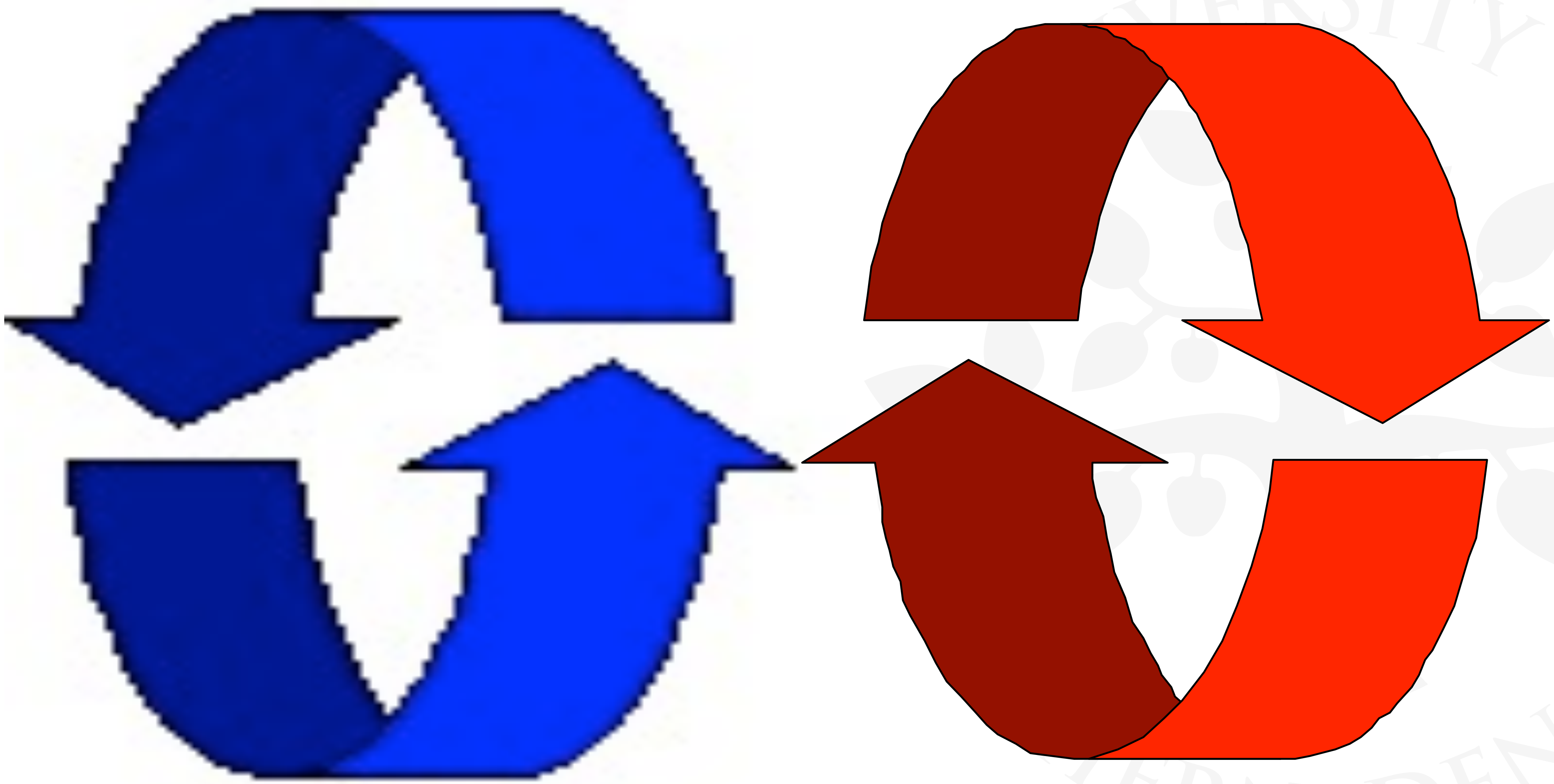
Chapter 3 Concurrent Execution

UNIVERSITY OF SOUTHERN DENMARK



Repetition (Concepts, Models, and Practice)

Repetition (Concepts, Models, and Practice)

- Concepts:

Repetition (Concepts, Models, and Practice)

University of Southern Denmark

- **Concepts**:
  - We adopt a **model-based approach** for the design and construction of concurrent programs

Repetition (Concepts, Models, and Practice)

- **Concepts:**
  - We adopt a **model-based approach** for the design and construction of concurrent programs
    - Safe model => safe program

Repetition (Concepts, Models, and Practice)

- **Concepts:**
  - We adopt a **model-based approach** for the design and construction of concurrent programs
    - Safe model => safe program
- **Models:**

Repetition (Concepts, Models, and Practice)

- **Concepts**:
  - We adopt a **model-based approach** for the design and construction of concurrent programs
    - Safe model => safe program
- **Models**:
  - We use finite state models to represent concurrent behaviour (**Finite State Processes** and **Labelled Transition Systems**)

Repetition (Concepts, Models, and Practice)

University of Southern Denmark

- **Concepts:**
  - We adopt a **model-based approach** for the design and construction of concurrent programs
    - Safe model => safe program
- **Models:**
  - We use finite state models to represent concurrent behaviour (**Finite State Processes** and **Labelled Transition Systems**)
- **Practice:**

Repetition (Concepts, Models, and Practice)

- **Concepts**:
  - We adopt a **model-based approach** for the design and construction of concurrent programs
    - Safe model => safe program
- **Models**:
  - We use finite state models to represent concurrent behaviour (**Finite State Processes** and **Labelled Transition Systems**)
- **Practice**:
  - We use **Java** for constructing concurrent programs

Repetition (Models; LTS, FSP)

Repetition (Models; LTS, FSP)

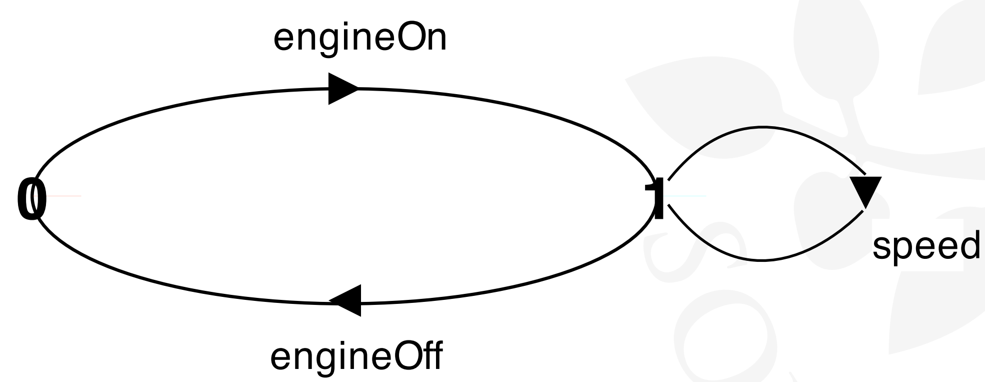
Model = simplified representation of the real world

Repetition (Models; LTS, FSP)

Model = simplified representation of the real world

- Based on **Labelled Transition Systems** (LTS):

Focuses on concurrency aspects (of the program)
- everything else abstracted away



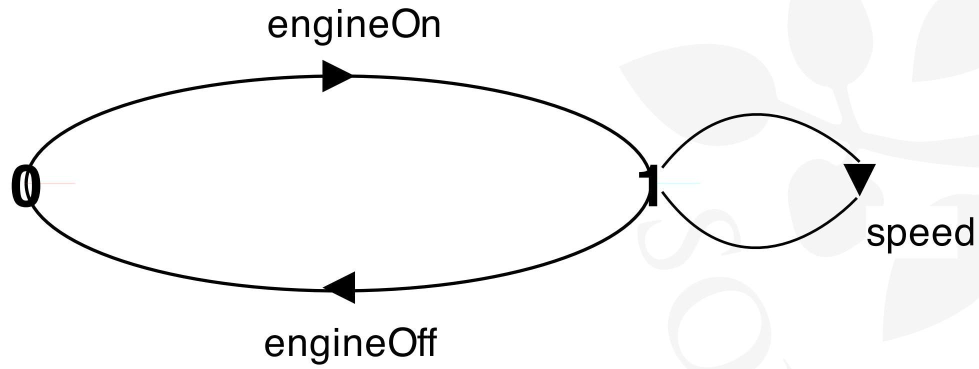
Repetition (Models; LTS, FSP)

UNIVERSITY OF SOUTHERN DENMARK

Model = simplified representation of the real world

- Based on **Labelled Transition Systems** (LTS):

Focuses on concurrency aspects (of the program)
- everything else abstracted away

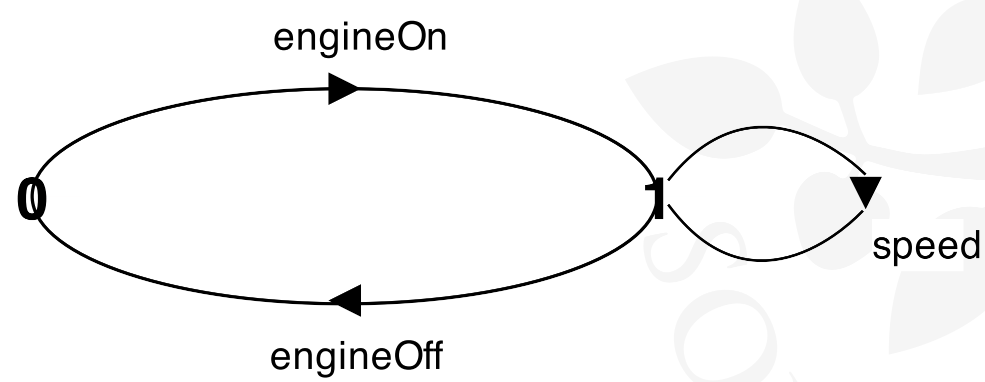


Repetition (Models; LTS, FSP)

Model = simplified representation of the real world

- Based on **Labelled Transition Systems** (LTS):

Focuses on concurrency aspects (of the program)
- everything else abstracted away



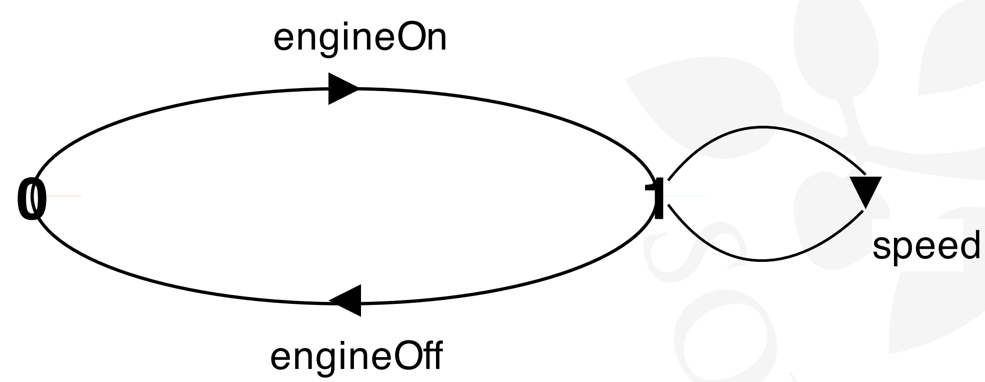
- Described textually as **Finite State Processes** (FSP):

Repetition (Models; LTS, FSP)

Model = simplified representation of the real world

- Based on **Labelled Transition Systems** (LTS):

Focuses on concurrency aspects (of the program)
- everything else abstracted away



- Described textually as **Finite State Processes**

(FSP):

```
EngineOff = (engineOn  -> EngineOn),
EngineOn  = (engineOff -> EngineOff
            |speed     -> EngineOn).
```

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

```
P        :      STOP                    // termination
```

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

```
P    :    STOP           // termination
     :    (x -> P)       // action prefix
```

Repetition (Finite State Processes; FSP)

University of Southern Denmark

Finite State Processes (FSP):

```
P   :   STOP                  // termination
    :   (x -> P)              // action prefix
    :   (when (...) x -> P)   // guard
```

Repetition (Finite State Processes; FSP)

UNIVERSITY OF SOUTHERN DENMARK

Finite State Processes (FSP):

P	:	STOP	// termination
	:	(x -> P)	// action prefix
	:	(when (...) x -> P)	// guard
	:	P | P'	// choice

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

P	:	STOP	// termination
	:	(x -> P)	// action prefix
	:	(when (…) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ … }	// alphabet extension

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

P	:	STOP	// termination
	:	(x -> P)	// action prefix
	:	(when (…) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ … }	// alphabet extension
	:	X	// process variable

Repetition (Finite State Processes; FSP)

University of Southern Denmark

Finite State Processes (FSP):

P	:	STOP	// termination
	:	(x -> P)	// action prefix
	:	(when (…) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ … }	// alphabet extension
	:	X	// process variable

- action indexing x[i:1..N] -> P or x[i] -> P

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

P	:	<u>STOP</u>	// termination
	:	(x -> P)	// action prefix
	:	(<u>when</u> (…) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ … }	// alphabet extension
	:	X	// process variable

- action indexing x[i:1..N] -> P or x[i] -> P
- process parameters P(N=3) = …

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

P	:	<u>STOP</u>	// termination
	:	(x -> P)	// action prefix
	:	(<u>when</u> (…) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ … }	// alphabet extension
	:	X	// process variable

- action indexing x[i:1..N] -> P or x[i] -> P
- process parameters P(N=3) = …
- constant definitions <u>const</u> N = 3

Repetition (Finite State Processes; FSP)

University of Southern Denmark

Finite State Processes (FSP):

```
P   :   STOP                  // termination
    :   (x -> P)              // action prefix
    :   (when (…) x -> P)     // guard
    :   P | P'                // choice
    :   P +{ … }              // alphabet extension
    :   X                     // process variable
```

- action indexing x[i:1..N] -> P or x[i] -> P
- process parameters P(N=3) = …
- constant definitions <u>const</u> N = 3
- range definitions <u>range</u> R = 0..N

Repetition (Finite State Processes; FSP)

Finite State Processes (FSP):

P	:	STOP	// termination
	:	(x -> P)	// action prefix
	:	(when (...) x -> P)	// guard
	:	P | P'	// choice
	:	P +{ ... }	// alphabet extension
	:	X	// process variable

- action indexing — x[i:1..N] -> P or x[i] -> P
- process parameters — P(N=3) = ...
- constant definitions — const N = 3
- range definitions — range R = 0..N

Which constructions do not add expressive power?
(and are thus only "syntactic sugar").

Repetition (Java Threads)

UNIVERSITY OF SOUTHERN DENMARK

Subclassing `java.lang.Thread:`

Implementing `java.lang.Runnable:`

Repetition (Java Threads)

Subclassing `java.lang.Thread`:

```
class MyThread extends Thread {
    public void run() {
        // ...
    }
}
```

```
Thread t = new MyThread();
t.start();
// ...
```

Implementing `java.lang.Runnable`:

Repetition (Java Threads)

Subclassing `java.lang.Thread`:

```
class MyThread extends Thread {
    public void run() {
        // ...
    }
}
```

```
Thread t = new MyThread();
t.start();
// ...
```

Implementing `java.lang.Runnable`:

```
class MyRun implements Runnable {
    public void run() {
        // ...
    }
}
```

```
Thread t = new Thread(new MyRun());
t.start();
// ...
```

Chapter 3: Concurrent Execution

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution and interleaving

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution and interleaving

process interaction

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution and interleaving
process interaction

Models: **parallel composition** of asynchronous processes
interleaving

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution
and interleaving
process interaction

Models: parallel composition of asynchronous processes
interleaving
interaction - shared actions
process labelling, and action relabelling and hiding

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution and interleaving
process interaction

Models: **parallel composition** of asynchronous processes interleaving
interaction - shared actions
process labelling, and action relabelling and hiding
structure diagrams

Chapter 3: Concurrent Execution

Concepts: processes - concurrent execution and interleaving
process interaction

Models: **parallel composition** of asynchronous processes interleaving
interaction - shared actions
process labelling, and action relabelling and hiding
structure diagrams

Practice: Multithreaded Java programs

Definition: Parallelism

Definition: Parallelism

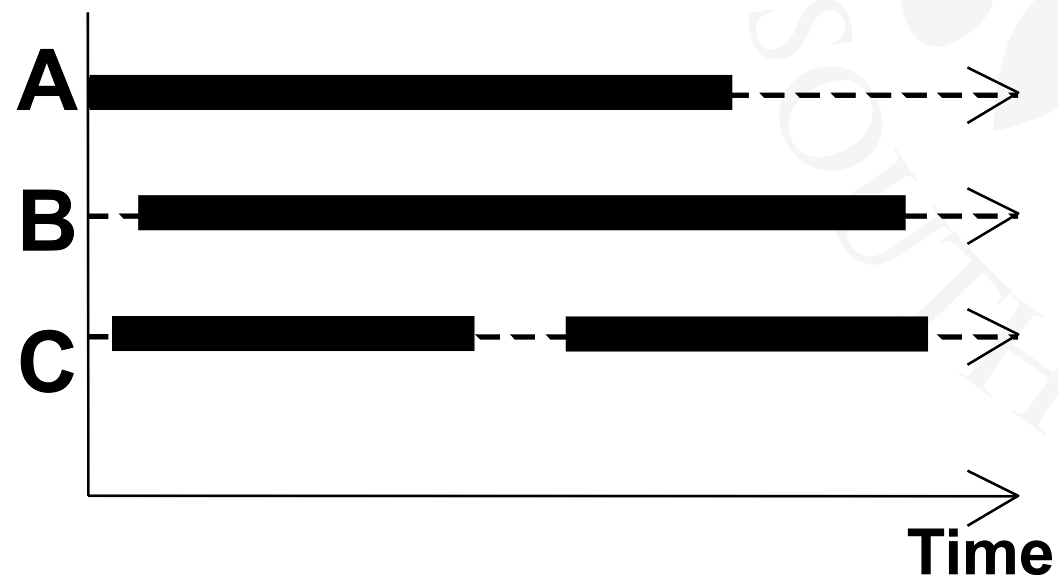
- **Parallelism** (aka. Real/True Concurrent Execution)

Definition: Parallelism

- **Parallelism** (aka. Real/True Concurrent Execution)
  - **Physically** simultaneous processing

Definition: Parallelism

- **Parallelism** (aka. Real/True Concurrent Execution)
  - **Physically** simultaneous processing
    - Involves multiple processing elements (PEs) and/or independent device operations



Definition: Concurrency

Definition: Concurrency

UNIVERSITY OF SOUTHERN DENMARK

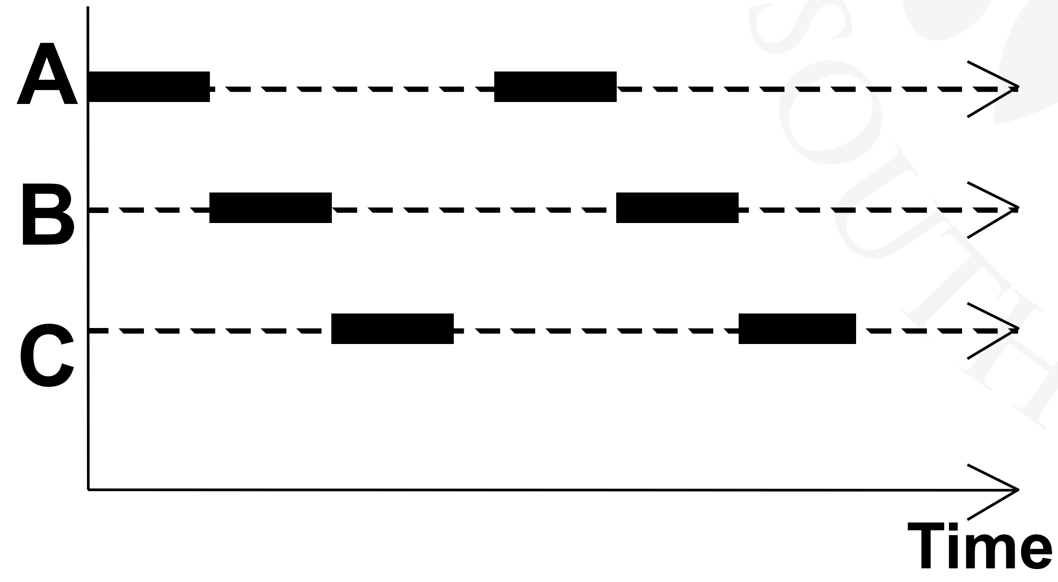
- **Concurrency** (aka. Pseudo-Concurrent Execution)
  - **Logically** simultaneous processing
    - Does not imply multiple processing elements (PEs)

Definition: Concurrency

- **Concurrency** (aka. Pseudo-Concurrent Execution)
  - **Logically** simultaneous processing
    - Does not imply multiple processing elements (PEs)
    - Requires interleaved execution on a single PE

Definition: Concurrency

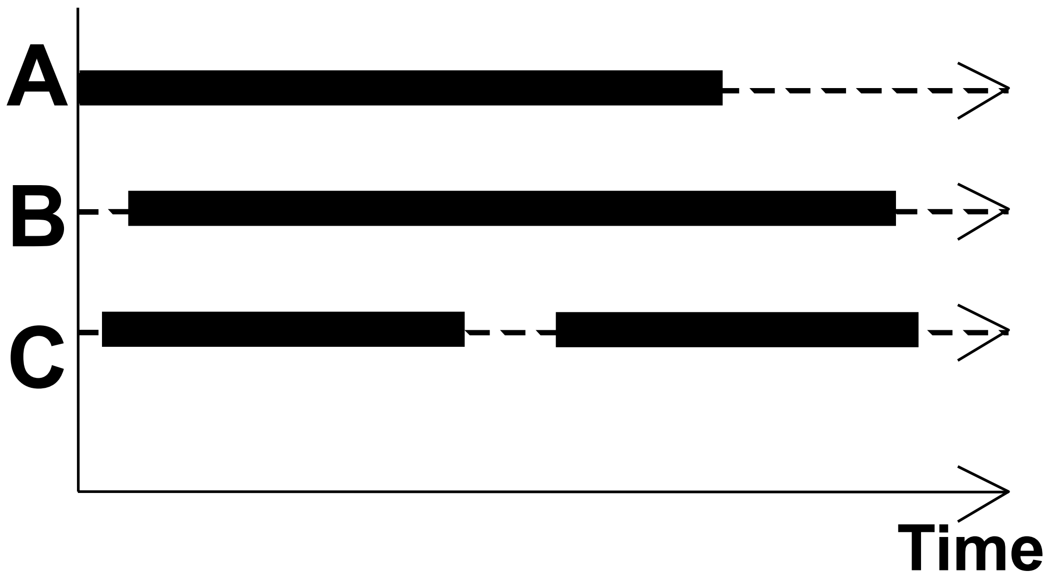
- **Concurrency** (aka. Pseudo-Concurrent Execution)
  - **Logically** simultaneous processing
    - Does not imply multiple processing elements (PEs)
    - Requires interleaved execution on a single PE



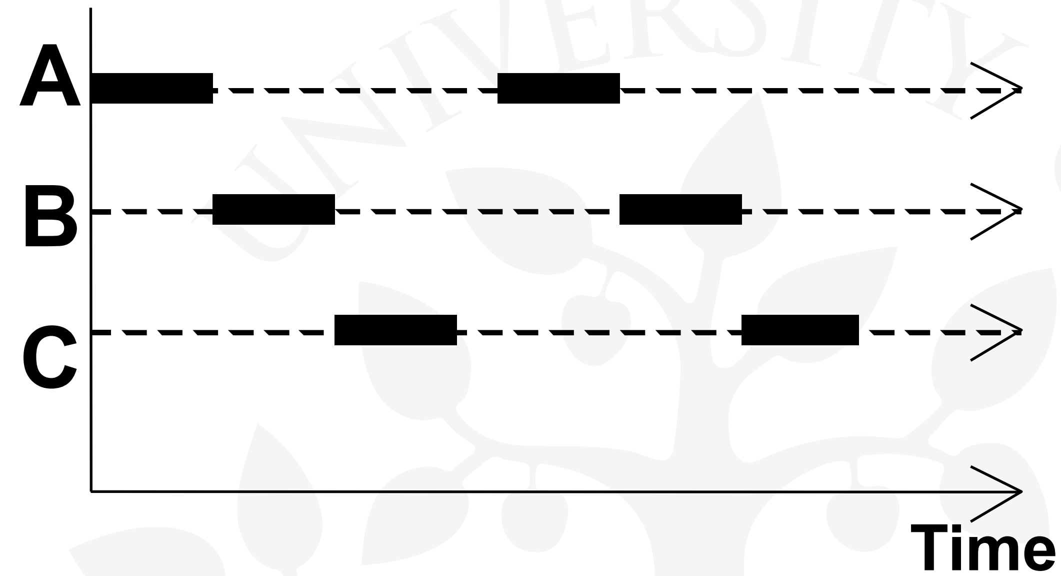
Parallelism vs Concurrency

University of Southern Denmark

- Parallelism



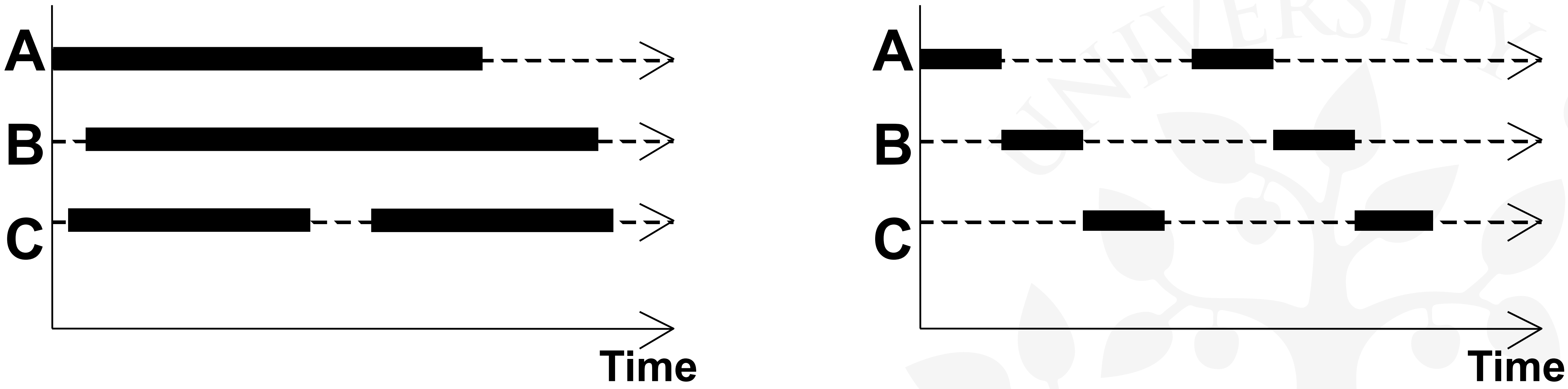
- Concurrency



Parallelism vs Concurrency

UNIVERSITY OF SOUTHERN DENMARK

- **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.

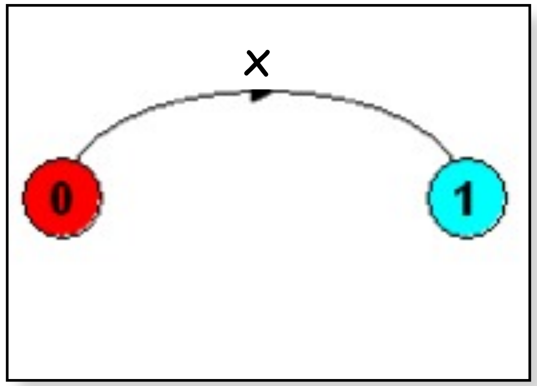
3.1 Modelling Concurrency

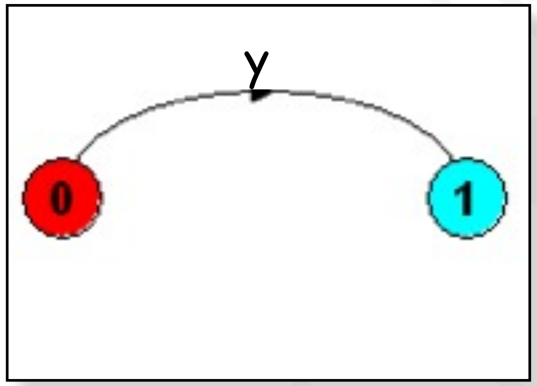
3.1 Modelling Concurrency

- How do we model concurrency?

3.1 Modelling Concurrency

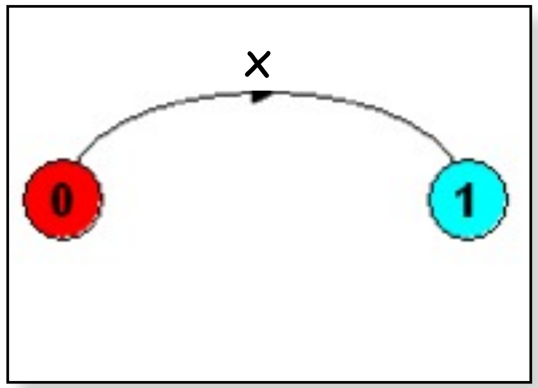
- How do we model concurrency?

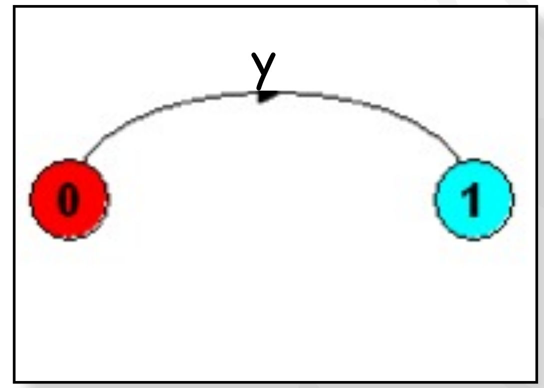




3.1 Modelling Concurrency

- How do we model concurrency?

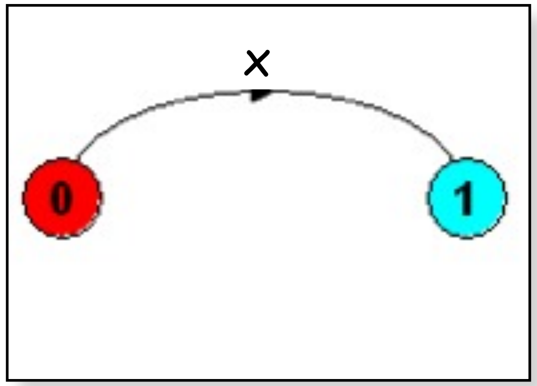


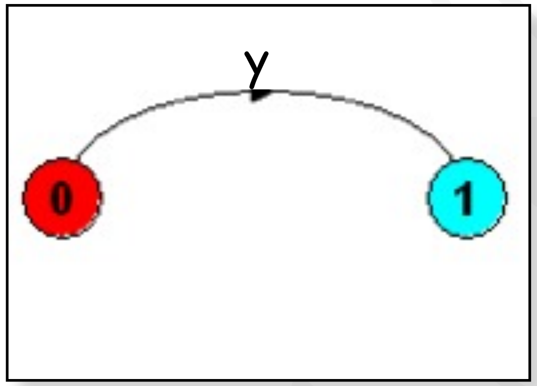


Possible execution sequences?

3.1 Modelling Concurrency

- How do we model concurrency?





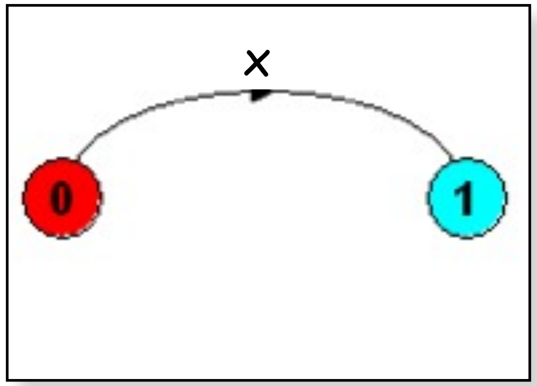
Possible execution sequences?

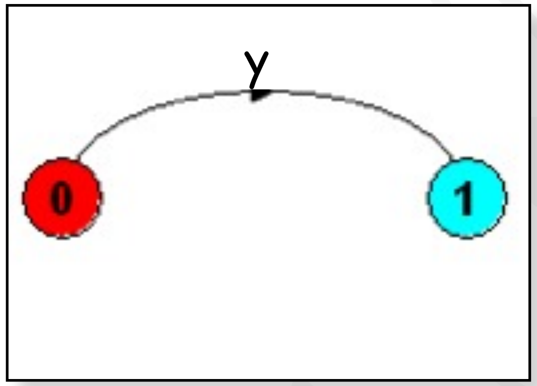
- x ; y

3.1 Modelling Concurrency

UNIVERSITY OF SOUTHERN DENMARK

- How do we model concurrency?



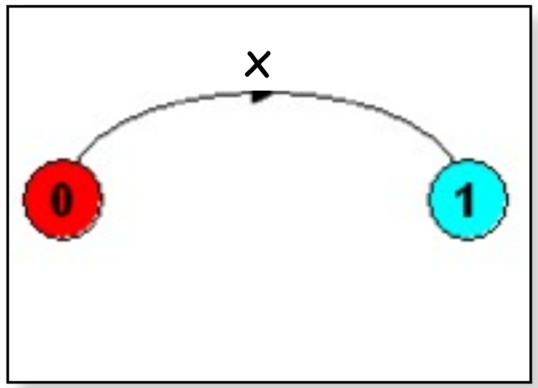


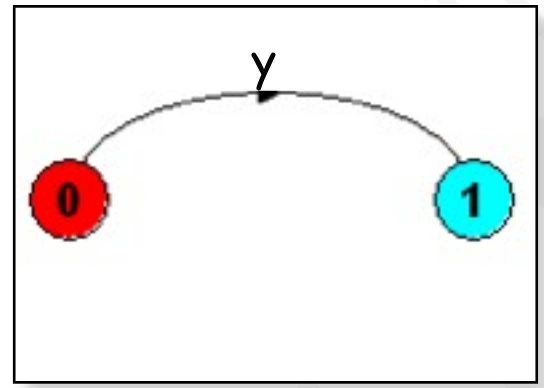
Possible execution sequences?

- x ; y
- y ; x

3.1 Modelling Concurrency

- How do we model concurrency?



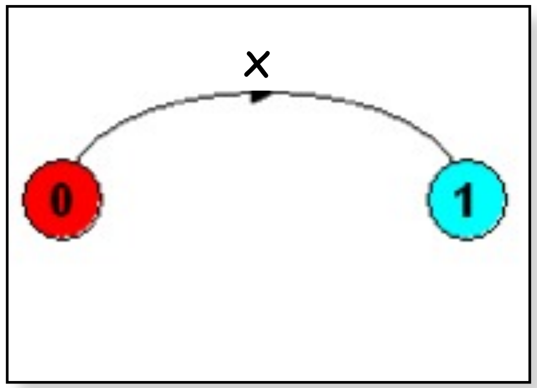


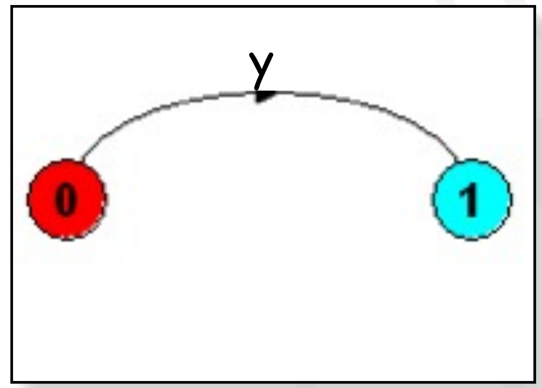
Possible execution sequences?

- x ; y
- y ; x
- x || y

3.1 Modelling Concurrency

- How do we model concurrency?



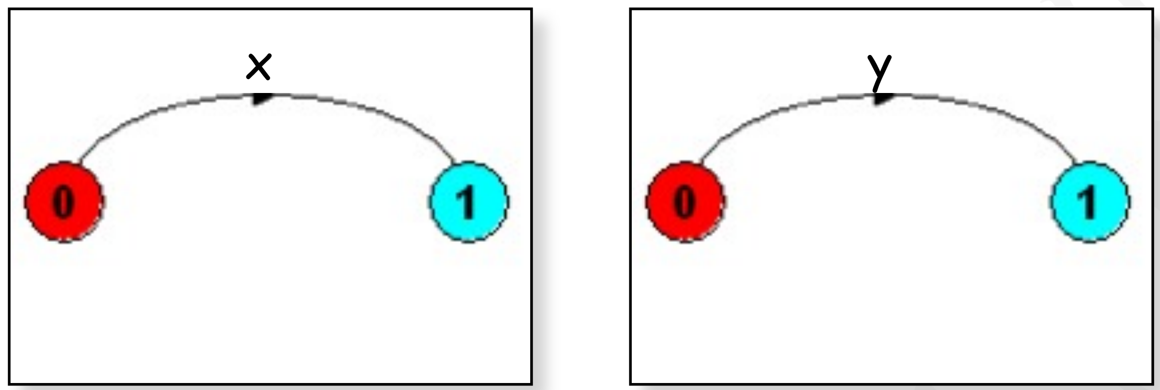


Possible execution sequences?
- x ; y
- y ; x
- ~~x || y~~

Asynchronous model of execution

3.1 Modelling Concurrency

- How do we model concurrency?



Possible execution sequences?

- x ; y
- y ; x
- ~~x || y~~

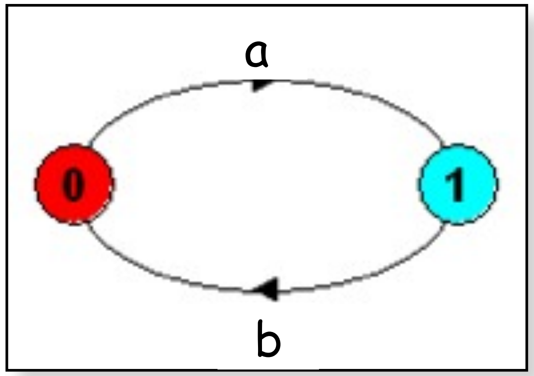
Asynchronous
model of execution

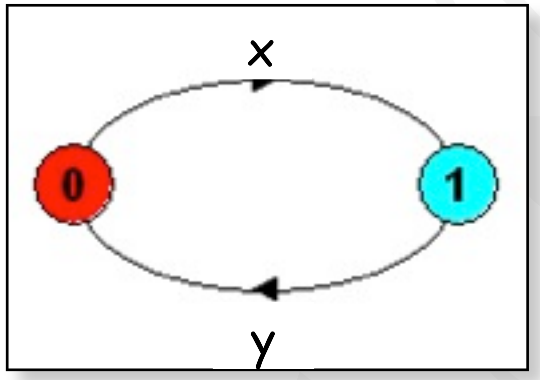
- Arbitrary relative order of actions from different processes (**interleaving** but preservation of each process order)

3.1 Modelling Concurrency

3.1 Modelling Concurrency

- How should we model process execution speed?

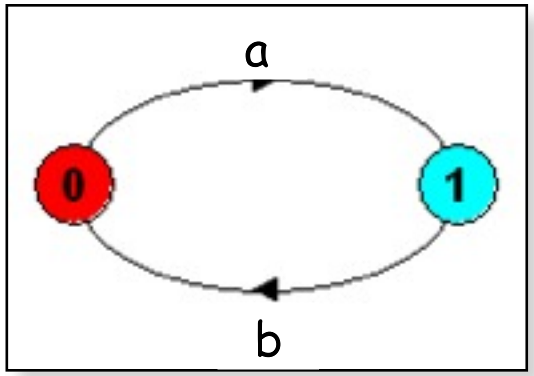


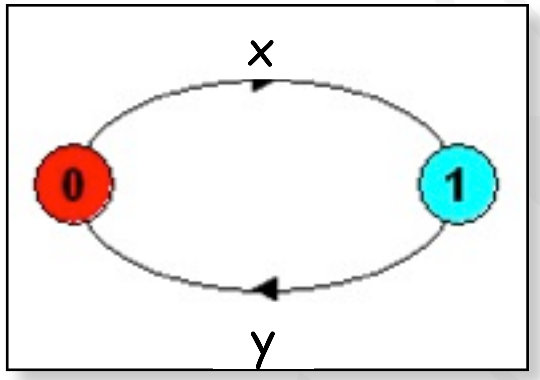


3.1 Modelling Concurrency

UNIVERSITY OF SOUTHERN DENMARK

- How should we model process execution speed?

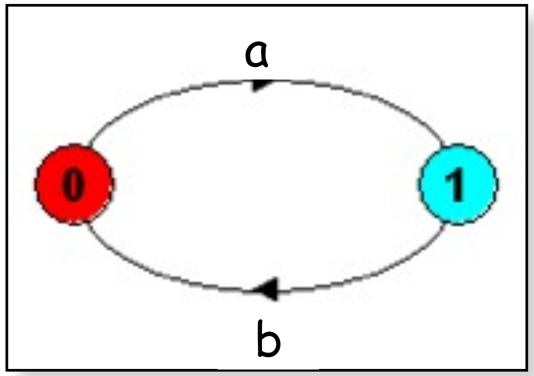


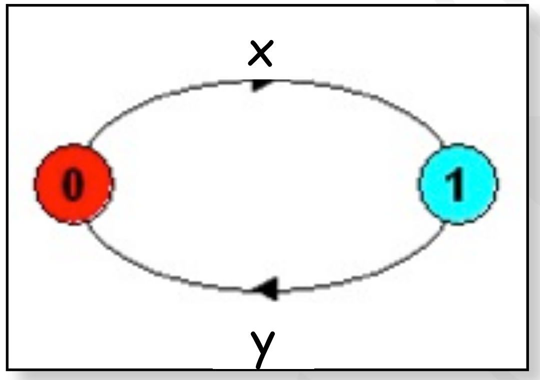


  - We choose to abstract away time:

3.1 Modelling Concurrency

- How should we model process execution speed?

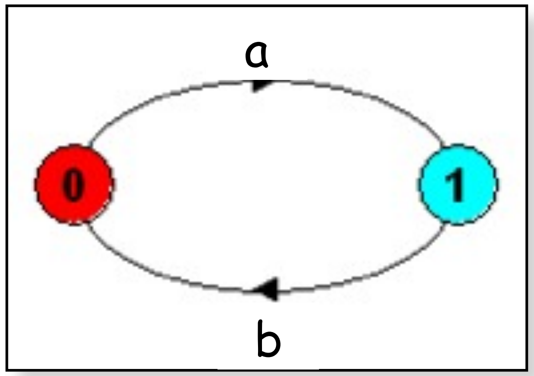


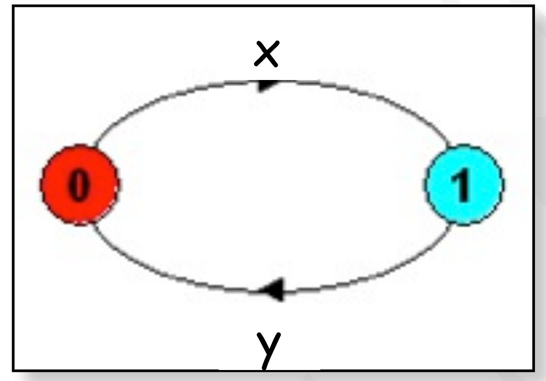


- We choose to abstract away time:
  - Arbitrary speed!

3.1 Modelling Concurrency

- How should we model process execution speed?



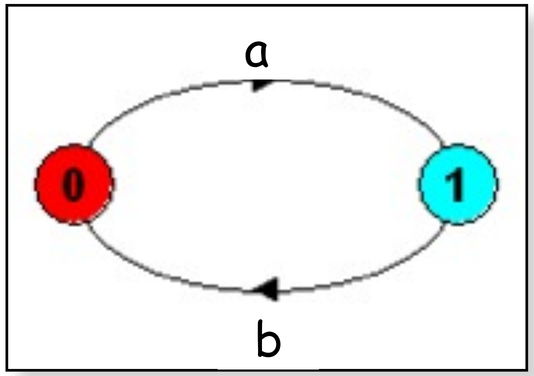


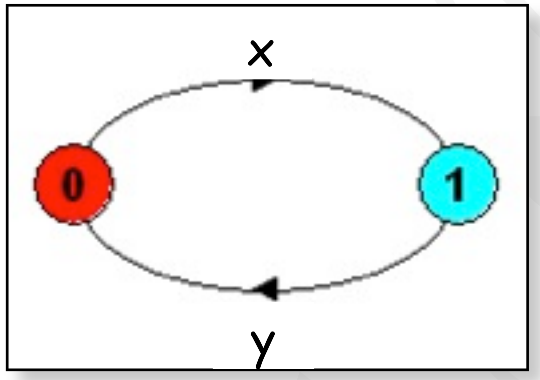
- We choose to abstract away time:
  - Arbitrary speed!

-: we can say nothing of real-time properties

3.1 Modelling Concurrency

- How should we model process execution speed?





- We choose to abstract away time:
  - Arbitrary speed!

-: we can say nothing of real-time properties

+: independent of architecture, processor speed, scheduling policies, …

Parallel Composition - Action Interleaving

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

Parallel Composition - Action Interleaving

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).
```

Parallel Composition - Action Interleaving

University of Southern Denmark

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).
```

Parallel Composition - Action Interleaving

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).
```

Parallel Composition - Action Interleaving

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).
```

Possible traces as a result of action interleaving?

Parallel Composition - Action Interleaving

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).
```

Possible traces as a result of action interleaving?

- `scratch→think→talk`

Parallel Composition - Action Interleaving

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).
```

Possible traces as a result of action interleaving?

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

Parallel Composition - Action Interleaving

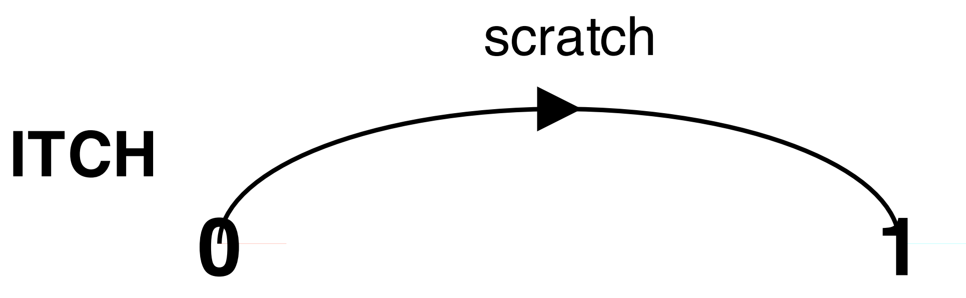
If P and Q are processes then **(P||Q)** represents the concurrent execution of P and Q. The operator '||' is the parallel composition operator.

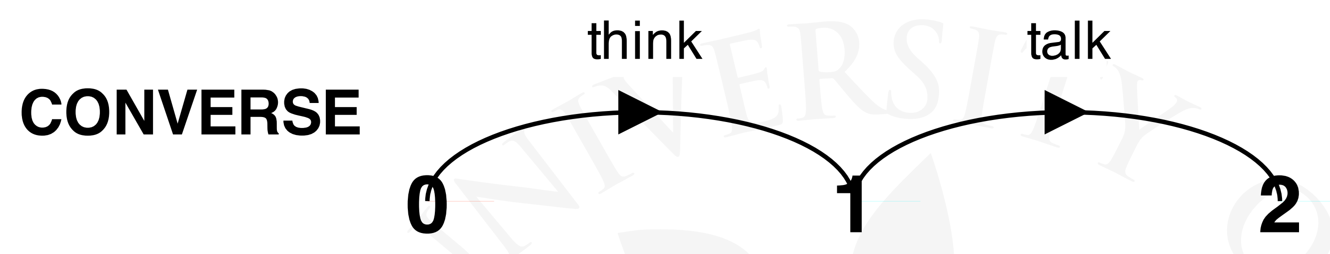
```
ITCH      = (scratch->STOP).
CONVERSE = (think->talk->STOP).

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

Possible traces as a result of action interleaving?

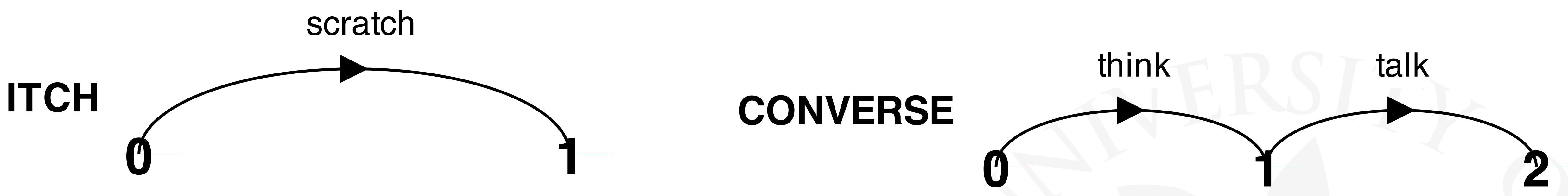
- scratch→think→talk
- think→scratch→talk
- think→talk→scratch

Parallel Composition - Action Interleaving





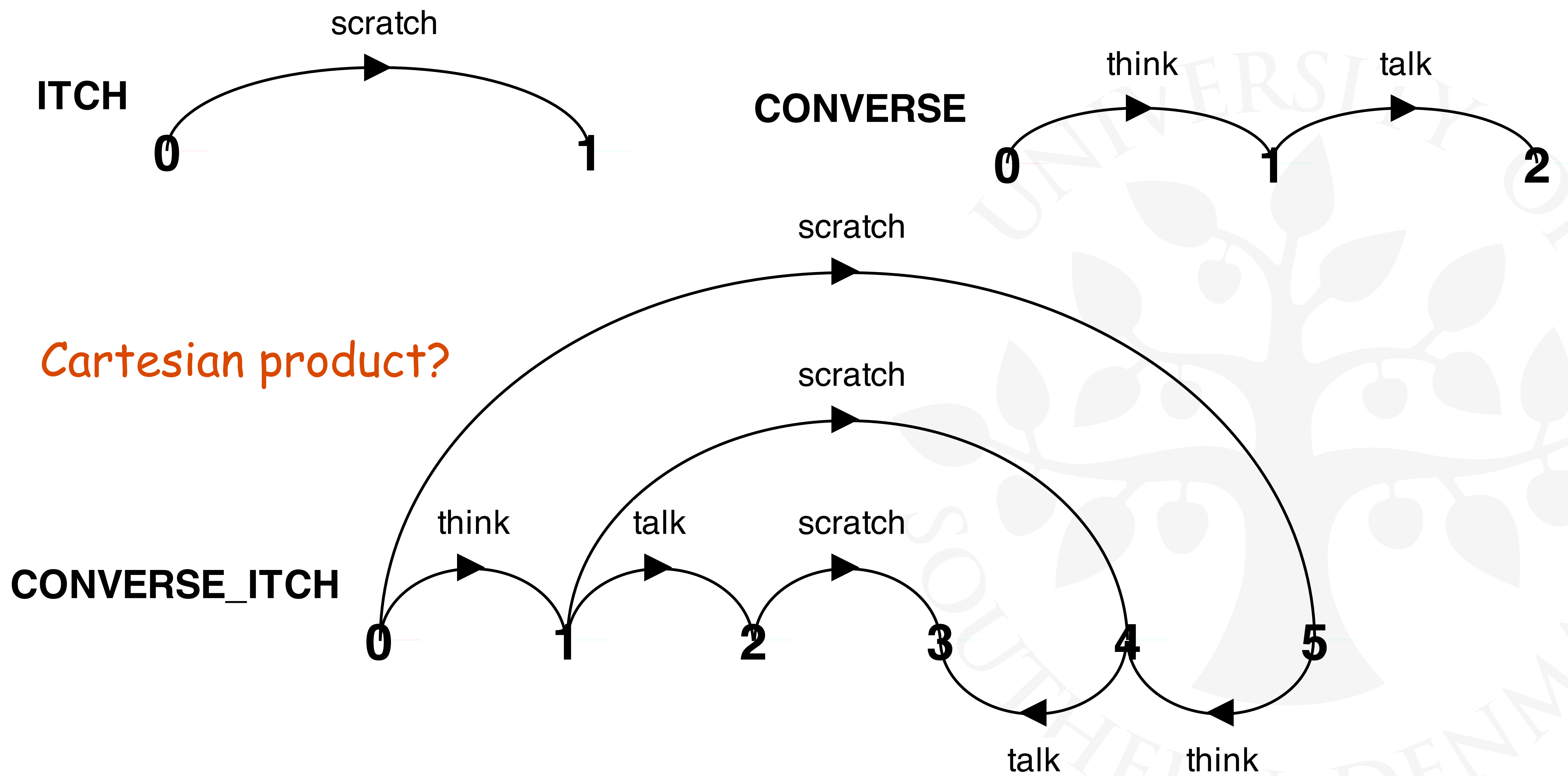
Parallel Composition - Action Interleaving



Cartesian product?

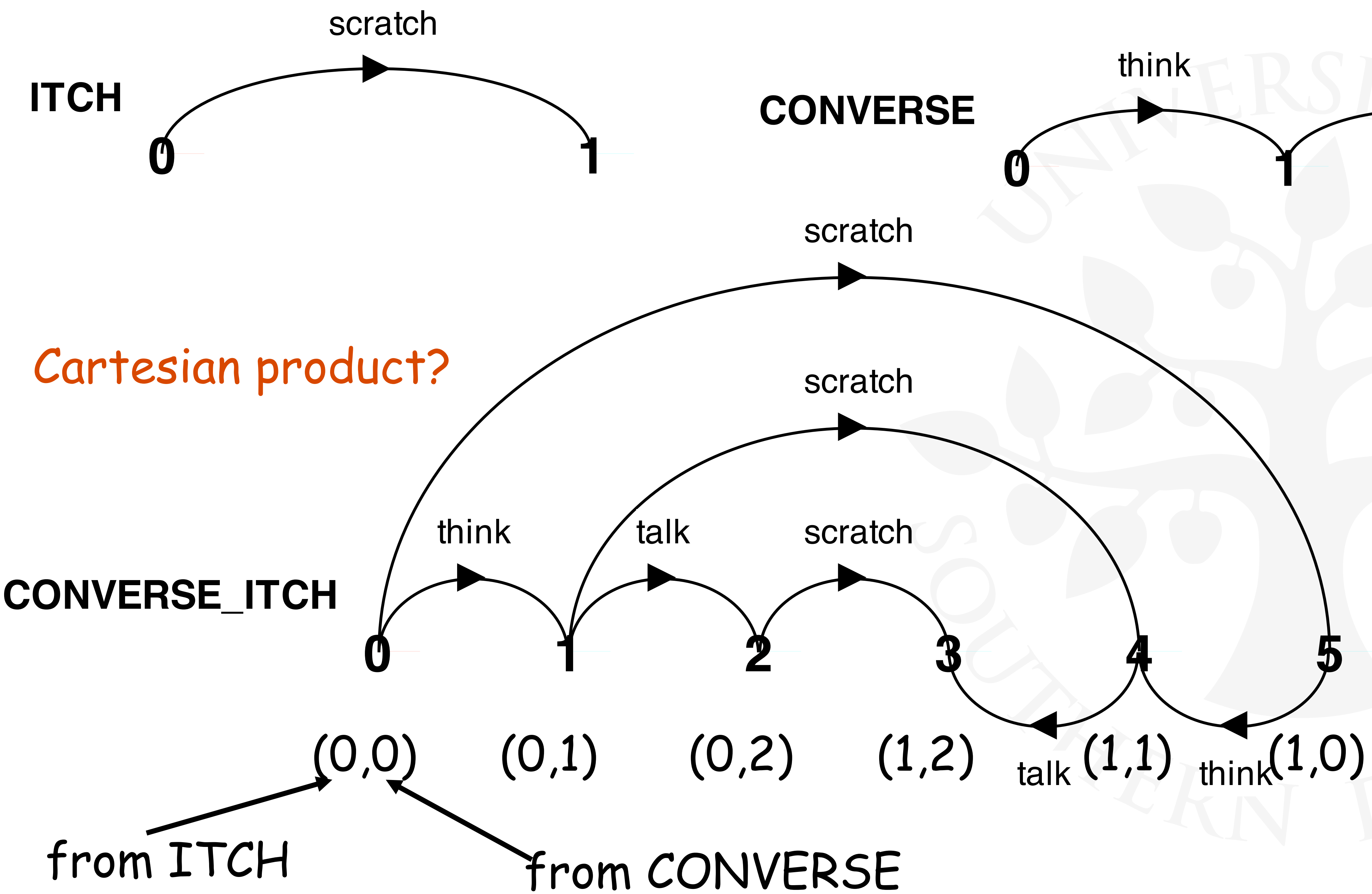
Parallel Composition - Action Interleaving

University of Southern Denmark

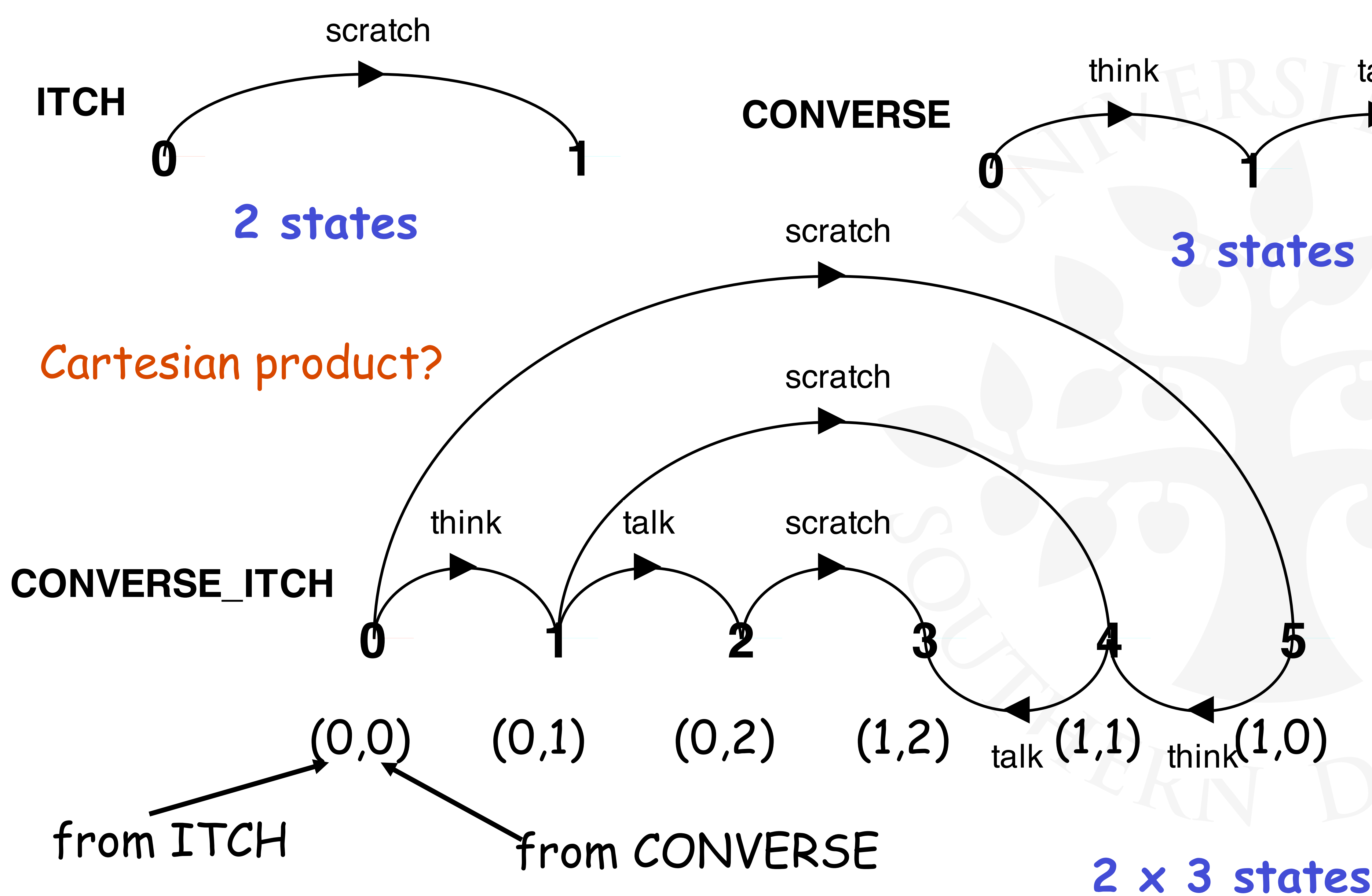


Cartesian product?

Parallel Composition - Action Interleaving



Parallel Composition - Action Interleaving



Parallel Composition - Algebraic Laws

Parallel Composition - Algebraic Laws

Commutative: `(P||Q) = (Q||P)`

Parallel Composition - Algebraic Laws

Commutative: `(P||Q) = (Q||P)`

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

Parallel Composition - Algebraic Laws

Commutative: `(P||Q) = (Q||P)`

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

Parallel Composition - Algebraic Laws

Commutative: `(P||Q) = (Q||P)`

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

Small example:

Parallel Composition - Algebraic Laws

University of Southern Denmark

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

Small example:

```
MALTHE = (climbTree->fall->MALTHE).
OSKAR  = (run->jump->OSKAR).

||MALTHE_OSKAR = (MALTHE || OSKAR).
```

Parallel Composition - Algebraic Laws

University of Southern Denmark

Commutative: `(P||Q) = (Q||P)`

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

Small example:

```
MALTHE = (climbTree->fall->MALTHE).
OSKAR  = (run->jump->OSKAR).

||MALTHE_OSKAR = (MALTHE || OSKAR).
```

LTS? Traces? Number of states?

Modelling Interaction - Shared Actions


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

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

Modelling Interaction - Shared Actions

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

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

- 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.

Modelling Interaction - Shared Actions

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

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

MAKE1 synchronises with **USE1** when **ready**.

◆ 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.

Modelling Interaction - Shared Actions


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

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

MAKE1 synchronises with **USE1** when **ready**.

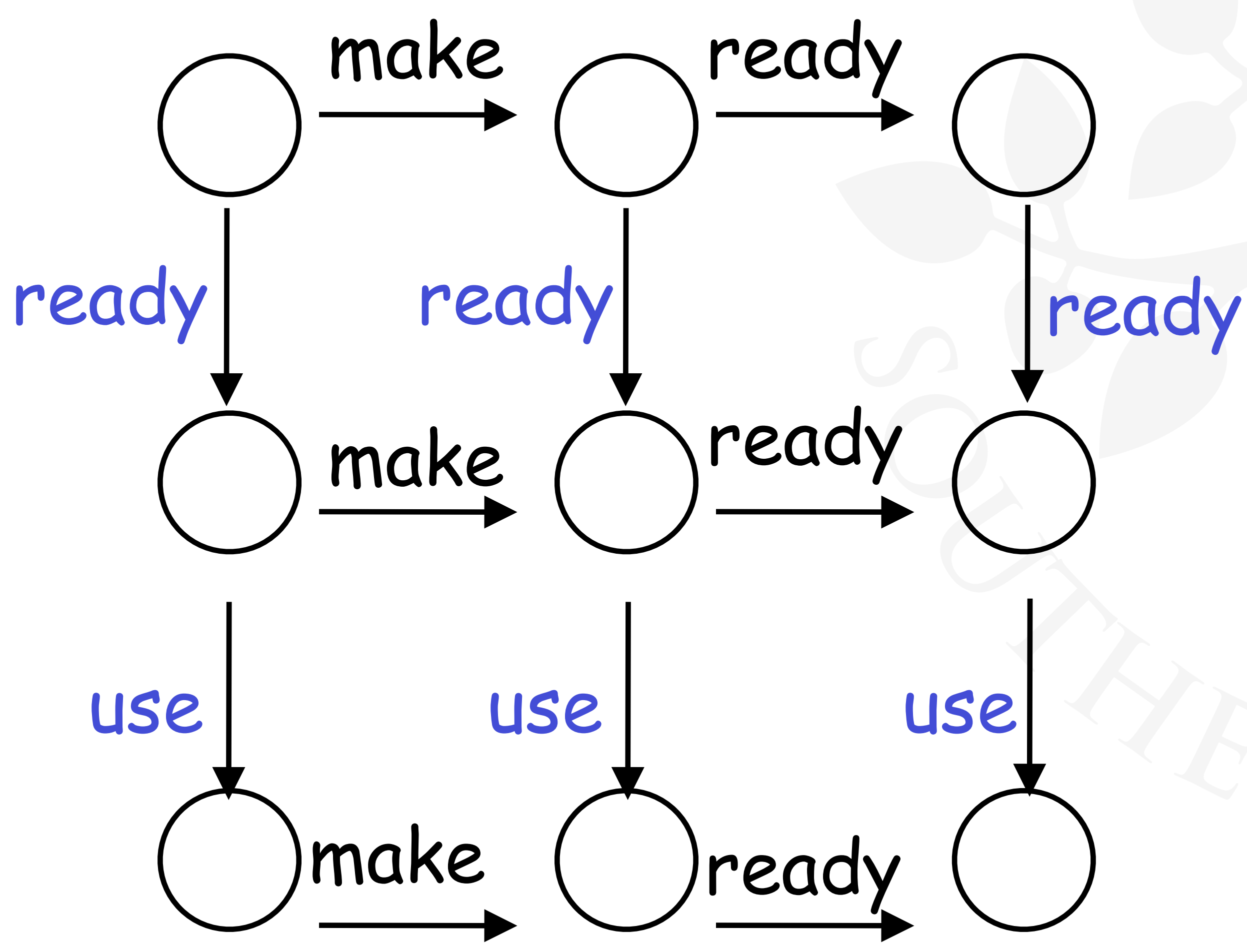
LTS? Traces? Number of states?

- 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.

Modelling Interaction - Example


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

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

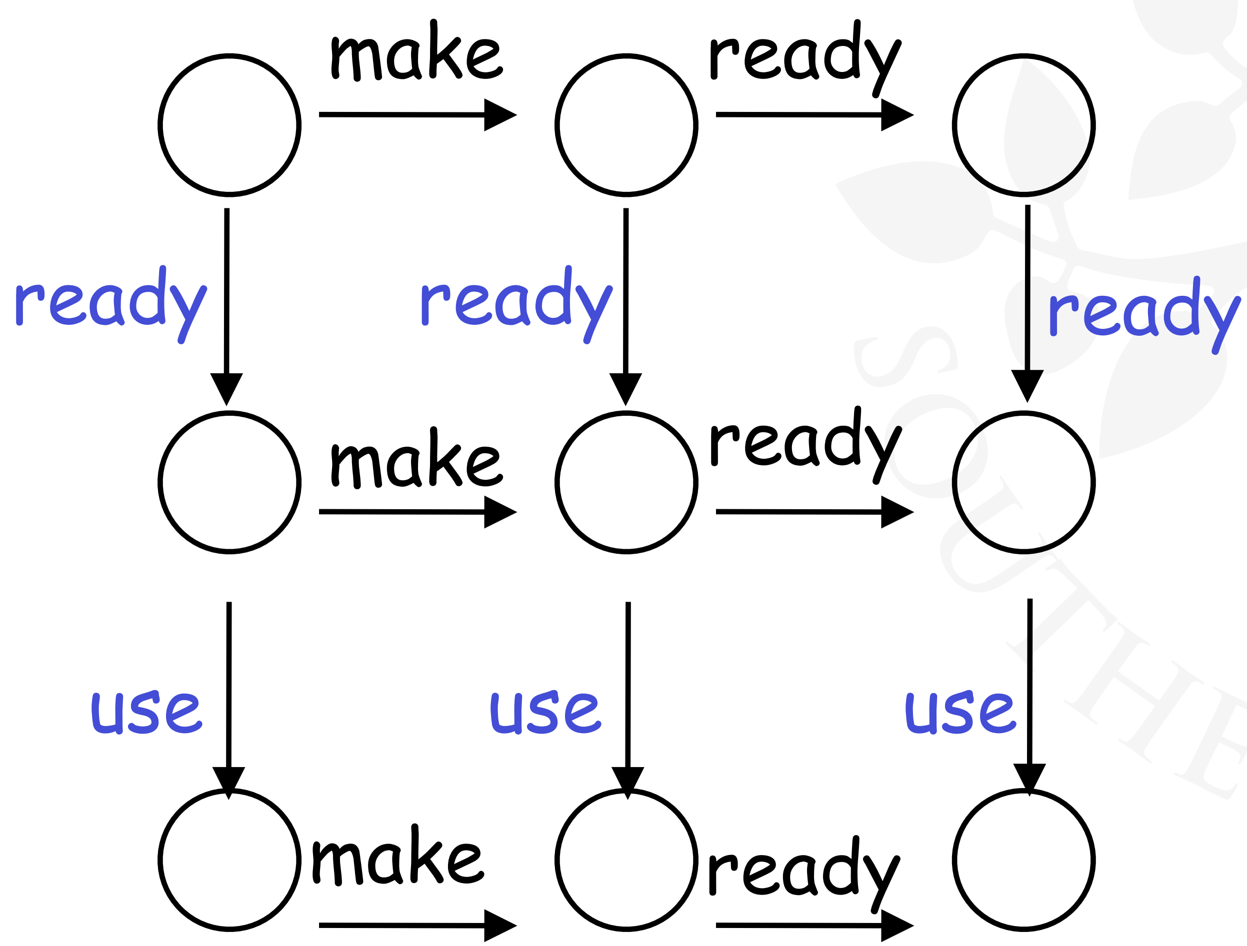


Modelling Interaction - Example

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

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

3 states

3 states

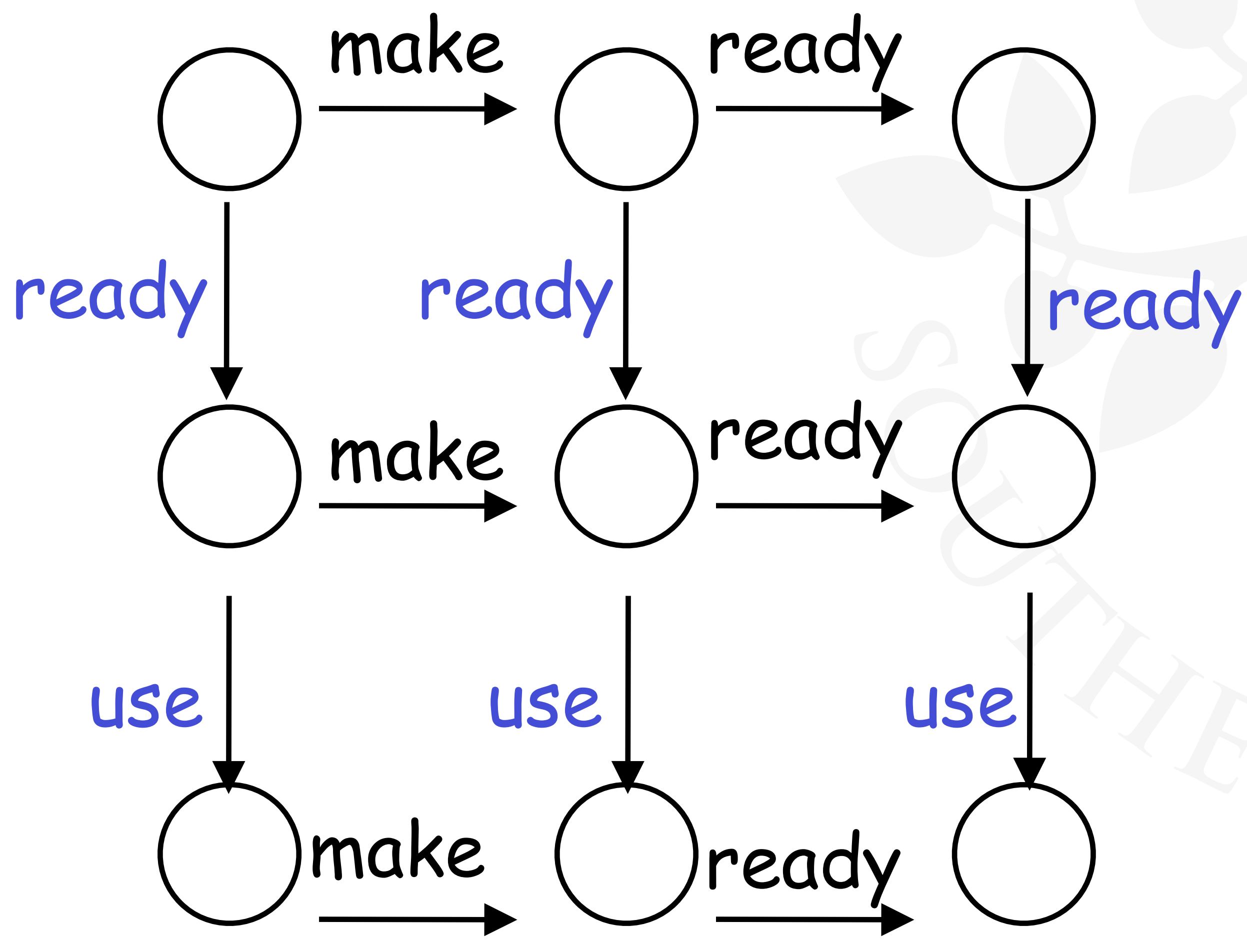


Modelling Interaction - Example

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

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

3 states

3 states



3 x 3 states?

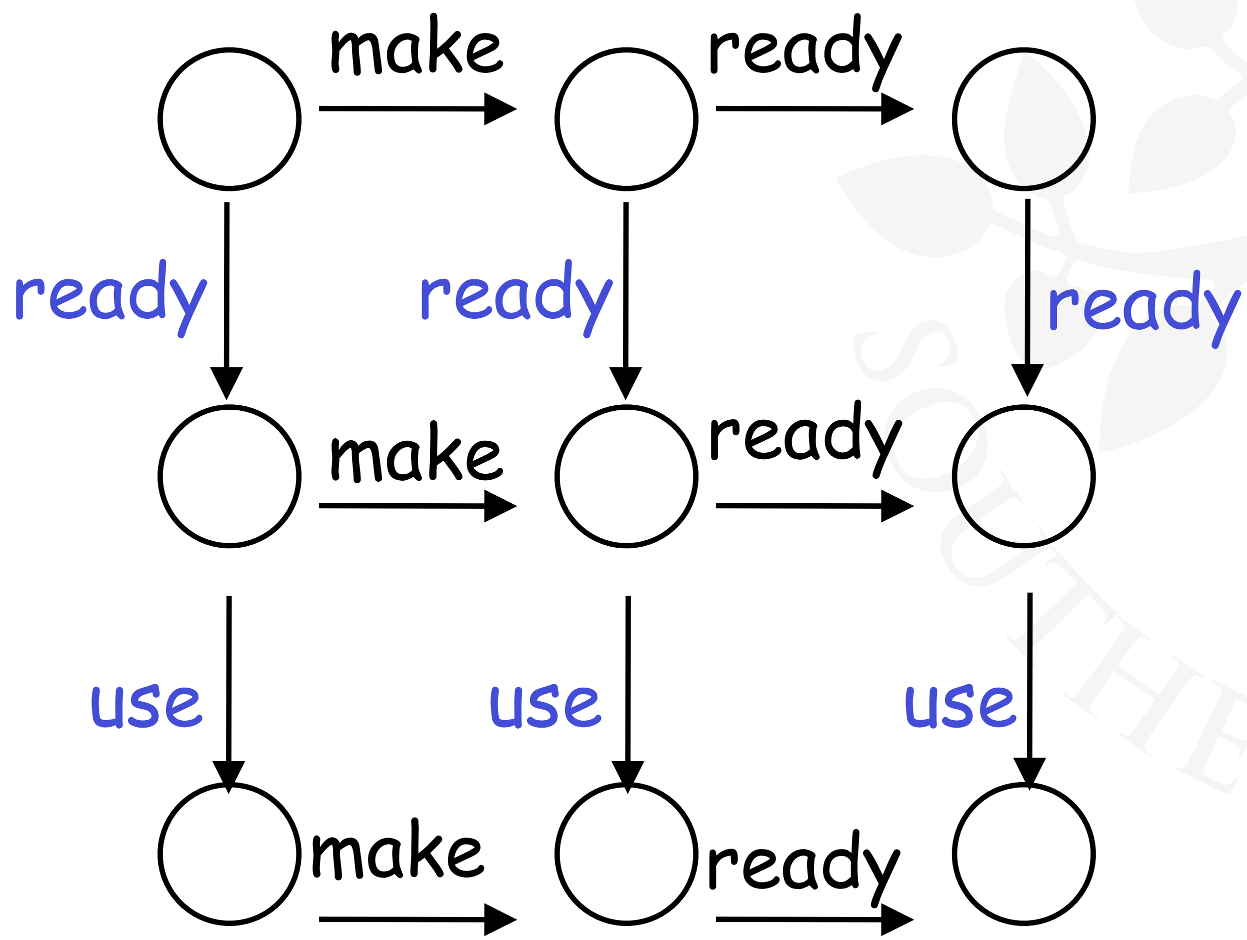
Modelling Interaction - Example

UNIVERSITY OF SOUTHERN DENMARK

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

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

3 states

3 states



3 x 3 states?

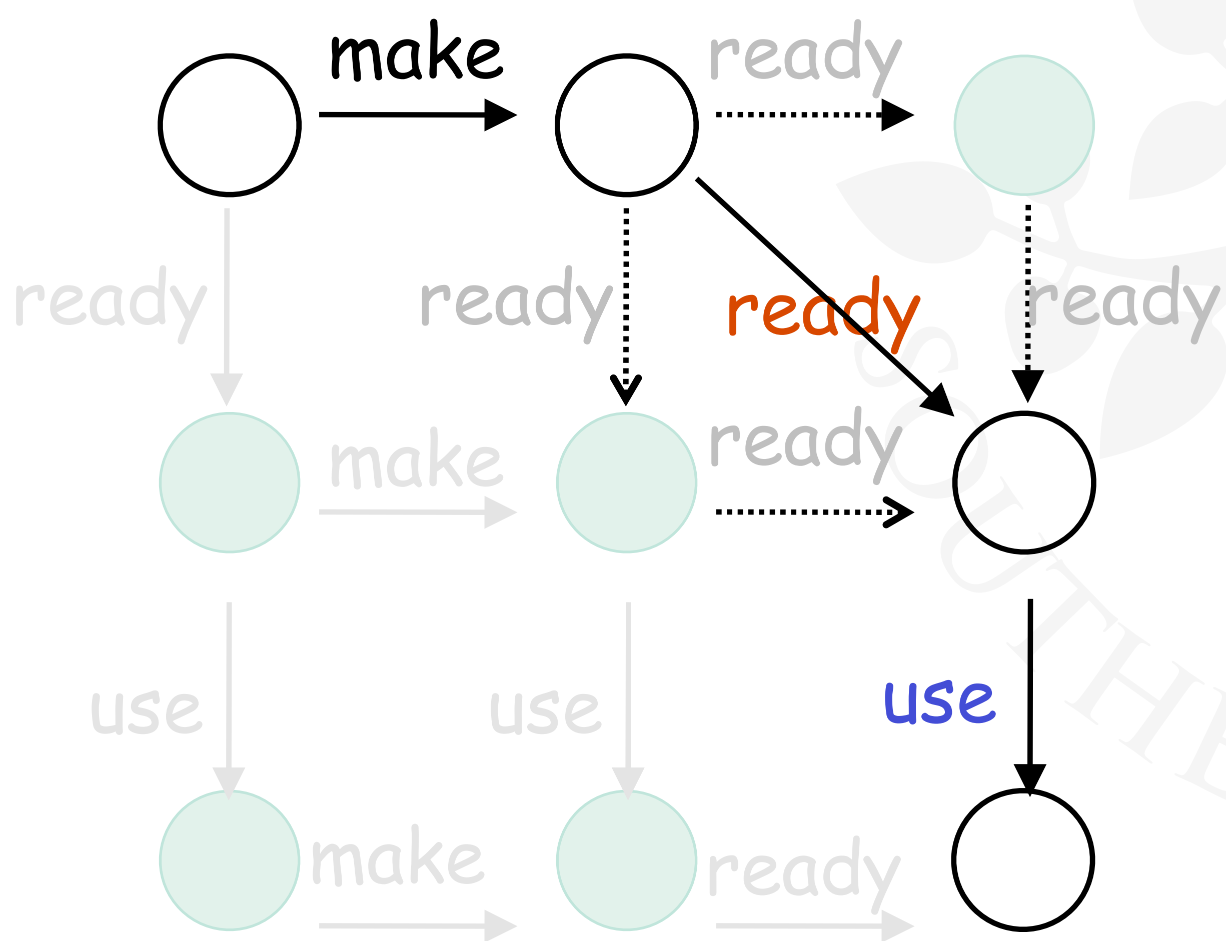
No...!

Modelling Interaction - Example

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

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

3 states

3 states

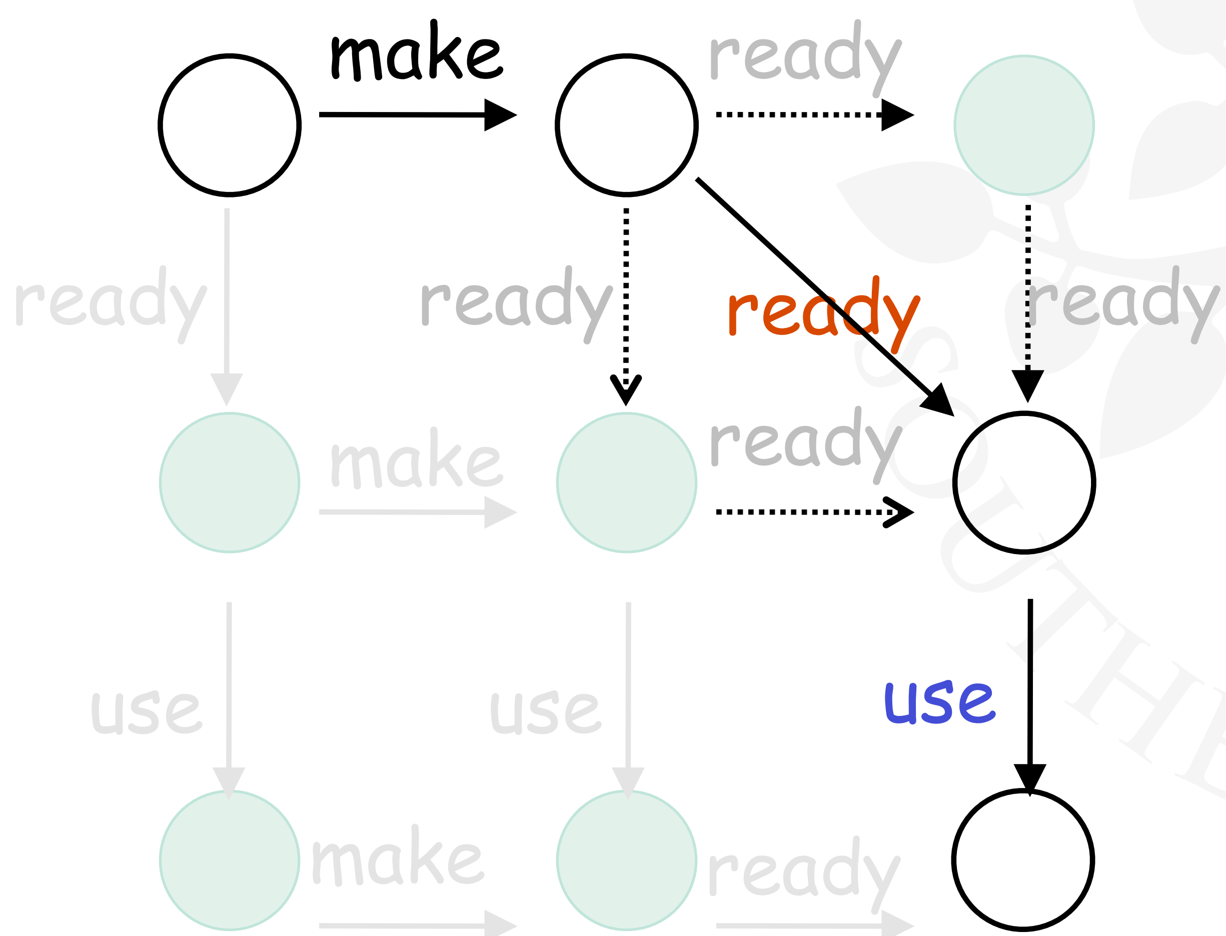


Modelling Interaction - Example

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

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

3 states

3 states



4 states!

Interaction may constrain the overall behaviour !

Example

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

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

Example

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

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

2 states

2 states

Example


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

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

2 states

2 states

LTS? Traces? Number of states?

Example

University of Southern Denmark

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

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

2 states

2 states

LTS?   Traces?   Number of states?

```
P = (a -> P | b -> P).
Q = (c -> Q) + {a}.

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

Example

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

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

2 states

2 states

LTS?   Traces?   Number of states?

```
P = (a -> P | b -> P).
Q = (c -> Q) + {a}.

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

LTS?   Traces?

Modelling Interaction - Example

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

Modelling Interaction - Example


```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

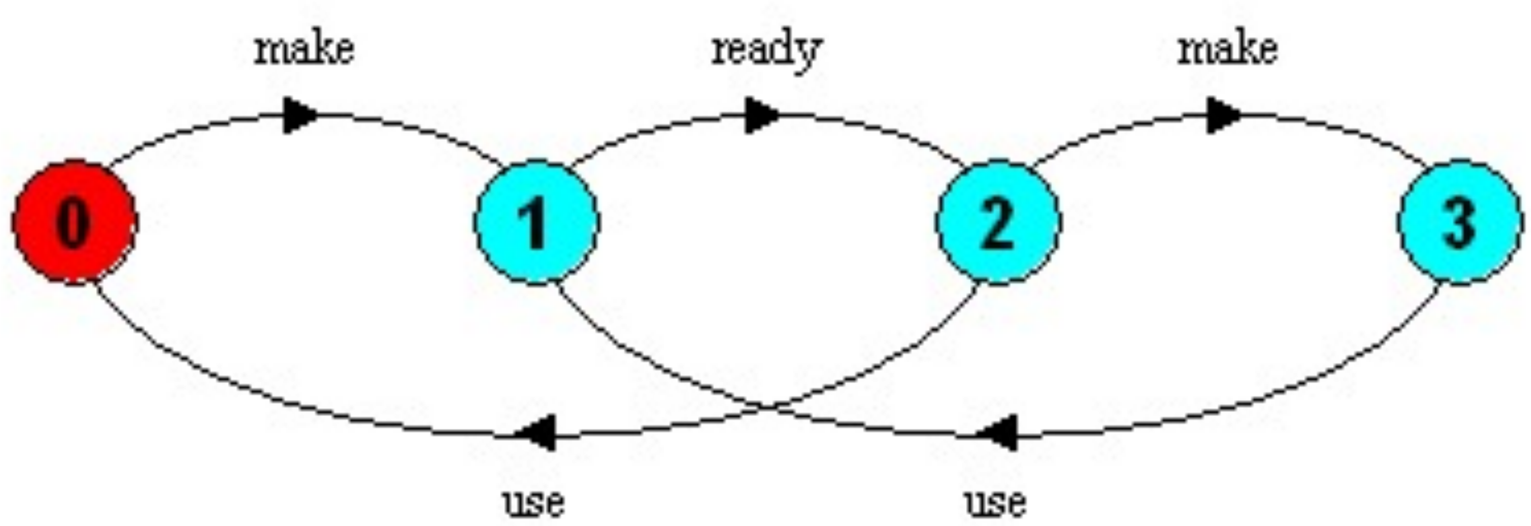
LTS? Traces?

Modelling Interaction - Example

UNIVERSITY OF SOUTHERN DENMARK

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

LTS?   Traces?

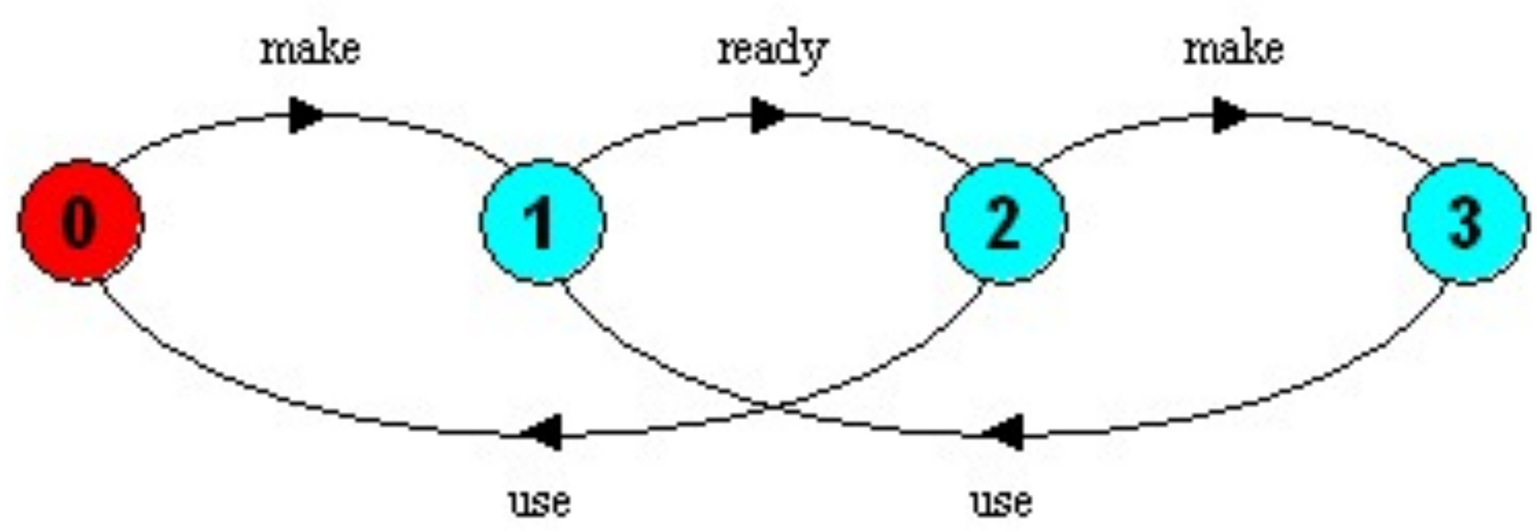


Modelling Interaction - Example

University of Southern Denmark

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

LTS? Traces?



Can we make sure the MAKER does not "get ahead of" the USER (i.e. never **make** before **use**); and if so, how?

Modelling Interaction - Handshake

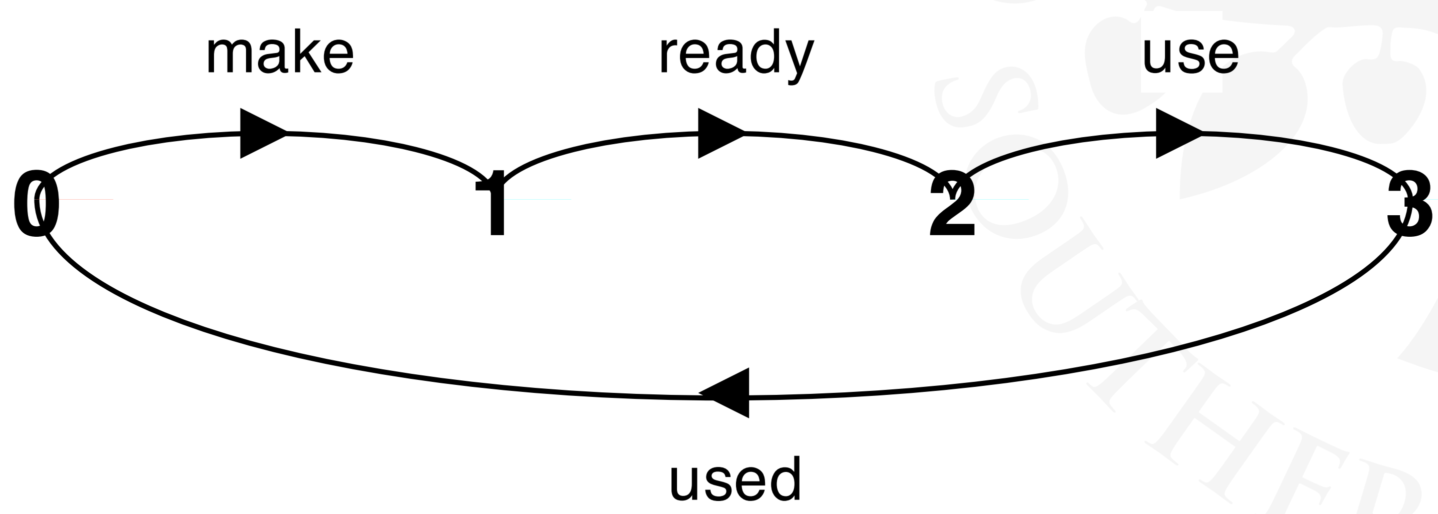
A handshake is an action acknowledged by another process:

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

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

Modelling Interaction - Handshake

A handshake is an action acknowledged by another process:

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

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



Modelling Interaction - Multiple Processes

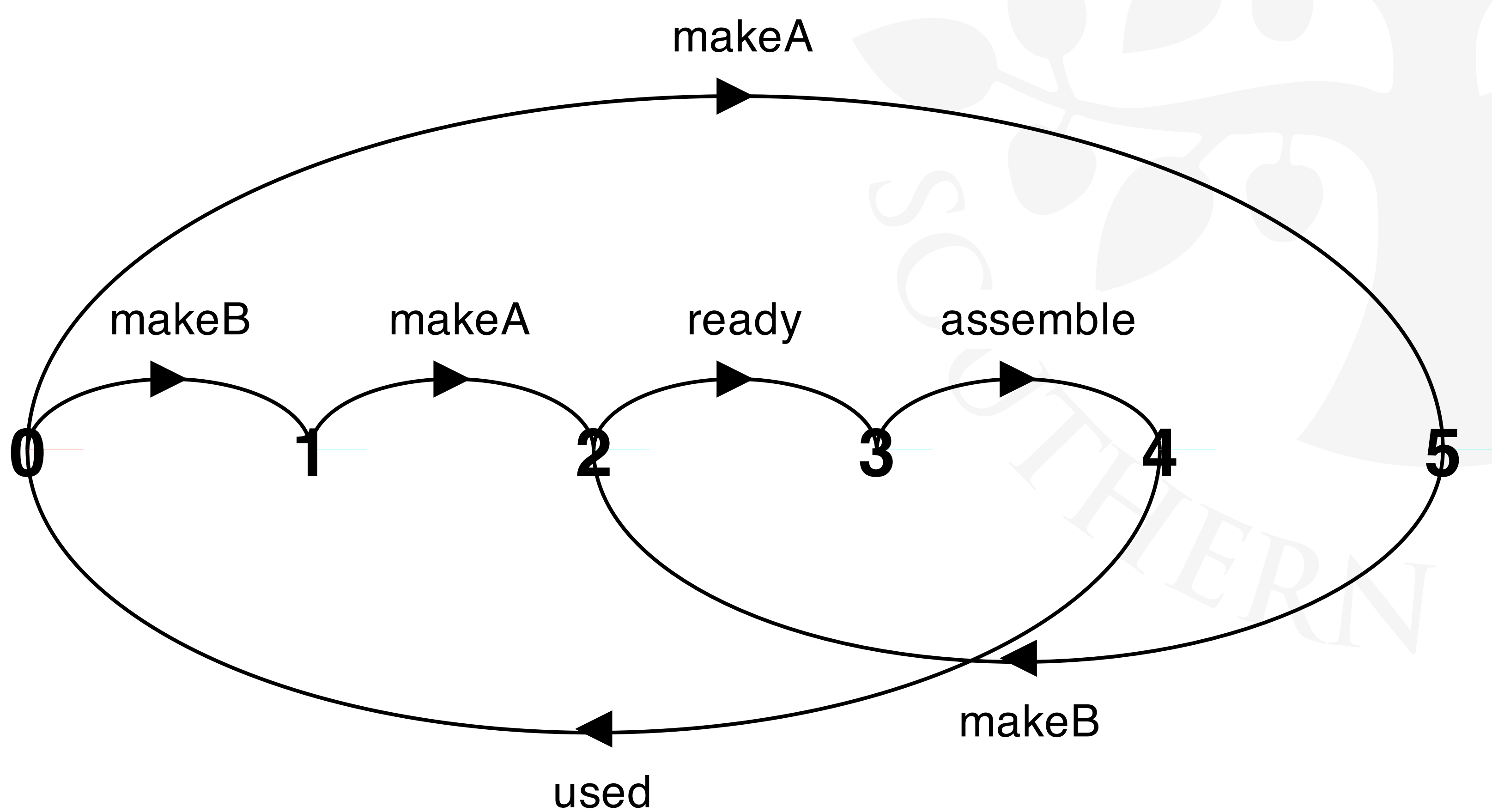
Multi-party synchronisation:

```
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).
```

Modelling Interaction - Multiple Processes

Multi-party synchronisation:

```
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

UNIVERSITY OF SOUTHERN DENMARK

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).
```

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).
```

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?

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).
```

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
```

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH = (on->off->SWITCH).
```

LTS? (a:SWITCH)

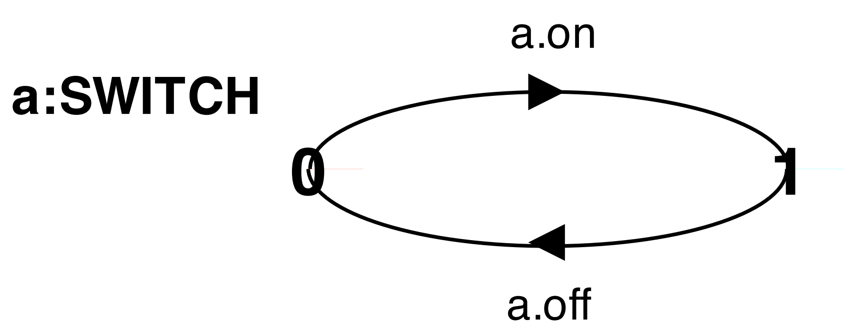
Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
```

LTS? (a:SWITCH)



Process Labelling

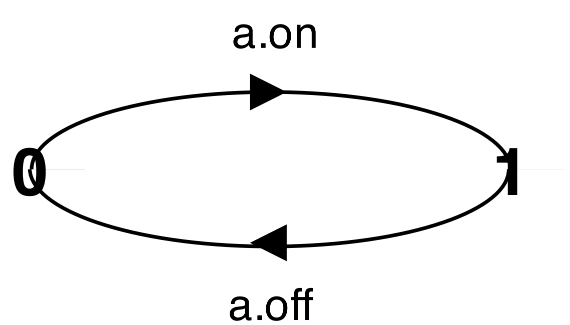
a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

LTS? (a:SWITCH)

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

a:SWITCH



Process Labelling

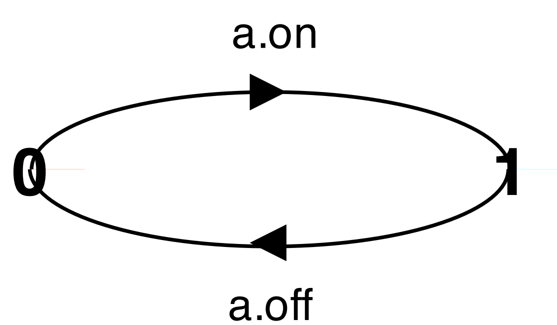
a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

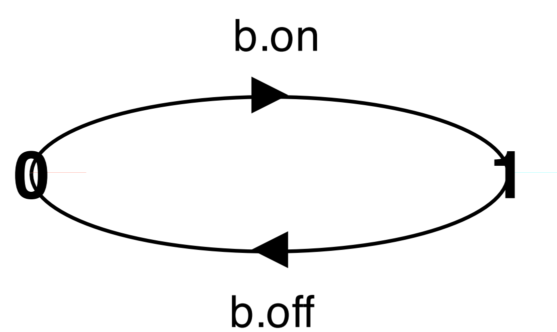
LTS? (a:SWITCH)

```
SWITCH =   (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

a:SWITCH



b:SWITCH



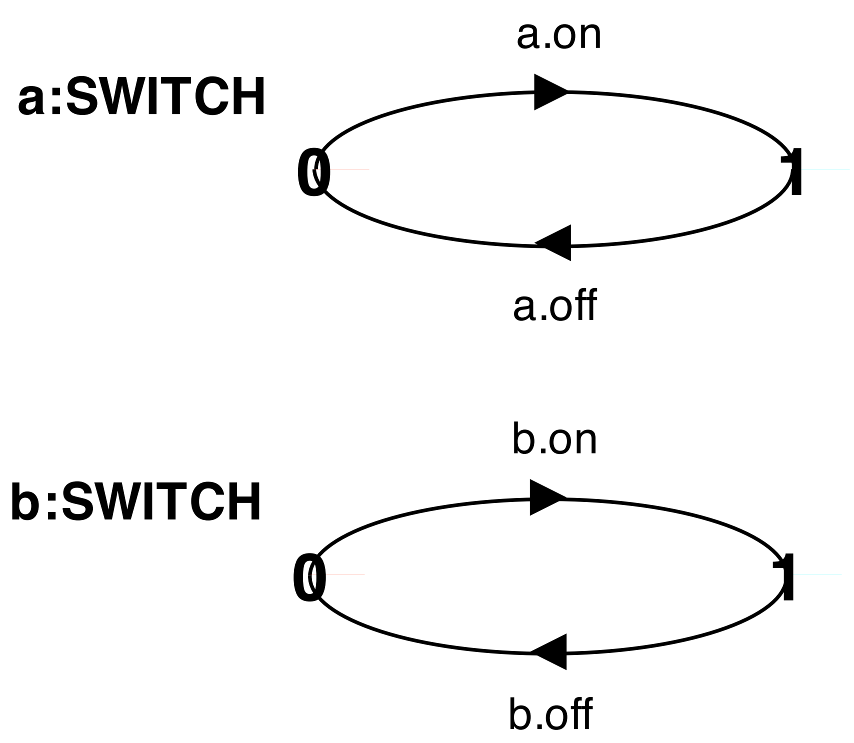
Process Labelling

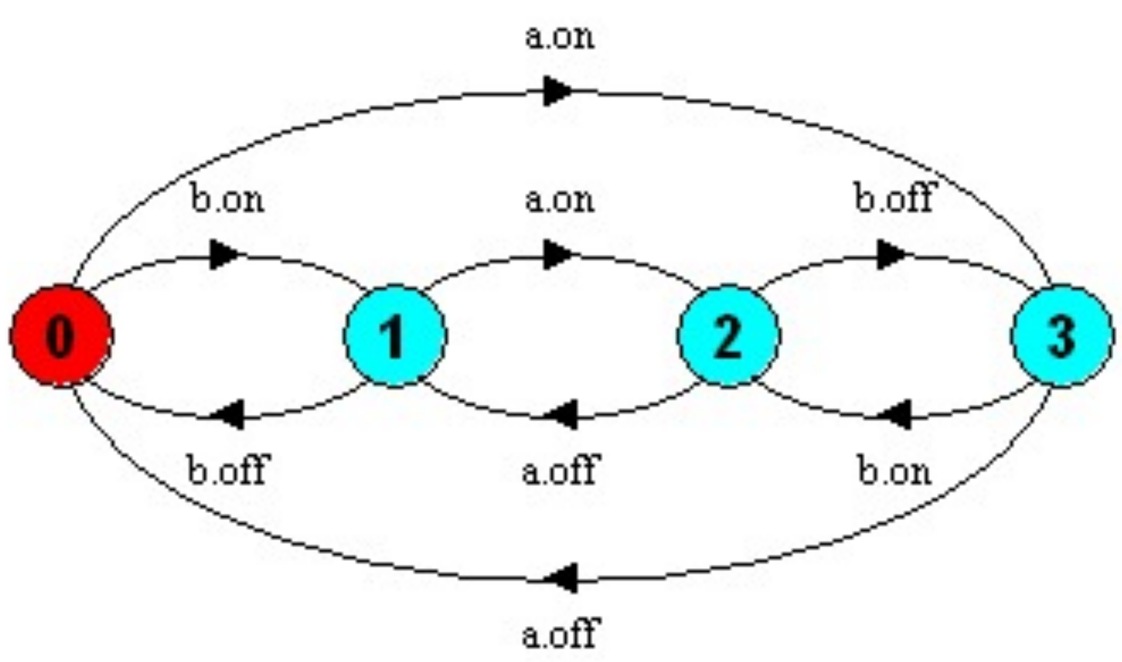
a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

LTS? (a:SWITCH)





Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

Create an array of **instances** of the switch process:

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

Create an array of instances of the switch process:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
```

Process Labelling

a:P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

Create an array of instances of the switch process:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

Process Labelling By A Set Of Prefix Labels

$\{a_1,\ldots,a_n\}::P$ replaces every action label x in the alphabet of P with the labels $a_1.x,\ldots,a_n.x$.

Further, every transition (x -> X) in the definition of P is replaced with the transitions ($\{a_1.x,\ldots,a_n.x\}$ -> X).

Process Labelling By A Set Of Prefix Labels

University of Southern Denmark

$\{a_1,\ldots,a_n\}$::P replaces every action label x in the alphabet of P with the labels $a_1.x,\ldots,a_n.x$.
Further, every transition (x -> X) in the definition of P is replaced with the transitions ($\{a_1.x,\ldots,a_n.x\}$ -> X).

Process prefixing is useful for modelling **shared** resources:

Process Labelling By A Set Of Prefix Labels

{a_1,…,a_n}::P replaces every action label x in the alphabet of P with the labels a_1.x,…,a_n.x.
Further, every transition (x -> X) in the definition of P is replaced with the transitions ({a_1.x,…,a_n.x} -> X).

Process prefixing is useful for modelling **shared** resources:

```
USER      = (acquire->use->release->USER).
```

Process Labelling By A Set Of Prefix Labels

University of Southern Denmark

$\{a_1,\ldots,a_n\}$::P replaces every action label x in the alphabet of P with the labels $a_1.x,\ldots,a_n.x$.

Further, every transition (x -> X) in the definition of P is replaced with the transitions ($\{a_1.x,\ldots,a_n.x\}$ -> X).

Process prefixing is useful for modelling **shared** resources:

```
USER     = (acquire->use->release->USER).
```

```
RESOURCE = (acquire->release->RESOURCE).
```

Process Labelling By A Set Of Prefix Labels

$\{a_1,\ldots,a_n\}$::P replaces every action label x in the alphabet of P with the labels $a_1.x,\ldots,a_n.x$.
Further, every transition (x -> X) in the definition of P is replaced with the transitions ($\{a_1.x,\ldots,a_n.x\}$ -> X).

Process prefixing is useful for modelling **shared** resources:

```
USER       = (acquire->use->release->USER).
```

```
RESOURCE  = (acquire->release->RESOURCE).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

Process Prefix Labels For Shared Resources

```
RESOURCE = (acquire->release->RESOURCE).
```

```
USER     = (acquire->use->release->USER).
```

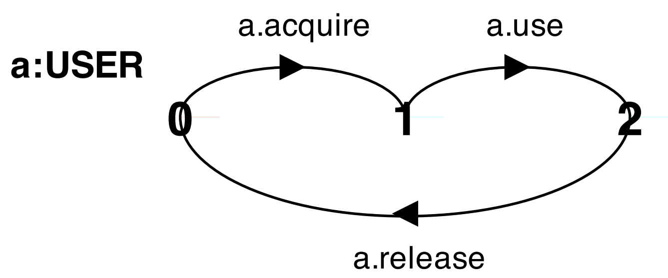
```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

Process Prefix Labels For Shared Resources

```
RESOURCE = (acquire->release->RESOURCE).
```

```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

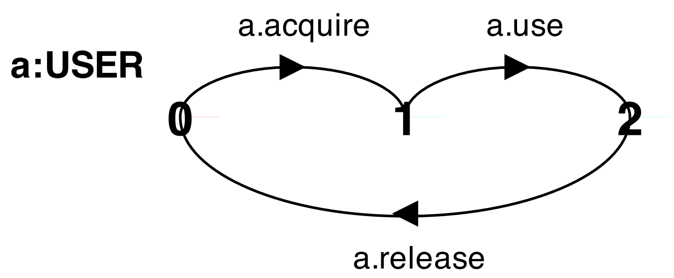


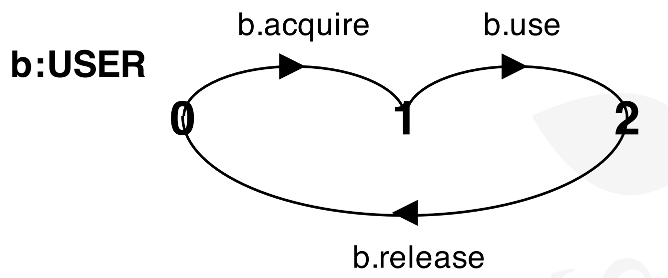
Process Prefix Labels For Shared Resources

```
RESOURCE = (acquire->release->RESOURCE).
```

```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```



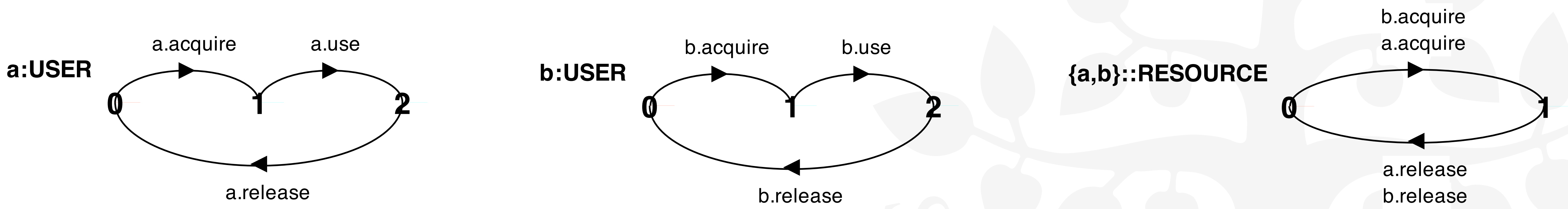


Process Prefix Labels For Shared Resources


```
RESOURCE = (acquire->release->RESOURCE).
```

```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

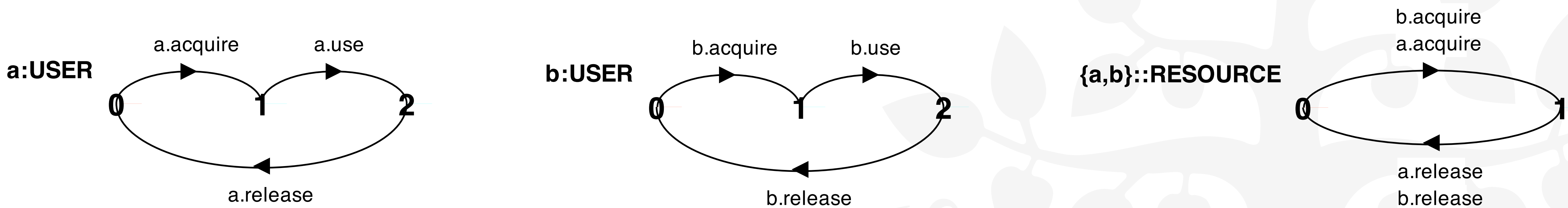


Process Prefix Labels For Shared Resources

```
RESOURCE = (acquire->release->RESOURCE).
```

```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```



How does the model ensure that the user that acquires the resource is the one to release it?

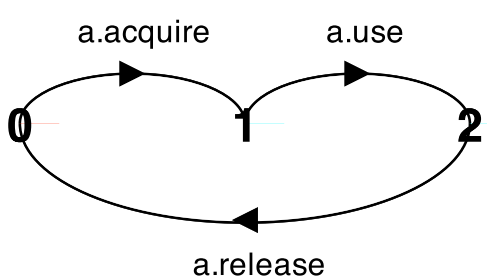
Process Prefix Labels For Shared Resources

```
RESOURCE = (acquire->release->RESOURCE).
```

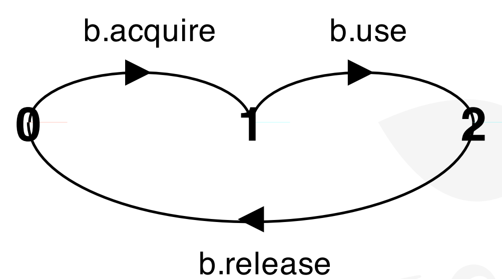
```
USER     = (acquire->use->release->USER).
```

```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

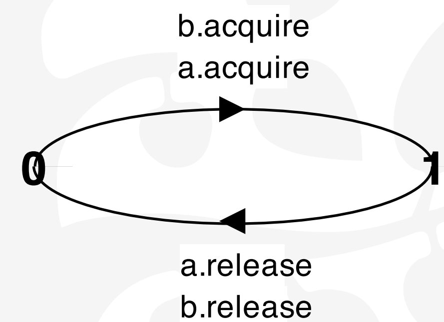
a:USER



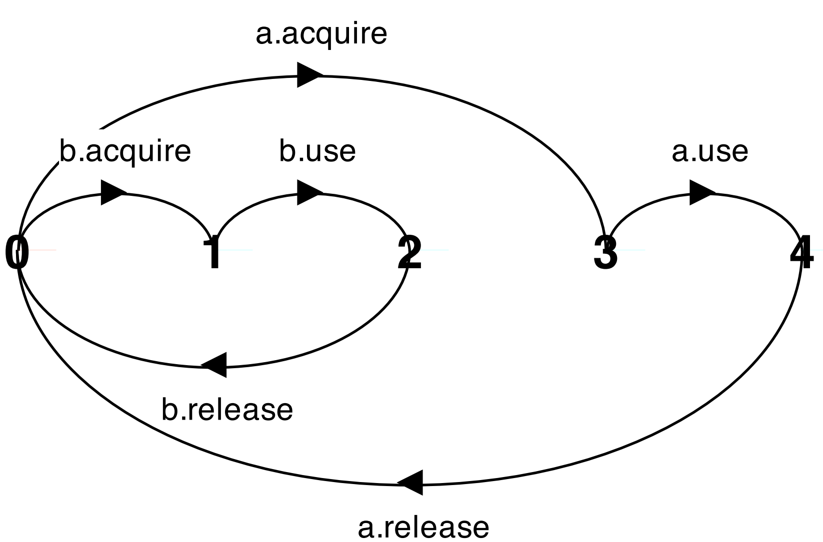
b:USER



{a,b}::RESOURCE



RESOURCE_SHARE



How does the model ensure that the user that acquires the resource is the one to release it?

Example

```
X  =  (x -> STOP).
```

Example

```
X  =  (x -> STOP).
```

```
||SYS_1 = {a,b}:X.
```

```
||SYS_2 = {a,b}::X.
```

Example

```
X  =  (x -> STOP).
```

```
||SYS_1 = {a,b}:X.
```

LTS? Traces? Number of states?

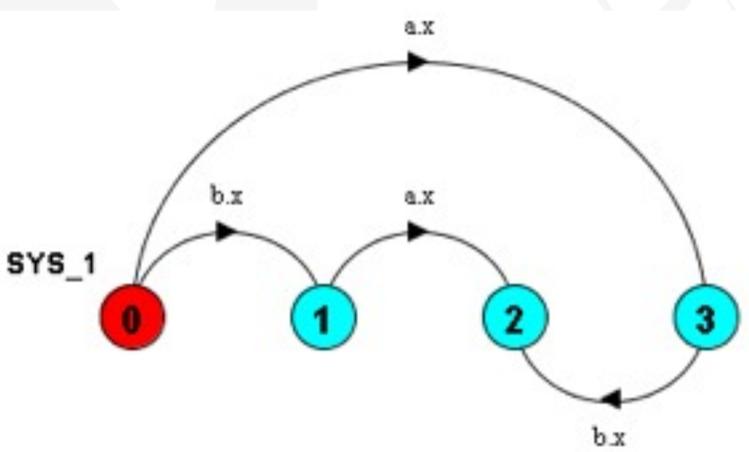
```
||SYS_2 = {a,b}::X.
```

Example


```
X   =   (x -> STOP).
```

```
||SYS_1 = {a,b}:X.
```

LTS? Traces? Number of states?



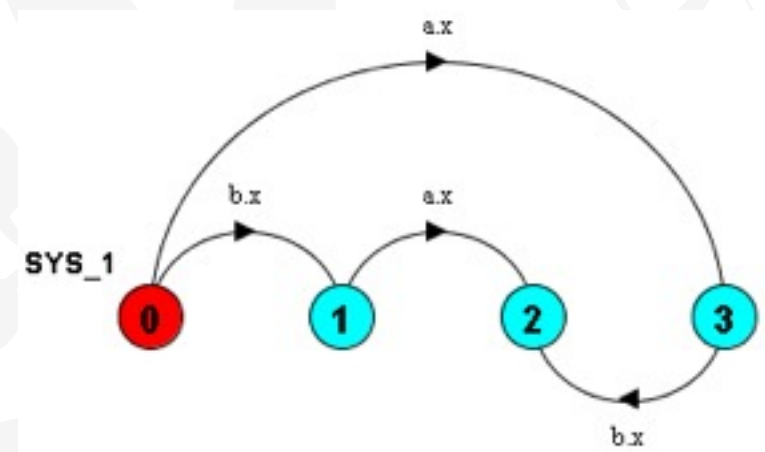

```
||SYS_2 = {a,b}::X.
```

Example

```
X   =   (x -> STOP).
```

```
||SYS_1 = {a,b}:X.
```

LTS?   Traces?   Number of states?




```
||SYS_2 = {a,b}::X.
```

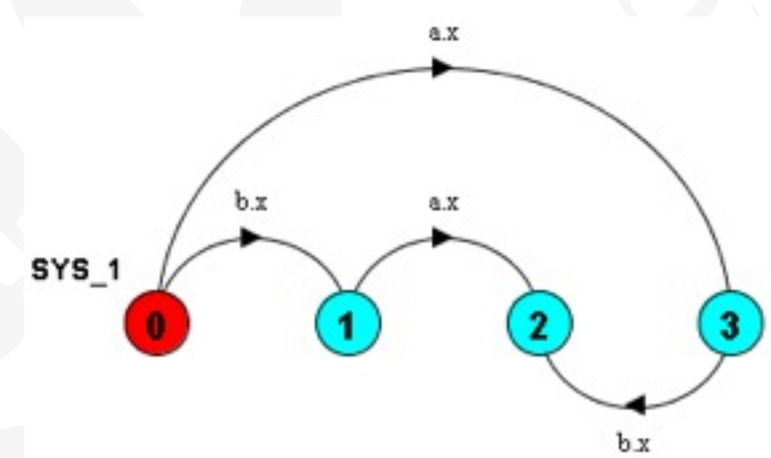
LTS?   Traces?   Number of states?

Example


```
X = (x -> STOP).
```

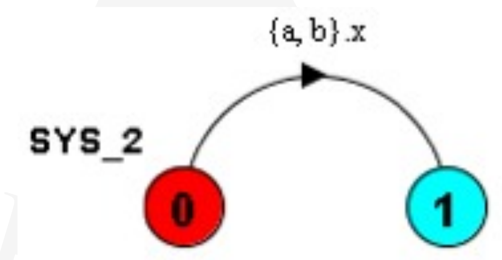
```
||SYS_1 = {a,b}:X.
```

LTS? Traces? Number of states?




```
||SYS_2 = {a,b}::X.
```

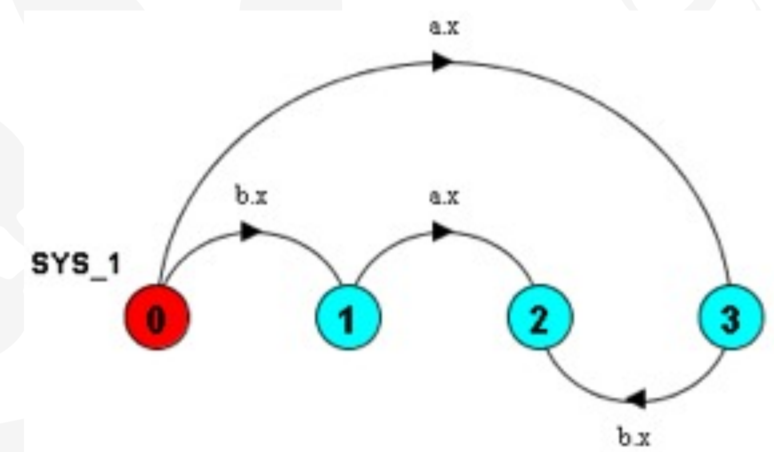
LTS? Traces? Number of states?



Example


```
X = (x -> STOP).
```

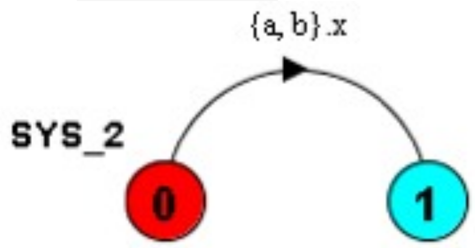
```
||SYS_1 = {a,b}:X.
```



LTS? Traces? Number of states?

{a...}:X creates one process per prefix

```
||SYS_2 = {a,b}::X.
```



LTS? Traces? Number of states?

{a...}::X creates one process with all prefixes

Action Relabelling

Relabelling functions are applied to processes to change the names of action labels. The general form of the relabelling function is:

$$/\{newlabel_1/oldlabel_1, \ldots\ newlabel_n/oldlabel_n\}.$$

Action Relabelling

Relabelling functions are applied to processes to change the names of action labels. The general form of the relabelling function is:

$$/\{\mathbf{newlabel_1/oldlabel_1, \dots\ newlabel_n/oldlabel_n}\}.$$

Relabelling to ensure that composed processes synchronise on particular actions:

Action Relabelling

Relabelling functions are applied to processes to change the names of action labels. The general form of the relabelling function is:

$$/\{newlabel_1/oldlabel_1, \dots\ newlabel_n/oldlabel_n\}.$$

Relabelling to ensure that composed processes synchronise on particular actions:

```
CLIENT = (call->wait->continue->CLIENT).
```

Action Relabelling

Relabelling functions are applied to processes to change the names of action labels. The general form of the relabelling function is:

$/\{newlabel_1/oldlabel_1, \ldots\ newlabel_n/oldlabel_n\}$.

Relabelling to ensure that composed processes synchronise on particular actions:

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

Action Relabelling

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

Action Relabelling

University of Southern Denmark

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

Action Relabelling

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

```
S = (SERVER /{call/request}).
```

Action Relabelling

University of Southern Denmark

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

```
S = (SERVER /{call/request}).
```

```
||C_S = (C || S).
```

Action Relabelling

```
CLIENT = (call->wait->continue->CLIENT).
```

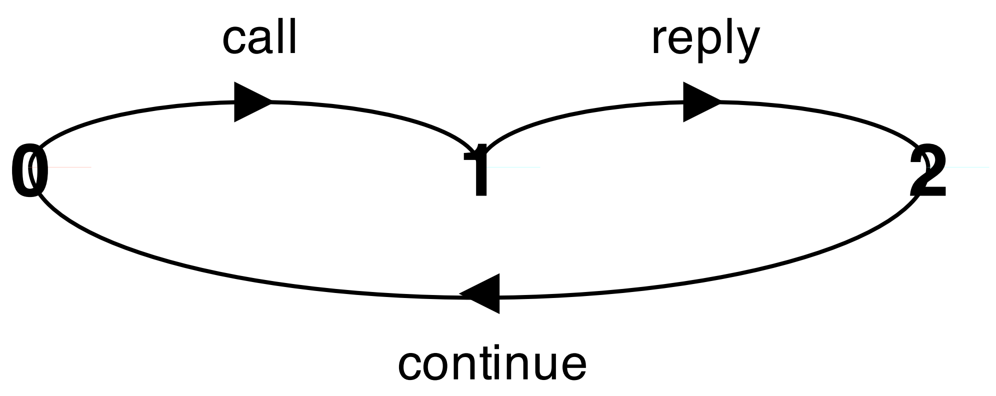
```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

```
S = (SERVER /{call/request}).
```

```
||C_S = (C || S).
```

C



Action Relabelling

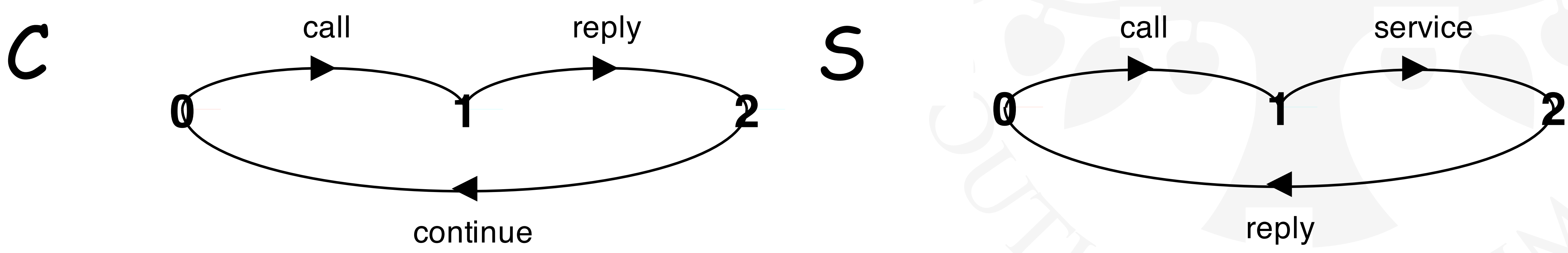

```
CLIENT = (call->wait->continue->CLIENT).
```

```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

```
S = (SERVER /{call/request}).
```

```
||C_S = (C || S).
```



Action Relabelling

UNIVERSITY OF SOUTHERN DENMARK

```
CLIENT = (call->wait->continue->CLIENT).
```

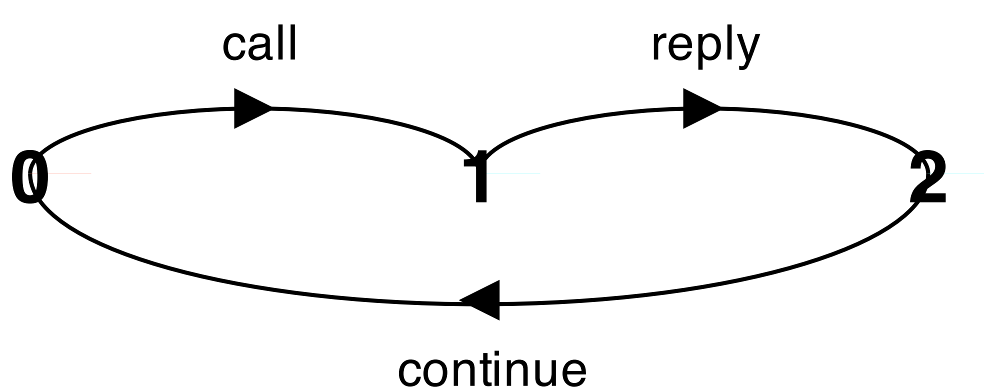
```
SERVER = (request->service->reply->SERVER).
```

```
C = (CLIENT /{reply/wait}).
```

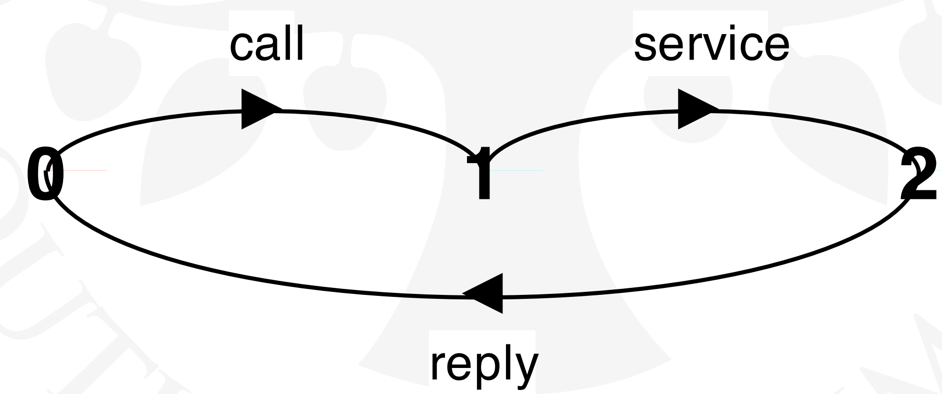
```
S = (SERVER /{call/request}).
```

```
||C_S = (C || S).
```

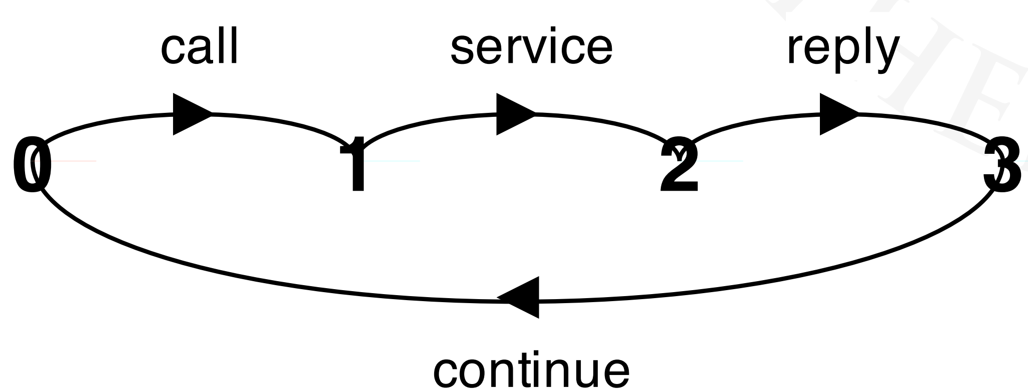
C



S



C_S



Action Relabelling - Prefix Labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

Action Relabelling - Prefix Labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
             ->service->accept.reply->SERVERv2).
```

Action Relabelling - Prefix Labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
             ->service->accept.reply->SERVERv2).

CLIENTv2 = (call.request
             ->call.reply->continue->CLIENTv2).
```

Action Relabelling - Prefix Labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
             ->service->accept.reply->SERVERv2).

CLIENTv2 = (call.request
             ->call.reply->continue->CLIENTv2).


||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                     /{call/accept}.
```

Action Hiding - Abstraction To Reduce Complexity

When applied to a process P, the hiding operator **\\{a_1,...,a_x}** removes the action names a_1..a_x from the alphabet of P and makes these concealed actions "silent".

These silent actions are labelled tau.

Silent actions in different processes are not shared.

Action Hiding - Abstraction To Reduce Complexity

When applied to a process P, the hiding operator **\{a_1,..,a_x}** removes the action names $a_1..a_x$ from the alphabet of P and makes these concealed actions "silent".

These silent actions are labelled tau.

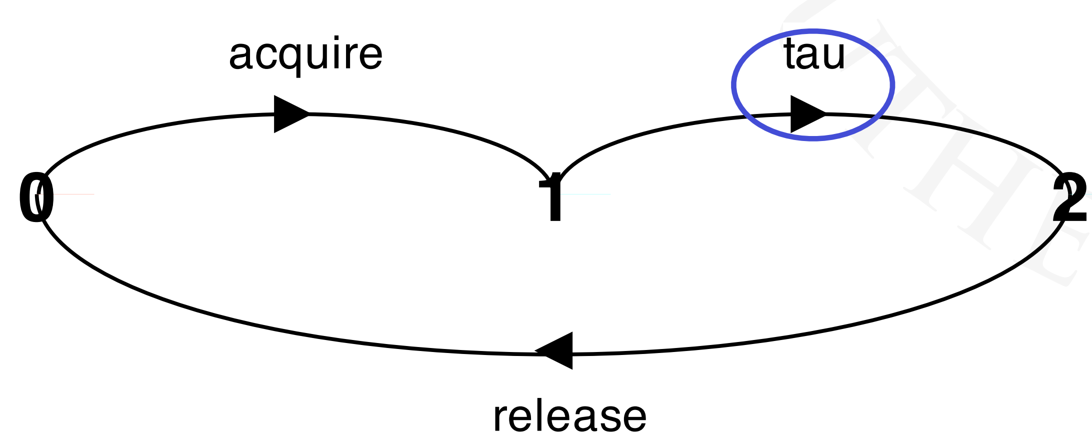
Silent actions in different processes are not shared.

```
USER = (acquire->use->release->USER)
                                     \{use}.
```

Action Hiding - Abstraction To Reduce Complexity

When applied to a process P, the hiding operator **\{a1,..,ax}** removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent".
These silent actions are labelled tau.
Silent actions in different processes are not shared.

```
USER = (acquire->use->release->USER)
                                       \{use}.
```



Action Hiding - Abstraction To Reduce Complexity

Sometimes it is more convenient to specify the set of labels to be exposed....

> When applied to a process P, the interface operator **@{a_1,...,a_x}** hides all actions in the alphabet of P not labelled in the set $a_1..a_x$.

Action Hiding - Abstraction To Reduce Complexity

Sometimes it is more convenient to specify the set of labels to be exposed....

When applied to a process P, the interface operator **@{a_1,..,a_x}** hides all actions in the alphabet of P not labelled in the set $a_1..a_x$.

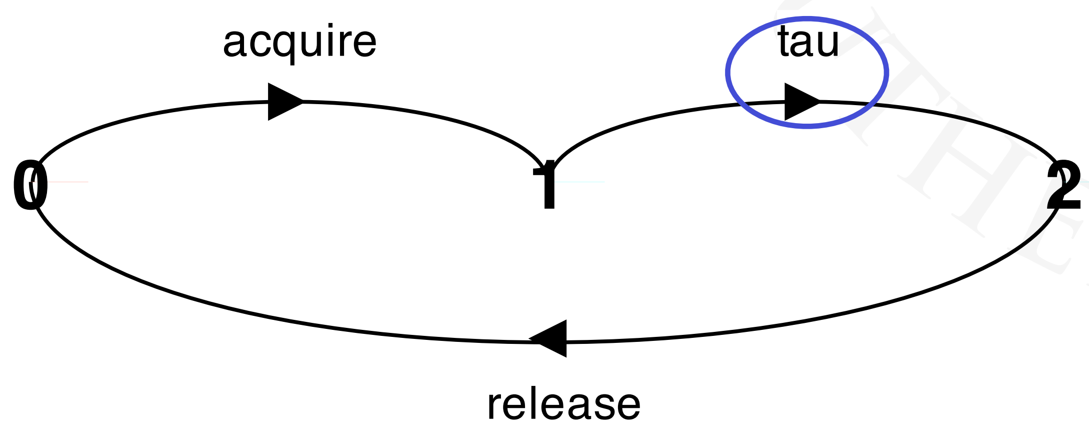
```
USER = (acquire->use->release->USER)
                          @{acquire,release}.
```

Action Hiding - Abstraction To Reduce Complexity

Sometimes it is more convenient to specify the set of labels to be exposed....

> When applied to a process P, the interface operator **@{a_1,..,a_x}** hides all actions in the alphabet of P not labelled in the set $a_1..a_x$.

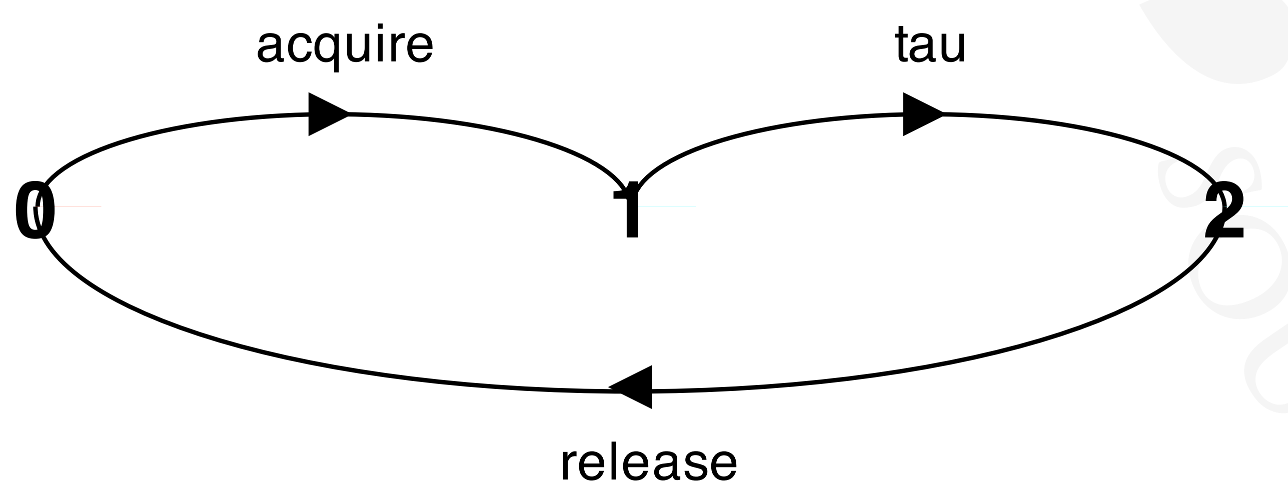
```
USER = (acquire->use->release->USER)
                          @{acquire,release}.
```



Action Hiding

The following definitions are equivalent:

```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
          @{acquire,release}.
```

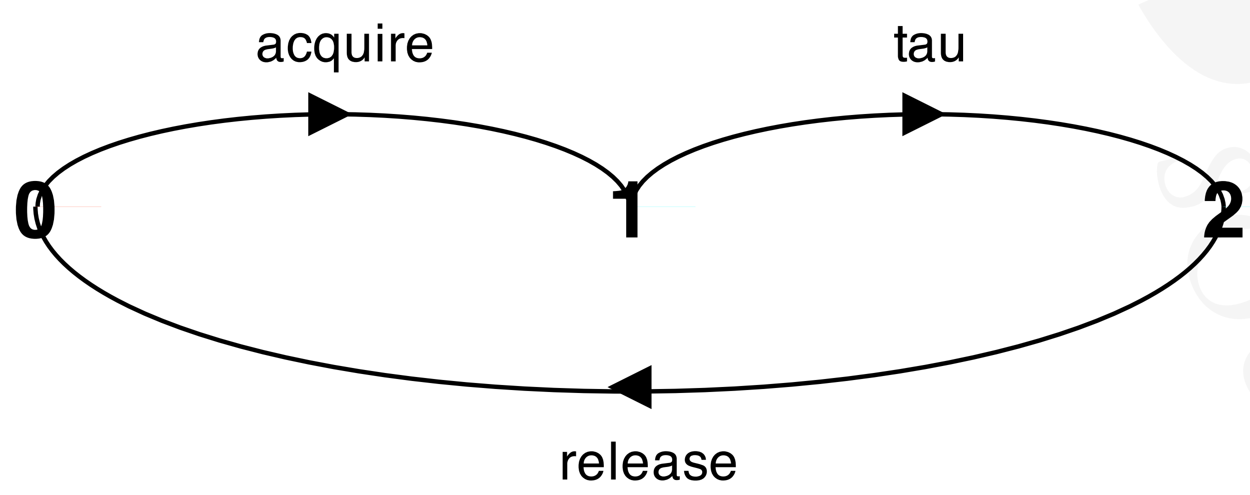


Action Hiding

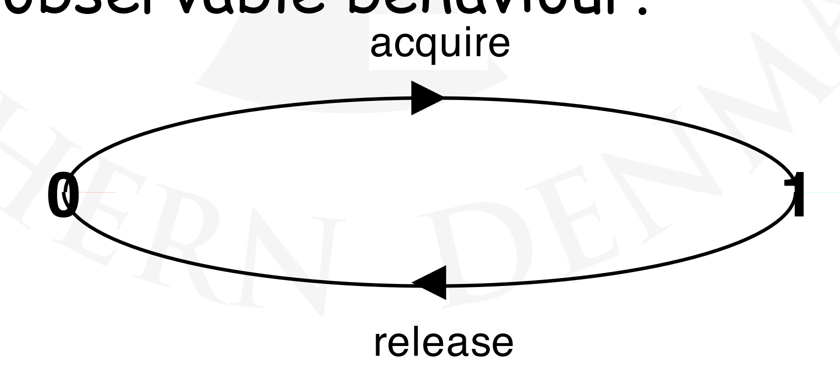
UNIVERSITY OF SOUTHERN DENMARK

The following definitions are equivalent:

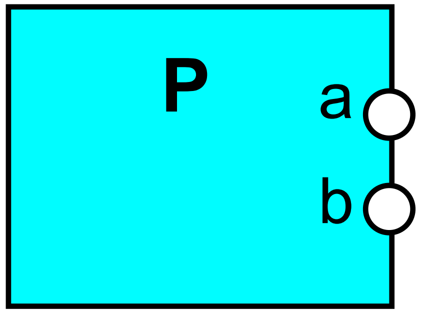
```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
          @{acquire,release}.
```



Minimisation removes hidden **tau** actions to produce an LTS with equivalent observable behaviour.

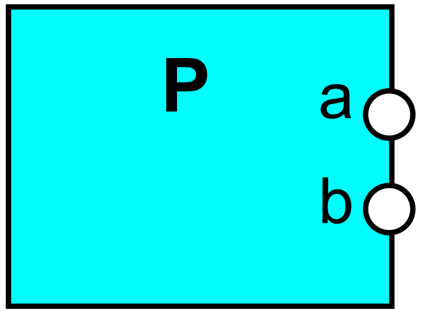


Structure Diagrams



Process P with alphabet {a,b}.

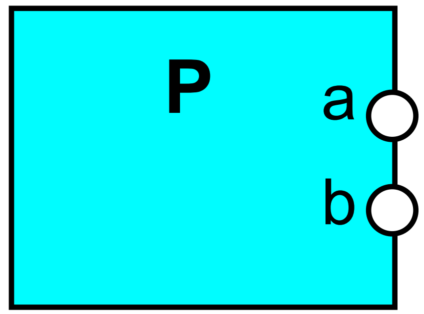
Structure Diagrams



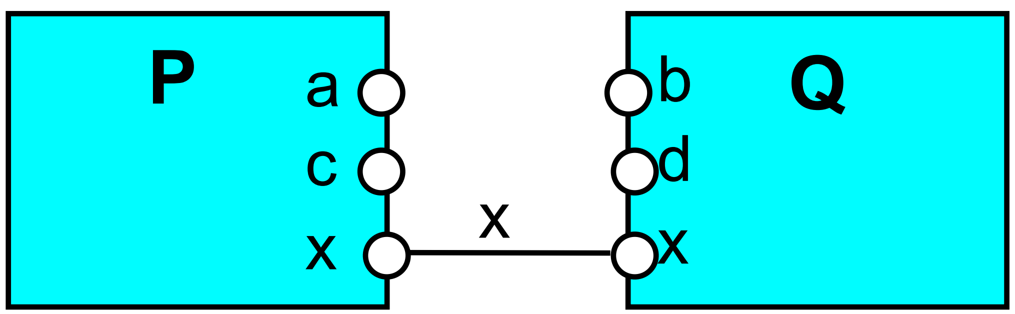
Process P with alphabet {a,b}.

Parallel Composition (P||Q)

Structure Diagrams

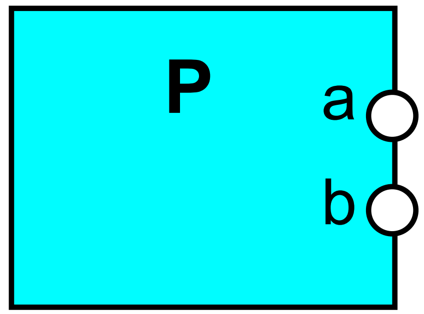


Process P with alphabet {a,b}.

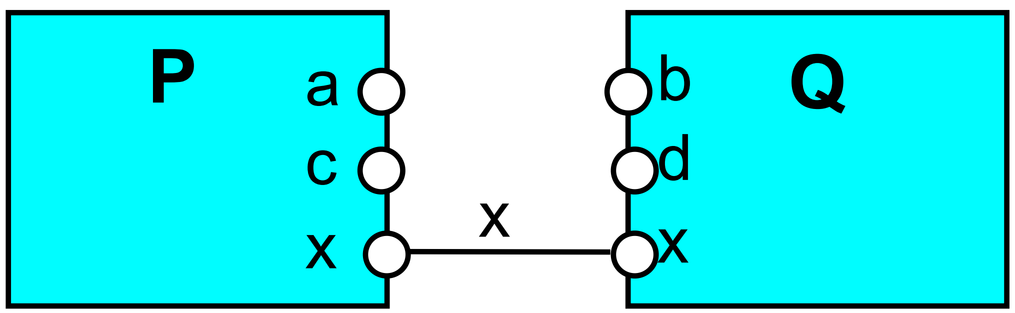


Parallel Composition (P||Q)

Structure Diagrams

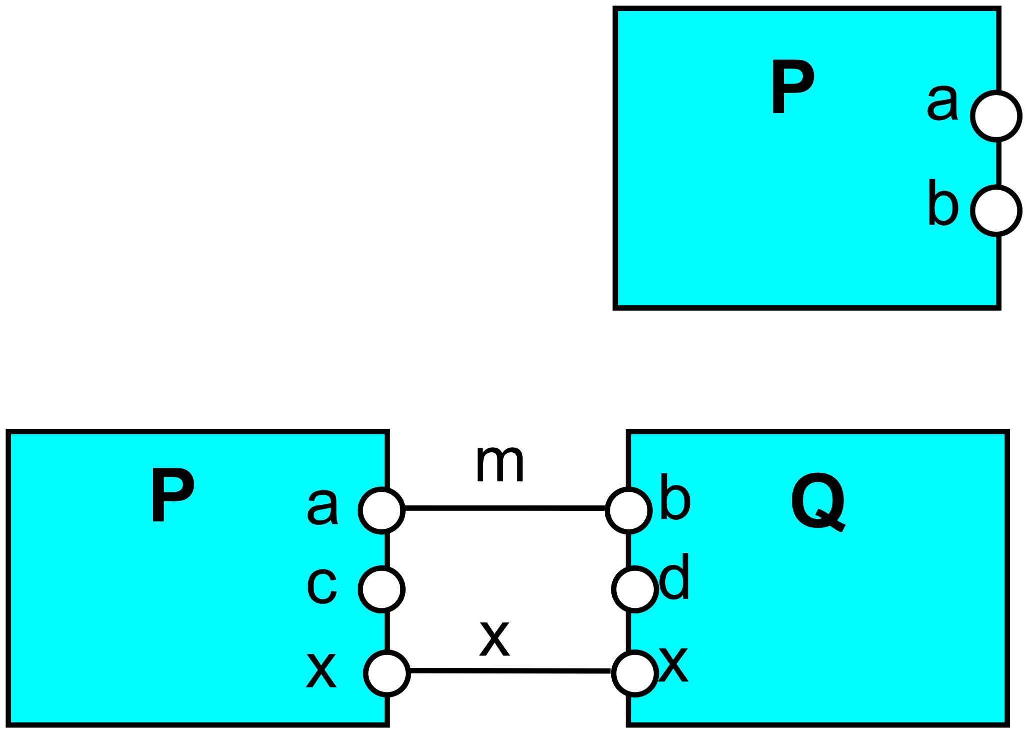


Process P with alphabet {a,b}.



Parallel Composition (P||Q) / {m/a,m/b }

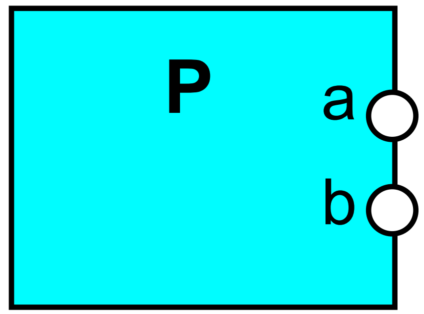
Structure Diagrams



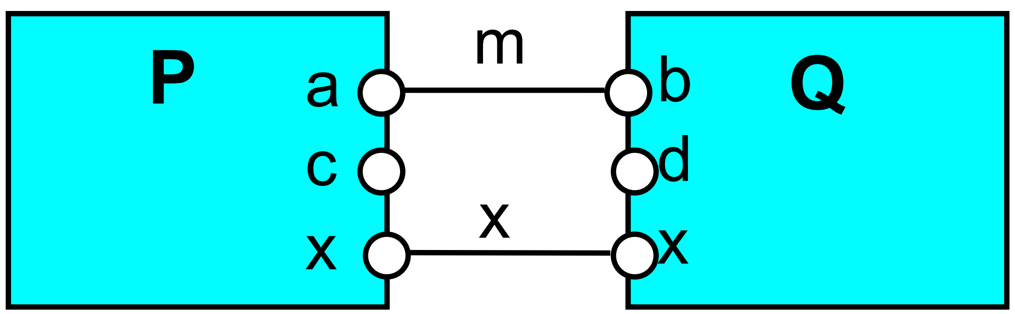
Process P with alphabet {a,b}.

Parallel Composition (P||Q) / {m/a,m/b }

Structure Diagrams

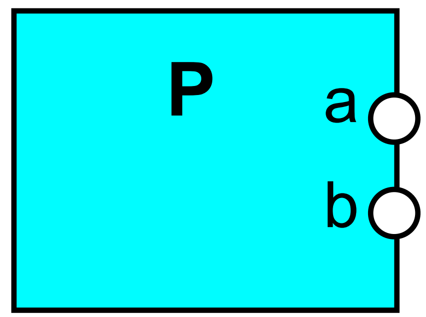


Process P with
alphabet {a,b}.

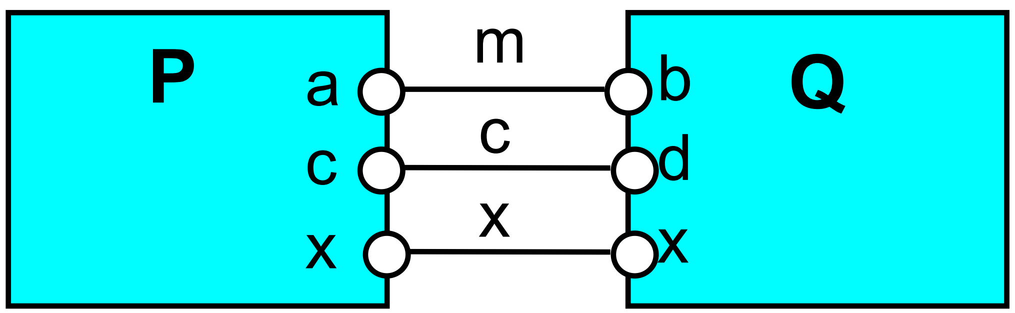


Parallel Composition
(P||Q) / {m/a,m/b,c/d}

Structure Diagrams

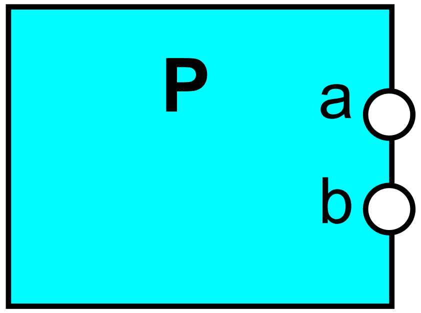


Process P with alphabet {a,b}.

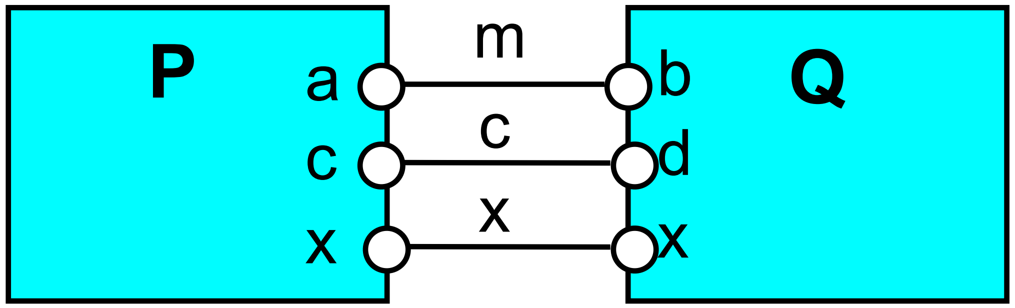


Parallel Composition $(P||Q) / \{m/a, m/b, c/d\}$

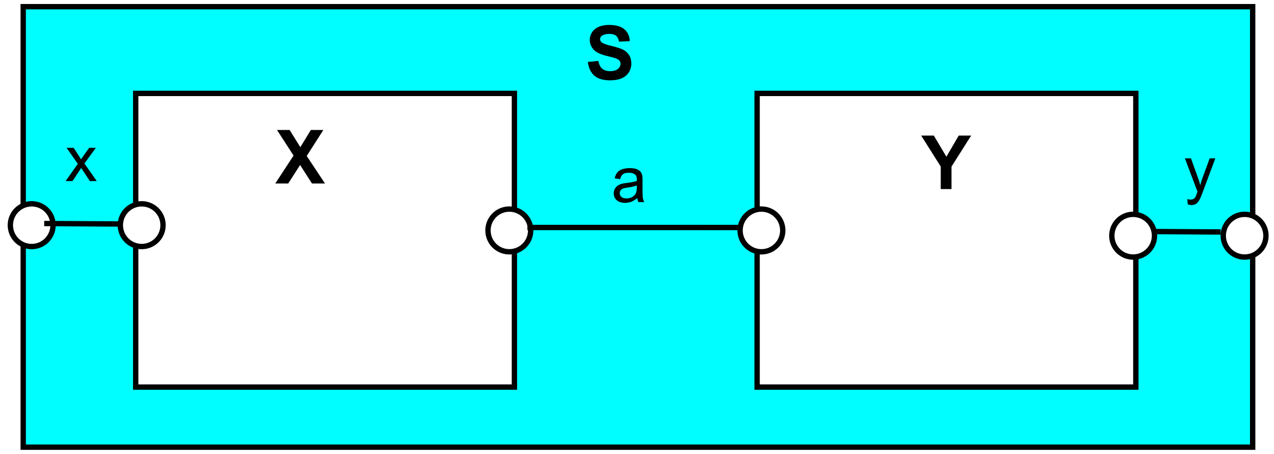
Structure Diagrams



Process P with alphabet {a,b}.



Parallel Composition
(P||Q) / {m/a,m/b,c/d}



Composite process
||S = (X||Y) @ {x,y}

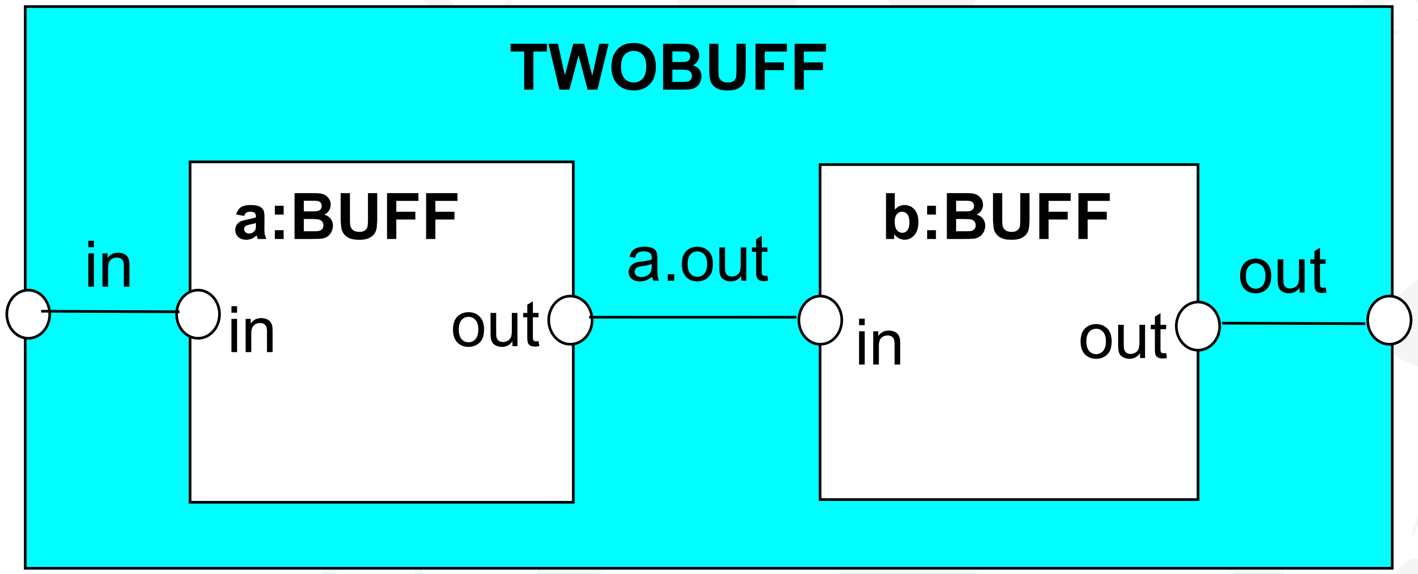
Structure Diagrams

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

Structure Diagrams

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

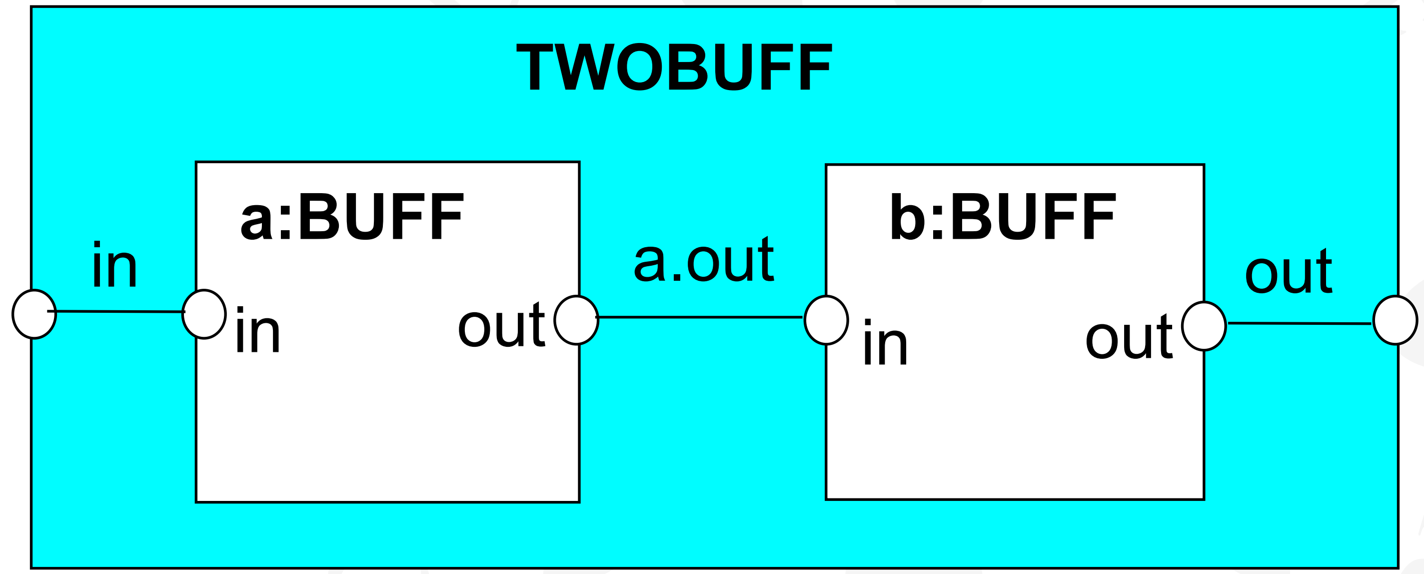
We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabelling and hiding.



Structure Diagrams


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

We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabelling and hiding.

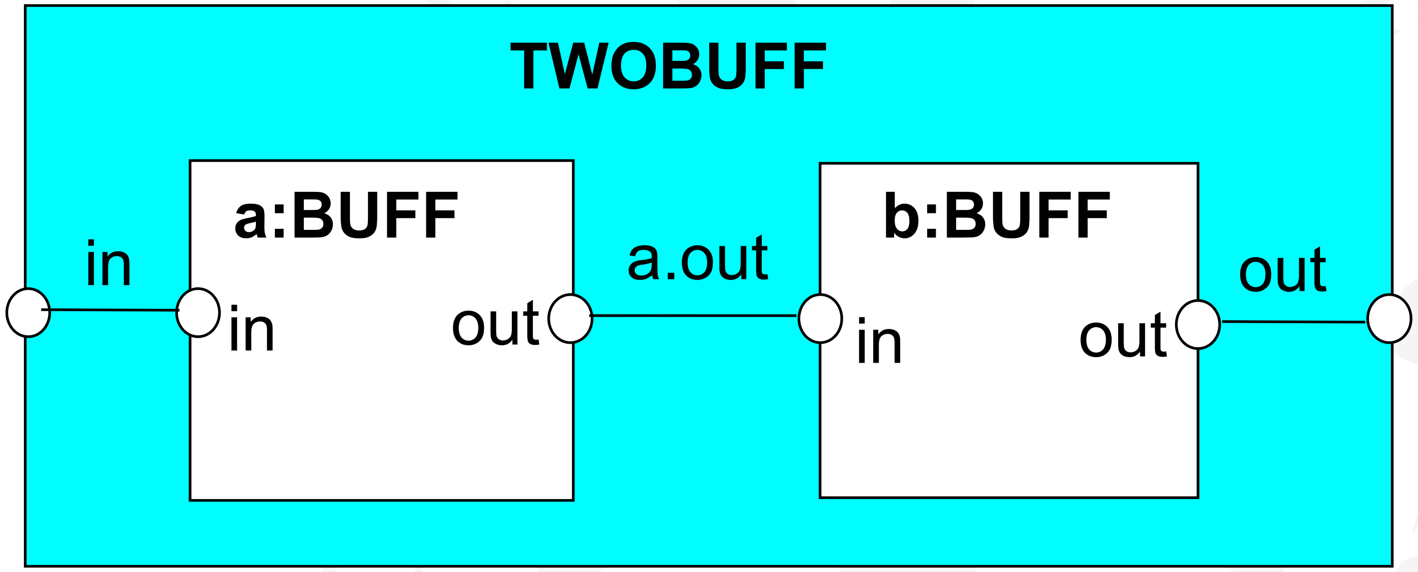



```
||TWOBUF =
```

Structure Diagrams


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

We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabelling and hiding.




```
||TWOBUF = (a:BUFF || b:BUFF)
              /{in/a.in, a.out/b.in, out/b.out}
                 @{in,out}.
```

Structure Diagrams

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT||SERVER)
                  /{reply/wait,
                    call/request}.
```

Structure Diagrams

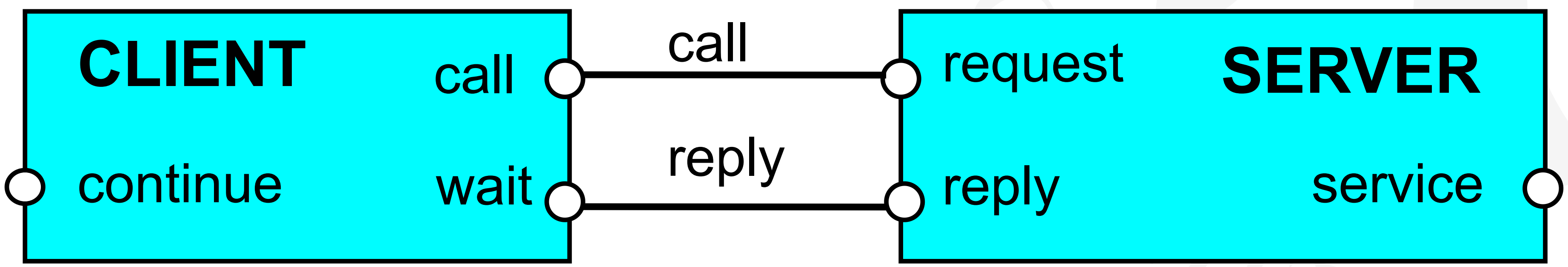
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT||SERVER)
                      /{reply/wait,
                        call/request}.
```

Structure diagram for CLIENT_SERVER ?

Structure Diagrams

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT||SERVER)
                    /{reply/wait,
                      call/request}.
```

Structure diagram for `CLIENT_SERVER` ?



Structure Diagrams

```
SERVERv2 = (accept.request
            ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                    /{call/accept}.
```

Structure Diagrams

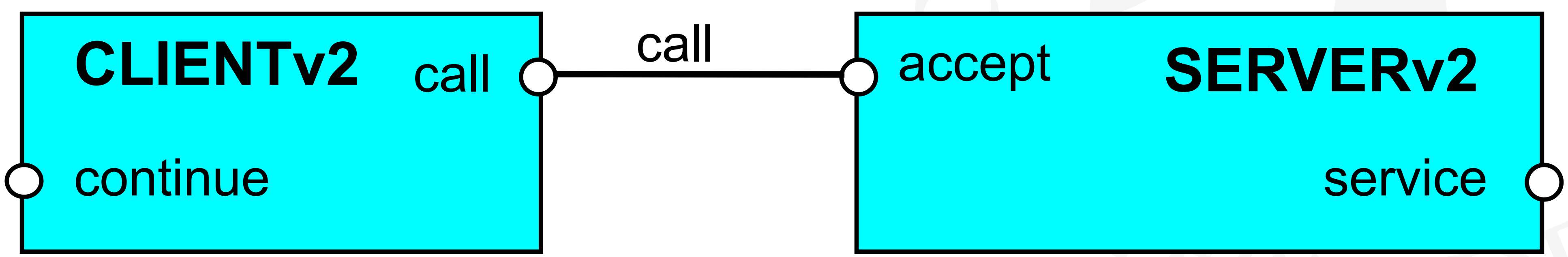

```
SERVERv2 = (accept.request
             ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
             ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                     /{call/accept}.
```

Structure diagram for `CLIENT_SERVERv2` ?

Structure Diagrams

```
SERVERv2 = (accept.request
              ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
              ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
                      /{call/accept}.
```

Structure diagram for **CLIENT_SERVERv2** ?



Simply use the shared prefix.

Structure Diagrams - Resource Sharing

Structure Diagrams - Resource Sharing

```
RESOURCE = (acquire->release->RESOURCE).
```

Structure Diagrams - Resource Sharing

```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

Structure Diagrams - Resource Sharing

```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

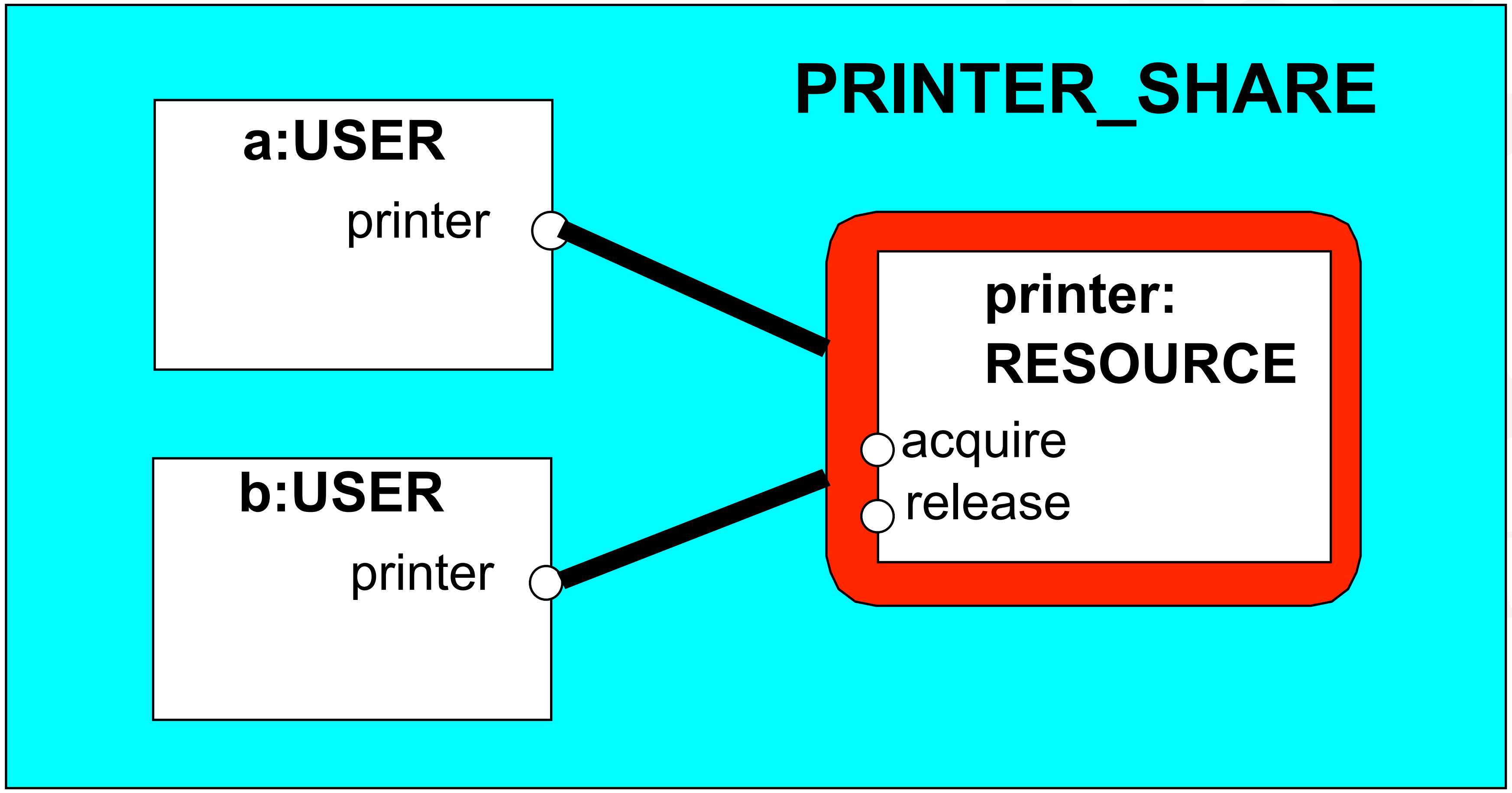
```
||PRINTER_SHARE =
    (a:USER || b:USER || {a,b}::printer:RESOURCE).
```

Structure Diagrams - Resource Sharing

```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

```
||PRINTER_SHARE =
    (a:USER || b:USER || {a,b}::printer:RESOURCE).
```

Shared resources are shown as “rounded rectangles”:



Java

ThreadDemo Model

ThreadDemo Model

```
THREAD = OFF,
OFF = (toggle->ON
```

ThreadDemo Model

```
THREAD = OFF,
OFF = (toggle->ON
      |abort->STOP),
ON  = (toggle->OFF
      |output->ON
      |abort->STOP).
```

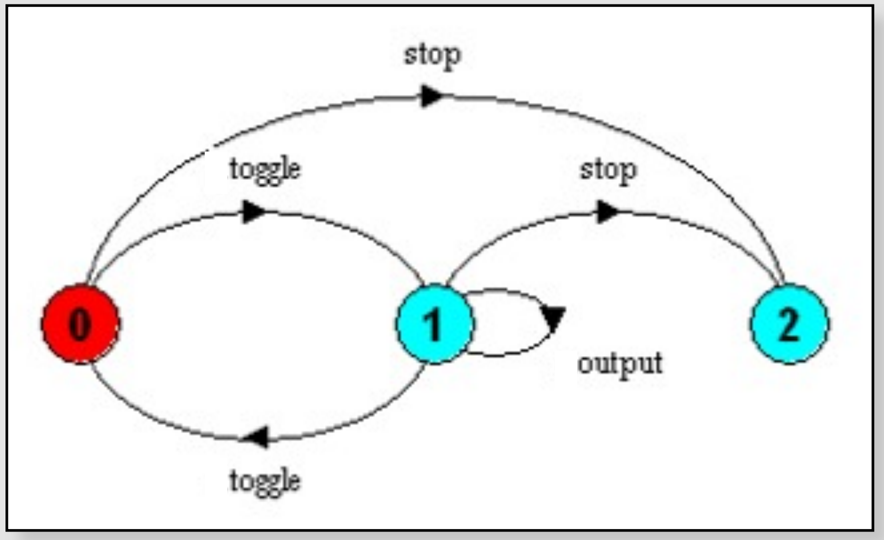
ThreadDemo Model

UNIVERSITY OF SOUTHERN DENMARK

```
THREAD = OFF,

OFF = (toggle->ON
      |abort->STOP),

ON  = (toggle->OFF
      |output->ON
      |abort->STOP).


||THREAD_DEMO =
      (a:THREAD || b:THREAD)
         /{stop/{a,b}.abort}.
```

ThreadDemo Model

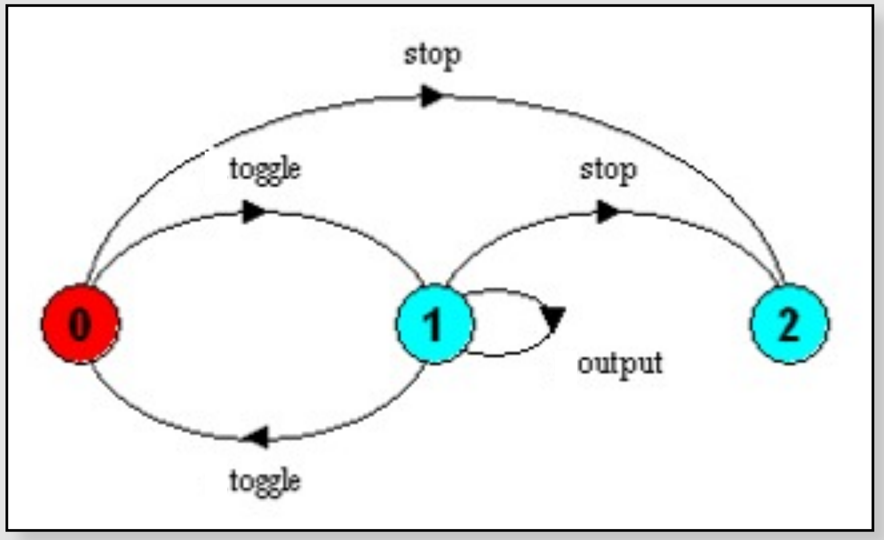

```
THREAD = OFF,
OFF = (toggle->ON
      |abort->STOP),
ON  = (toggle->OFF
      |output->ON
      |abort->STOP).

||THREAD_DEMO =
      (a:THREAD || b:THREAD)
         /{stop/{a,b}.abort}.
```



ThreadDemo Model

University of Southern Denmark

```
THREAD = OFF,

OFF = (toggle->ON
      |abort->STOP),

ON  = (toggle->OFF
      |output->ON
      |abort->STOP).


||THREAD_DEMO =
      (a:THREAD || b:THREAD)
         /{stop/{a,b}.abort}.
```



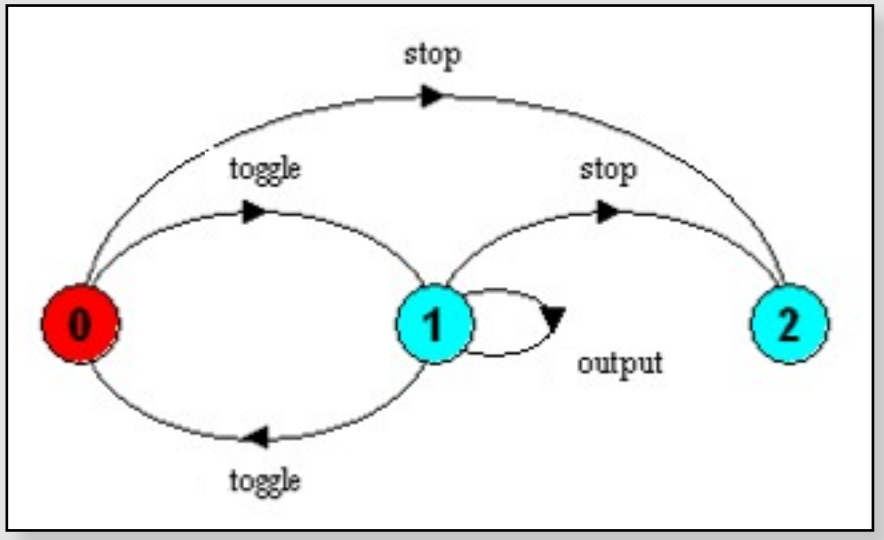
Interpret:

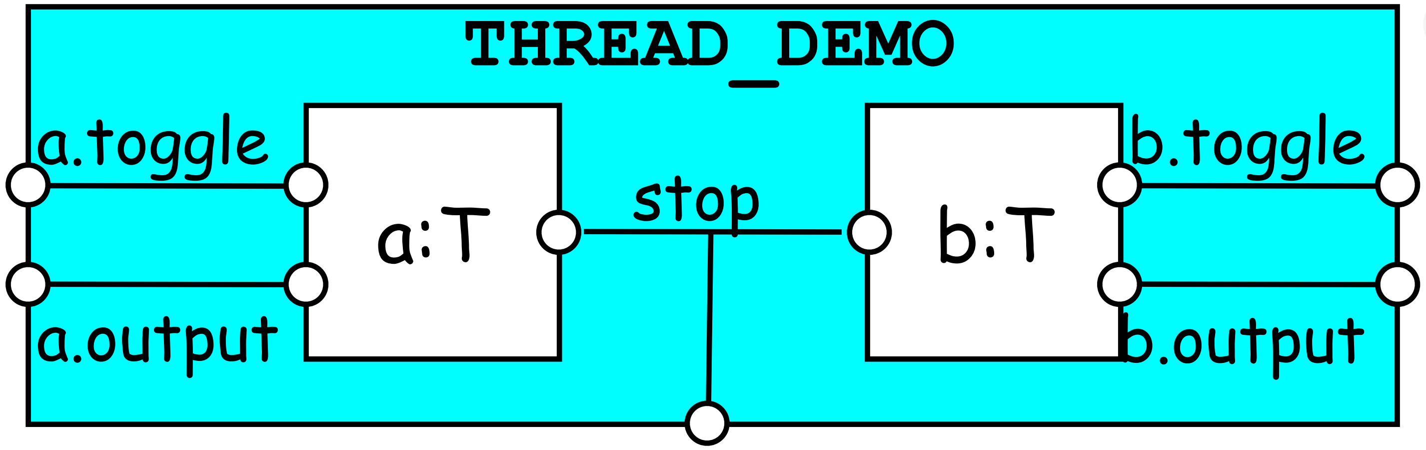
toggle, *abort*
as inputs;

output
as output

ThreadDemo Model


```
THREAD = OFF,
OFF = (toggle->ON
      |abort->STOP),
ON  = (toggle->OFF
      |output->ON
      |abort->STOP).

||THREAD_DEMO =
      (a:THREAD || b:THREAD)
         /{stop/{a,b}.abort}.
```



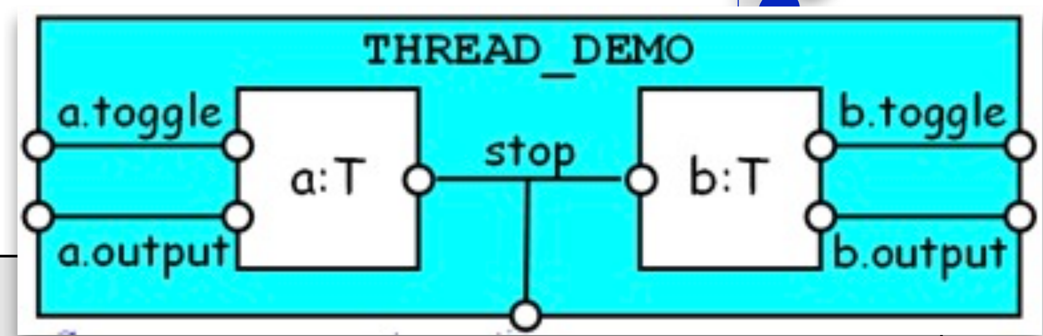


Interpret:

toggle, *abort* as inputs;

output as output

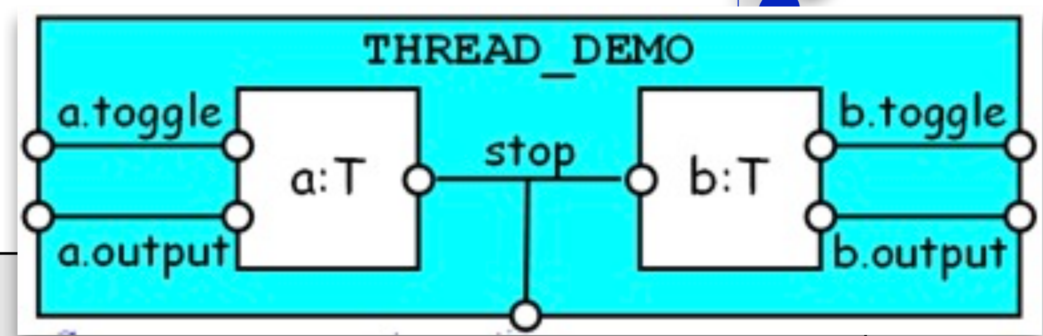
ThreadDemo Code: MyThread




```
class MyThread extends Thread {
    private boolean on;

    MyThread(String name) { super(name); this.on = false; }

    public void toggle() {   on = !on; }

    public void abort() { this.interrupt(); }

    private void output() {
                 System.out.println(getName()+": output");
    }
    public void run() {
        try {
            while (!interrupted()) {
                if (on) output();
                sleep(500);
            }
        } catch(Int'Exc' _) {}
        System.out.println("Done!");
}}}
```

```
THREAD = OFF,
OFF = (toggle->ON
      |abort->STOP),
ON  = (toggle->OFF
      |output->ON
      |abort->STOP).

||THREAD_DEMO =
      (a:THREAD || b:THREAD)
        /{stop/{a,b}.abort}.
```

ThreadDemo Code: ThreadDemo




```
class ThreadDemo {
    public static void main(String[] args) {
        MyThread a = new MyThread("a");
        MyThread b = new MyThread("b");
        a.start(); b.start();
        while (true) {
            switch (readChar()) {
                case 'a': a.toggle();
                          break;
                case 'b': b.toggle();
                          break;
                case 'i': stop(a,b);
                          return;
            }
        }
    }
    private stop(MyThread a, MyThread b) {
        a.abort();
        b.abort();
    }
}
```

```
THREAD = OFF,

OFF = (toggle->ON
      |abort->STOP),

ON  = (toggle->OFF
      |output->ON
      |abort->STOP).

||THREAD_DEMO =
      (a:THREAD || b:THREAD)
        /{stop/{a,b}.abort}.
```

Summary

Summary

- Concepts

Summary

- Concepts
  - Concurrent processes and process interaction

Summary

- Concepts
  - Concurrent processes and process interaction
- Models

Summary

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).

Summary

- **Concepts**
  - Concurrent processes and process interaction
- **Models**
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.

Summary

UNIVERSITY OF SOUTHERN DENMARK

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.

Summary

UNIVERSITY OF SOUTHERN DENMARK

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labelling and action relabelling and hiding.

Summary

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labelling and action relabelling and hiding.
  - Structure diagrams

Summary

University of Southern Denmark

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labelling and action relabelling and hiding.
  - Structure diagrams
- Practice

Summary

- Concepts
  - Concurrent processes and process interaction
- Models
  - Asynchronous (arbitrary speed) & interleaving (arbitrary order).
  - Parallel composition as a finite state process with action interleaving.
  - Process interaction by shared actions.
  - Process labelling and action relabelling and hiding.
  - Structure diagrams
- Practice
  - Multiple threads in Java.