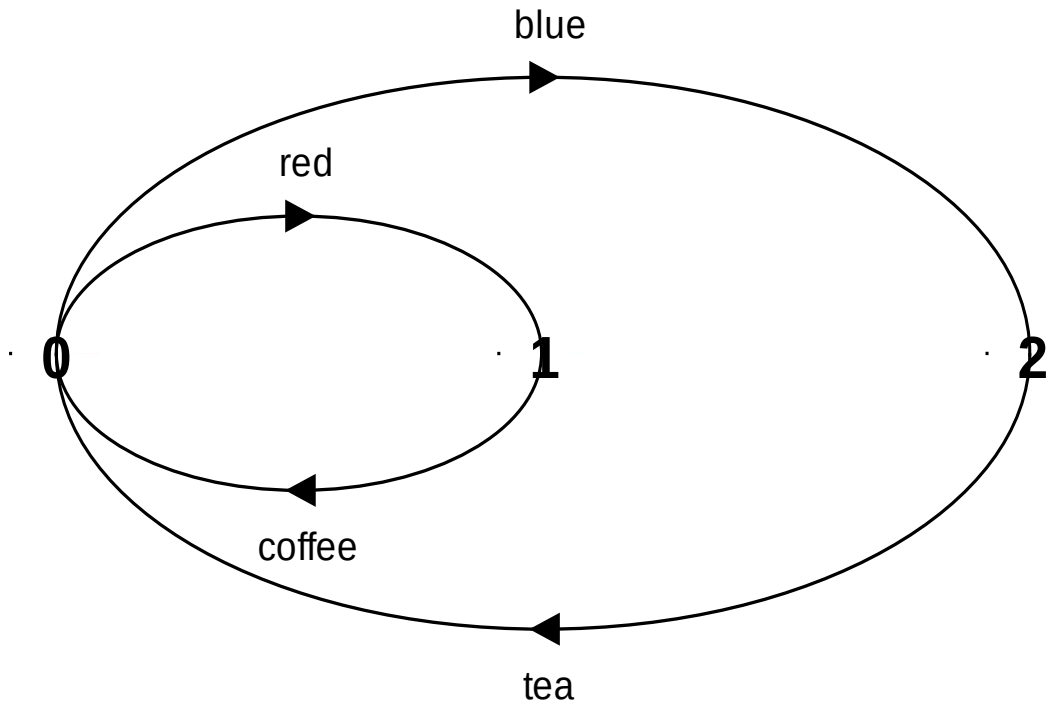
Is there a difference between input and output actions?

FSP - choice

FSP model of a drinks machine :

```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

LTS generated using *LTSA*:



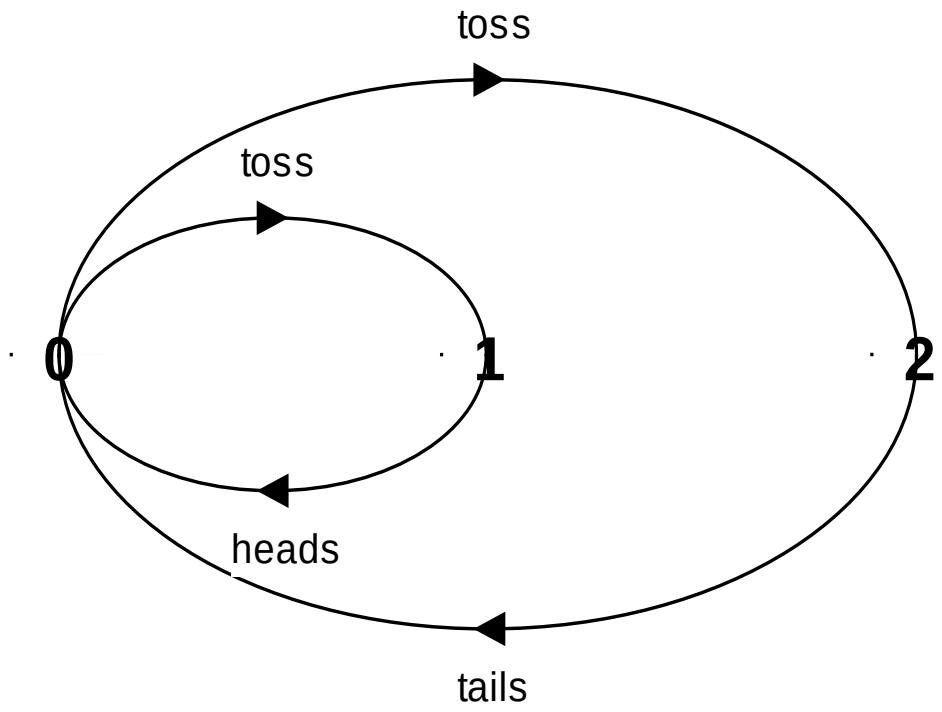
Possible traces?

Non-deterministic choice

Process (x-> P | x -> Q) describes a process which engages in **x** and then behaves as either P or Q.

```
COIN = (toss->HEADS|toss->TAILS),
HEADS= (heads->COIN),
TAILS= (tails->COIN).
```

Tossing a coin.



Possible traces?

FSP – Finite State Processes

If **x** is an action and **P** a process then **(x-> P)** describes a process that initially engages in the action **x** and then behaves exactly as described by **P**.

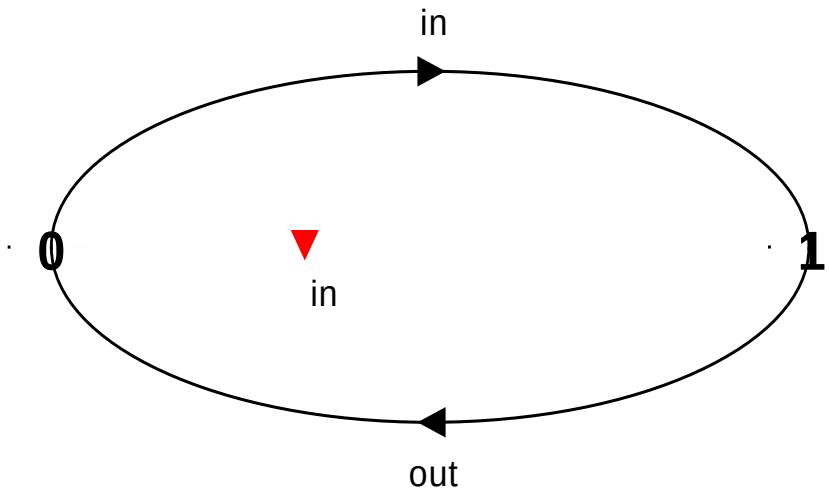
If **x** and **y** are actions then **(x-> P | y-> Q)** describes a process which initially engages in either of the actions **x** or **y**. After the first action has occurred, the subsequent behavior is described by **P** if the first action was **x** and **Q** if the first action was **y**.

Process (x-> P | x -> Q) describes a process which engages in **x** and then behaves as either P or Q.

Modeling failure

How do we model an unreliable communication channel which accepts **in** actions and if a failure occurs produces no output, otherwise performs an **out** action?

Use non-determinism...




```
CHAN = (in->CHAN
       |in->out->CHAN
       ).
```

FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

equivalent to

```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

indexed actions generate labels of the form *action.index*

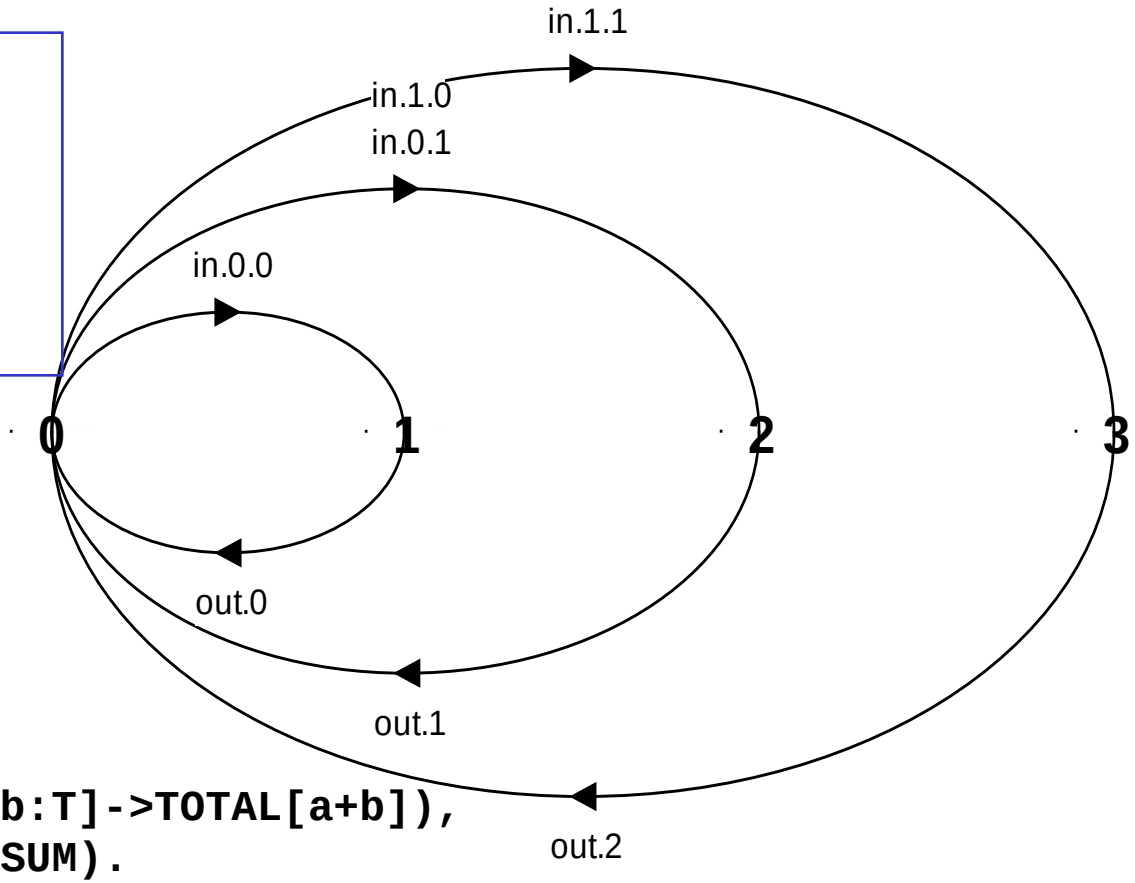
or using a process parameter with default value:

```
BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
```

FSP - indexed processes and actions

Local indexed process definitions are equivalent to process definitions for each index value

index expressions to model calculation:

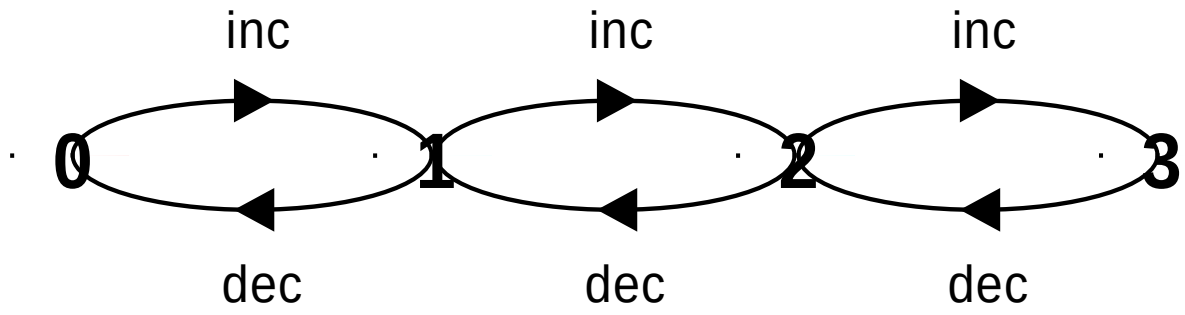



```
const N = 1
range T = 0..N
range R = 0..2*N

SUM         = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).
```

FSP - guarded actions

The choice **(when B x -> P | y -> Q)** means that when the guard **B** is true then the actions **x** and **y** are both eligible to be chosen, otherwise if **B** is false then the action **x** cannot be chosen.

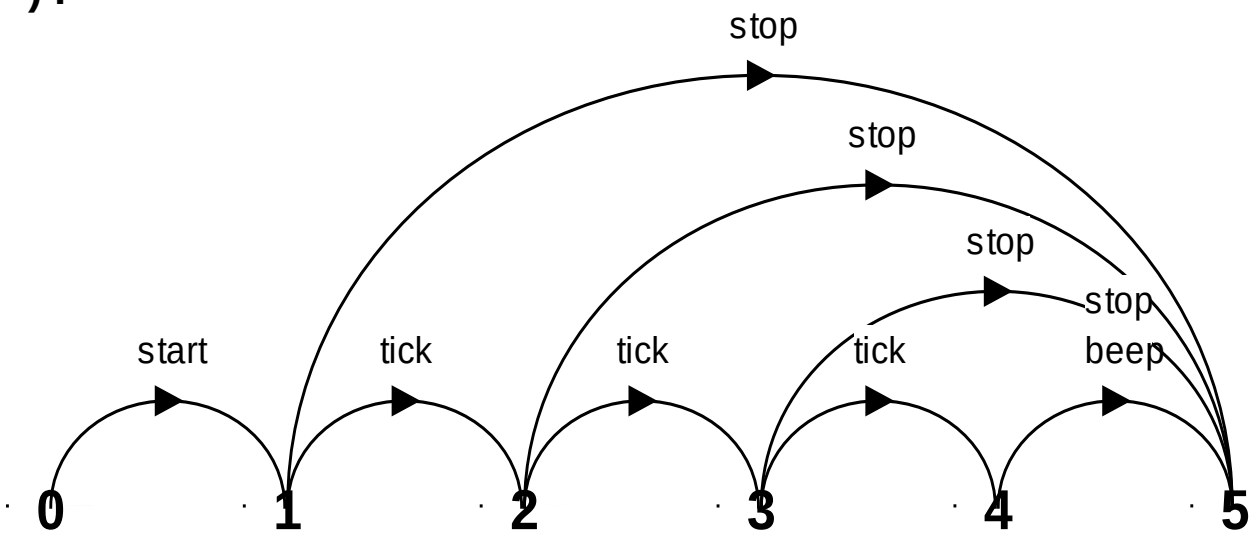
```
COUNT (N=3)   = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```



FSP - guarded actions

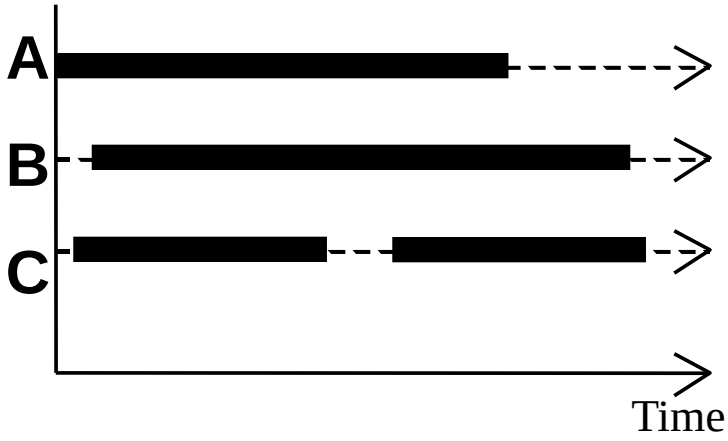
A countdown timer which beeps after N ticks, or can be stopped.

```
COUNTDOWN (N=3)   = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
            (when(i>0) tick->COUNTDOWN[i-1]
            |when(i==0)beep->STOP
            |stop->STOP
            ).
```

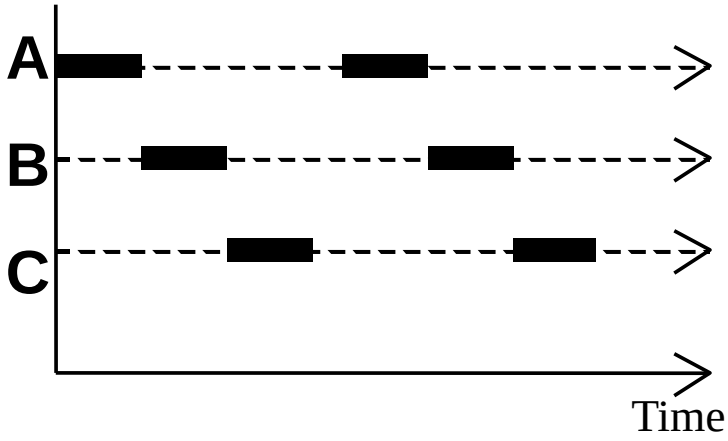


Parallelism vs. Concurrency

- **Parallelism**



- **Concurrency**



Both **concurrency** and **parallelism** require controlled access to *shared resources*.

We use the terms parallel and concurrent interchangeably (and generally do not distinguish between real and pseudo-concurrent execution).

Also, creating software independent of the physical setup, makes us capable of deploying it on *any* platform!

Parallel composition

If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

```
ITCH  = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

Disjoint alphabets

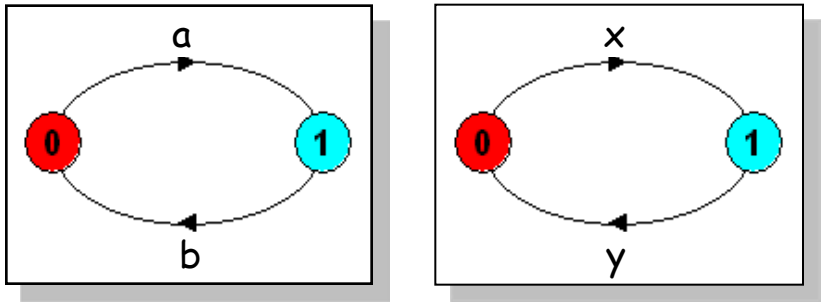
think→talk→scratch
think→scratch→talk
scratch→think→talk

Possible traces as a result of action interleaving.

How are we modeling concurrency

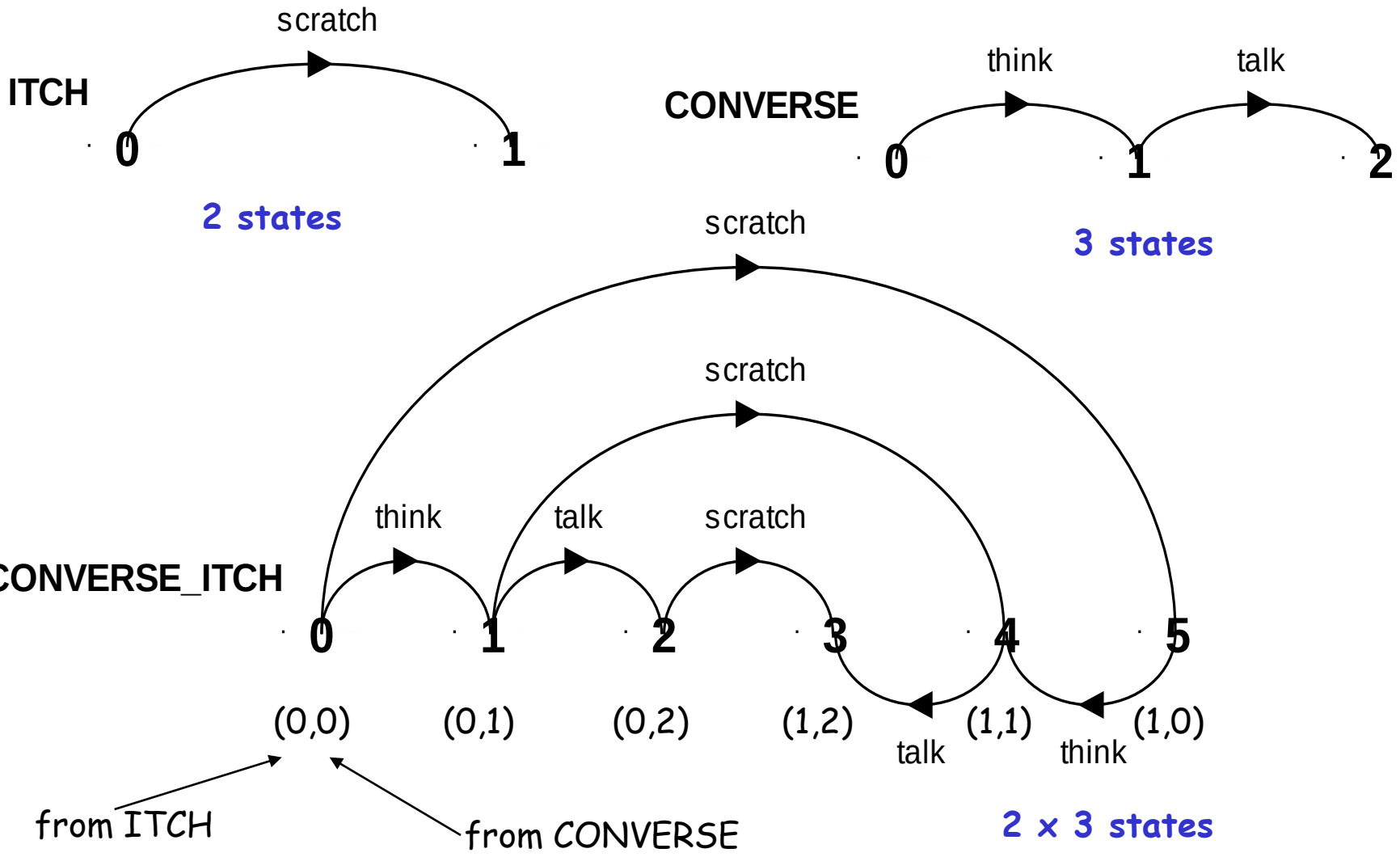
- How do we model concurrency?
  - arbitrary relative order of actions from different processes (**interleaving** but preservation of each process order)
- Independent from process execution speed?
  - arbitrary speed

    (we abstract away time)



- What is the result?
  - independent of architecture, processor speed, scheduling policies, …(**asynchronous** model of execution)
  - -: we can say nothing of real-time properties

parallel composition



parallel composition - algebraic laws

```
Commutative:   (P||Q) = (Q||P)
Associative:   (P||(Q||R))    = ((P||Q)||R)
                              = (P||Q||R).
```

Clock radio example:

```
CLOCK = (tick->CLOCK).
RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).
```

LTS? Traces? Number of states?

modeling interaction - shared actions

```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```

MAKE1 synchronizes with **USE1** when **ready**.

LTS? Traces? Number of states?

Non-disjoint action alphabets

- Shared Actions:

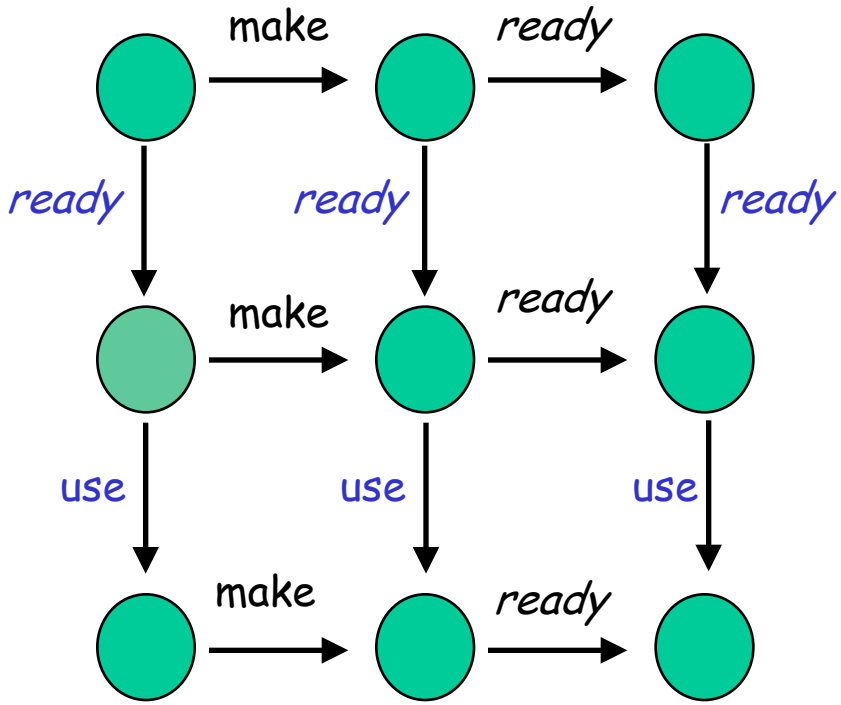
If processes in a composition have actions in common, these actions are said to be *shared*. Shared actions are the way that process interaction is modelled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

modeling interaction - example

```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```

3 states

3 states



3 x 3 states?

No...!

make, ready, ready, use

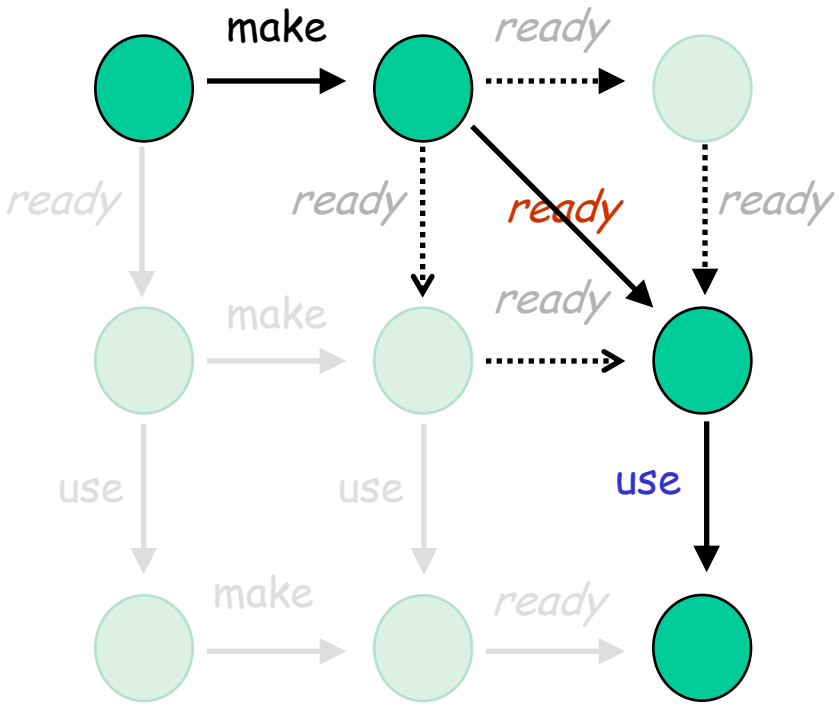
ready, make, ready, use

modeling interaction - example

```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```

3 states

3 states



Must be in a state where both or none of the machines execute ready

4 states!

Interaction may constrain the overall behaviour !

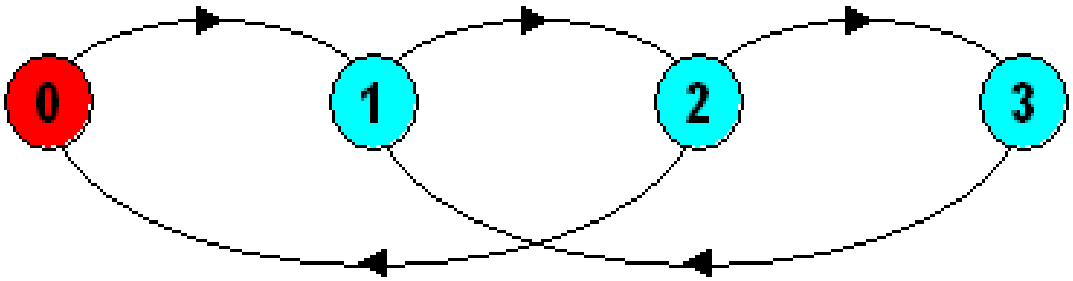
Example

```
P  =  (x -> y -> P).
Q  =  (y -> x -> Q).

||R = (P || Q).
```

2 states

2 states

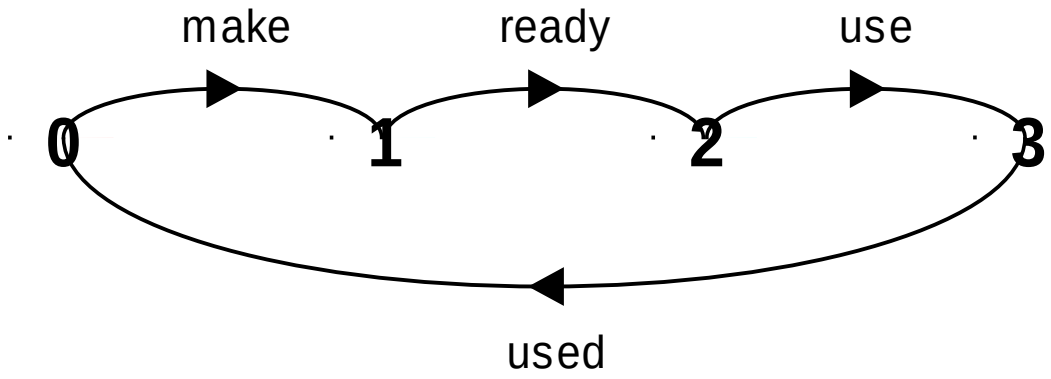
LTS? Traces? Number of states?



modeling interaction - handshake

A *handshake* is an action acknowledged by another:

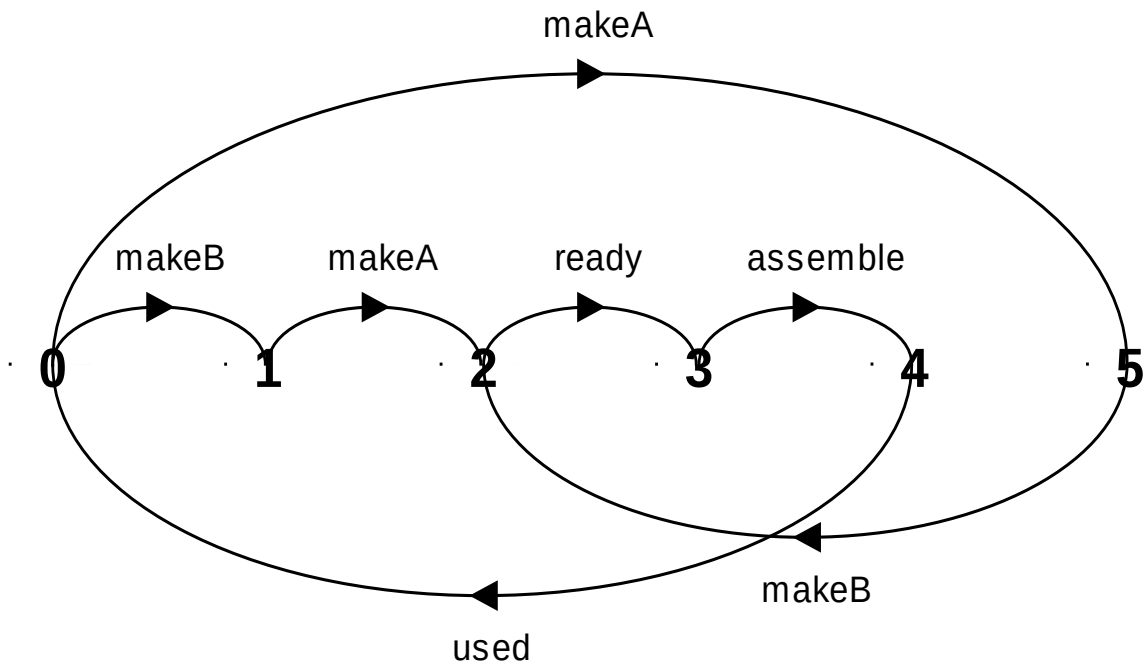
```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2  = (ready->use->used->USERv2).

||MAKER_USERv2 = (MAKERv2 || USERv2).
```



interaction - multiple processes

Multi-party synchronization:

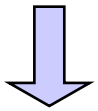
```
MAKE_A   = (makeA->ready->used->MAKE_A).
MAKE_B   = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```



composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

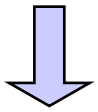
```
||MAKERS   = (MAKE_A || MAKE_B).
||FACTORY = (MAKERS || ASSEMBLE).
```



substitution of
def'n of MAKERS

```
||FACTORY = ((MAKE_A || MAKE_B)|| ASSEMBLE).
```

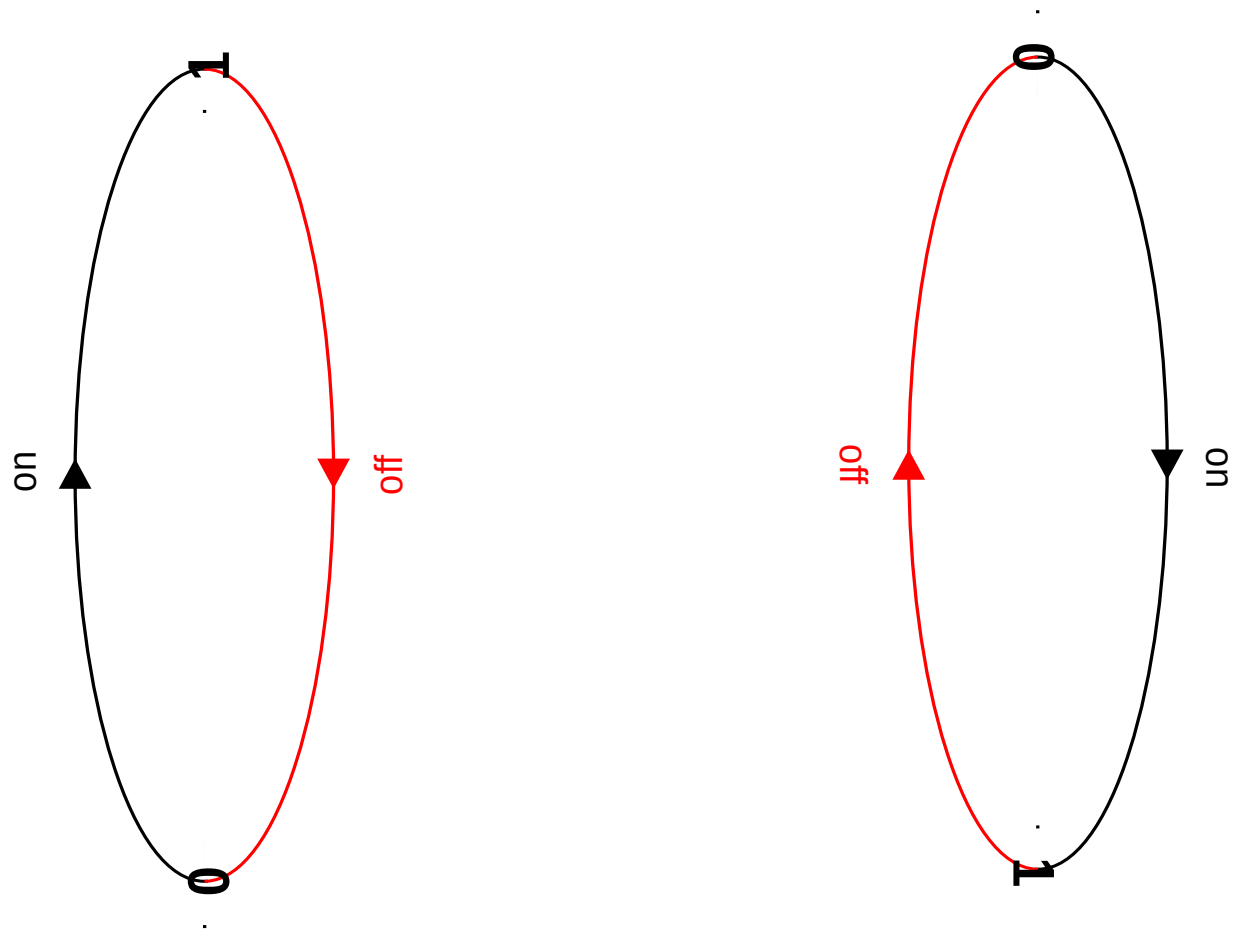
Further simplification?



associativity!

```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

Alternative – TRUE concurrency



Alternative – TRUE concurrency

Petri-Nets