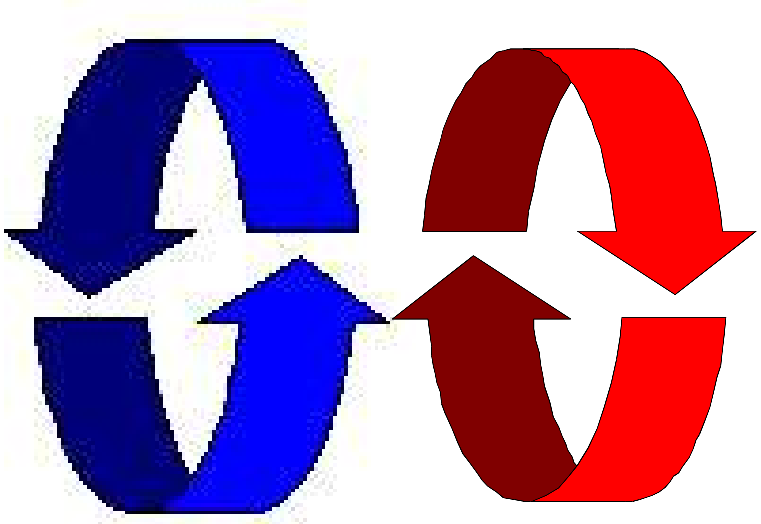
3 – Concurrent Execution



Alexandre David

adavid@cs.aau.dk

Credits for the slides:
Claus Brabrand
Jeff Magee & Jeff Kramer

Repetition

- **Concepts:** We adopt a model-based approach for the design and construction of concurrent programs.
- **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 (and later C).

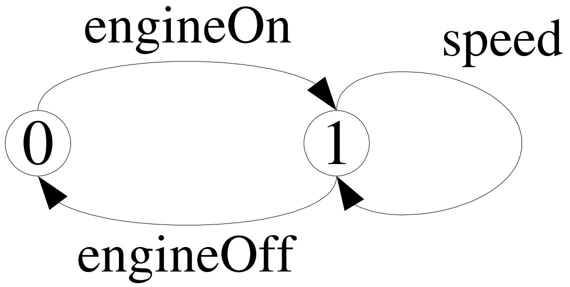
Repetition

Model = simplified representation of the real world.

- Based on Labelled Transition Systems (LTS)

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

Aka. ***Finite State Machine*** *(FSM)*



- Described textually as Finite State Processes (FSP)

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

Repetition

- 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[i:1..N] = …*
◆ *constant definitions*	*const N = 3*
◆ *range definitions*	*range R = 0..N*

Repetition

- 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();
// ...
```

Concurrent Execution

Concepts: *processes - concurrent execution and interleaving.*

process interaction.

Models: **parallel composition**

of asynchronous processes - interleaving

interaction

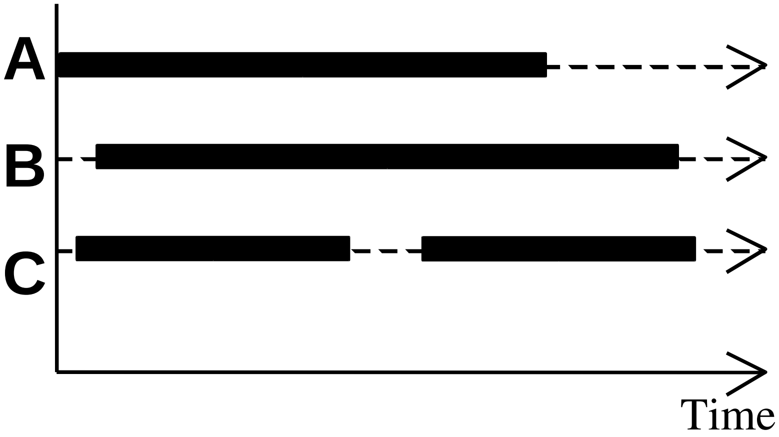
shared actions, process labelling, and action relabelling and hiding

structure diagrams

Practice: *multi-threaded Java programs*

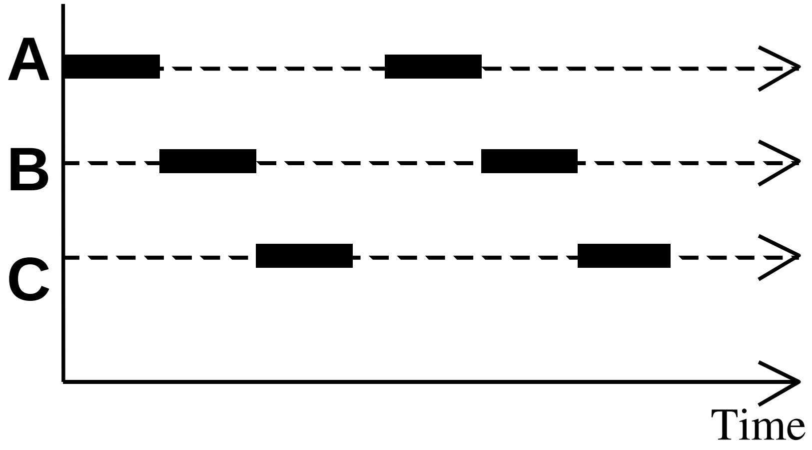
Definition: Parallelism

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



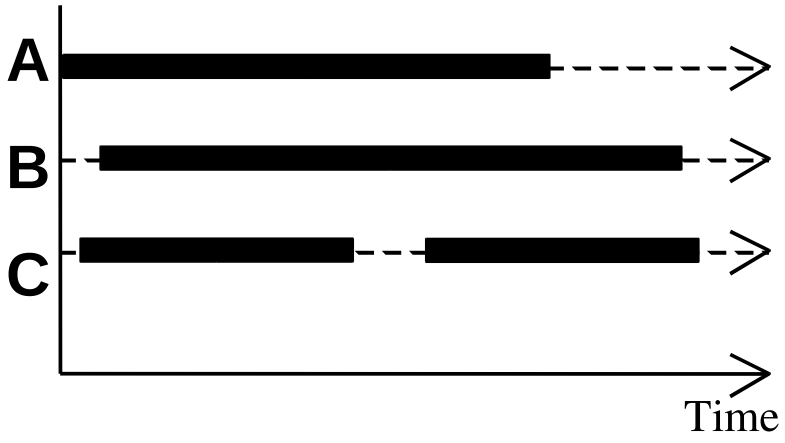
Definition: Concurrency

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

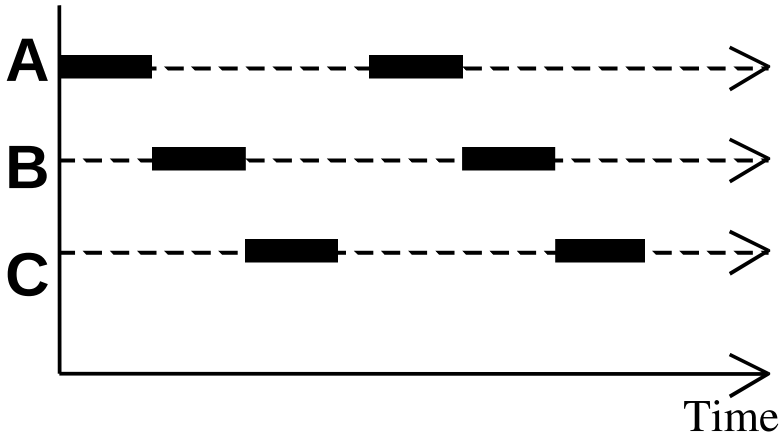


Parallelism vs. Concurrency

Parallelism



Concurrency

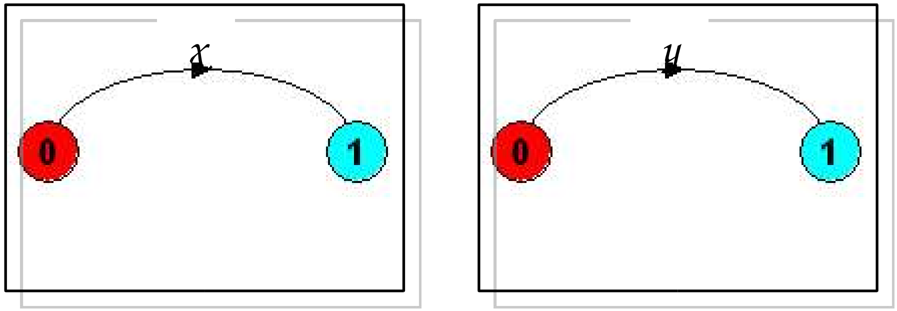


Both **concurrency** *and* **parallelism** *require controlled access to shared resources.*

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

Modeling Concurrency

- How do we model concurrency?



Possible execution sequences?

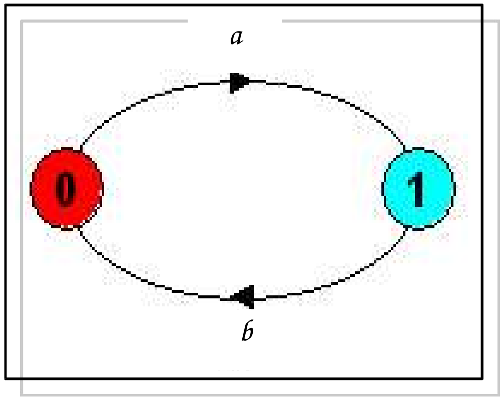
- $x ; y$
- $y ; x$
- ~~$x \mid\mid y$~~

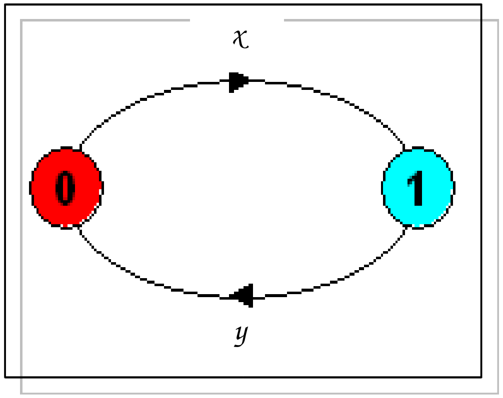
Asynchronous model of execution

- Arbitrary relative order of actions from different processes – ***interleaving*** but preservation of each process order.

Modeling Concurrency

- How should we model process execution speed?





- We 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.*

```
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**

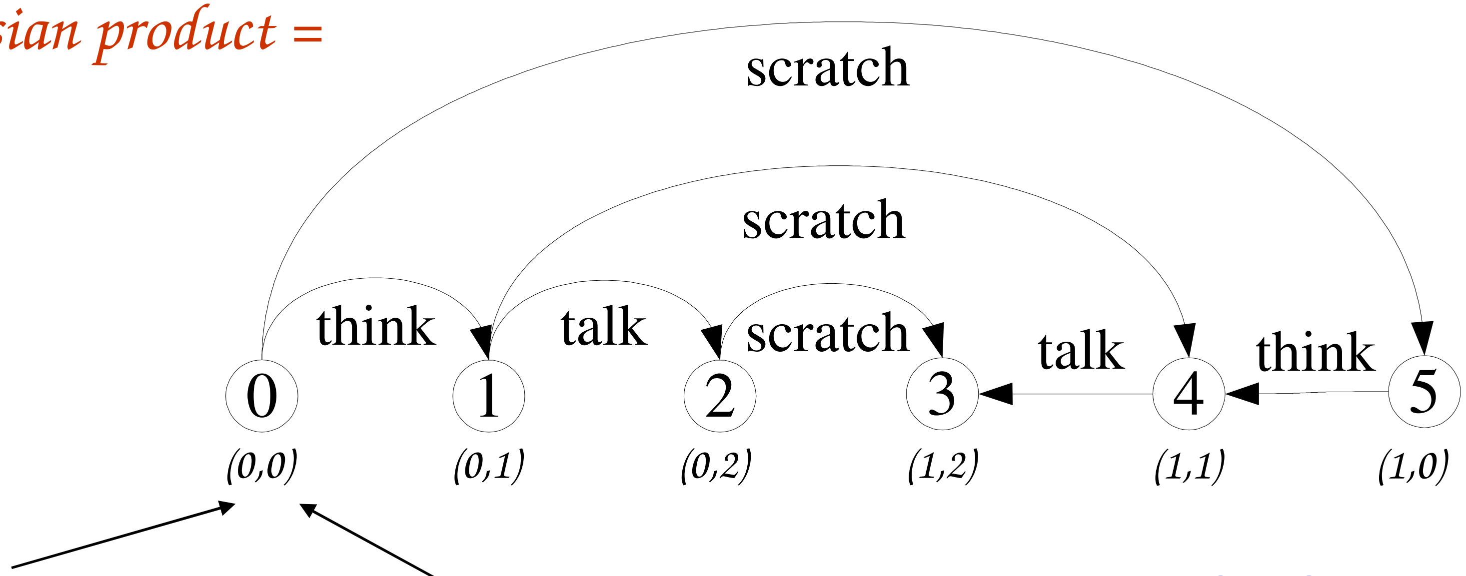
Parallel Composition – Action Interleaving

Parallel composition =

2 states

3 states

Cartesian product =



from ITCH

from CONVERSE

2 x 3 states

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 Action

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

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

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

LTS? Traces? Number of states?

Shared Action:

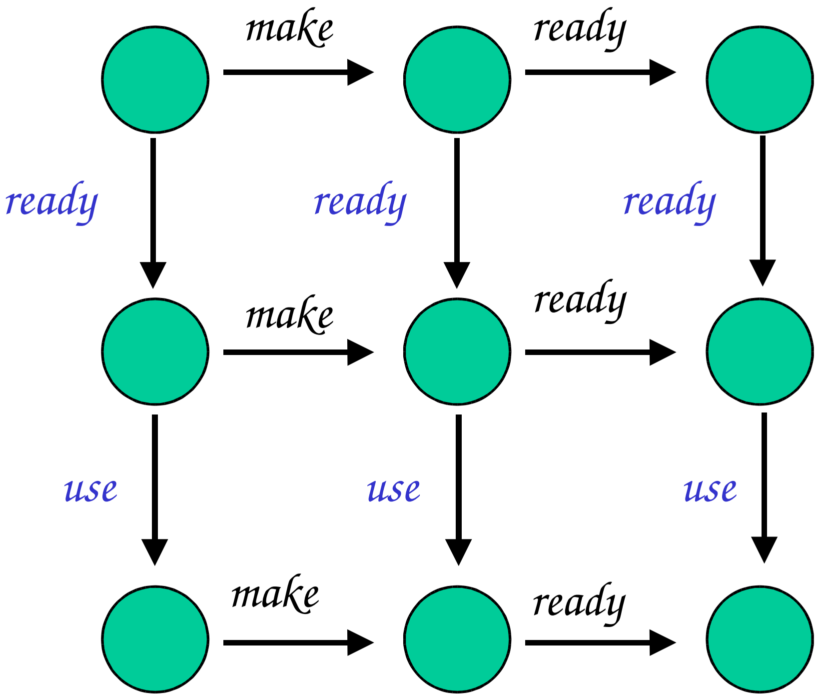
If processes in a composition have actions in common, these actions are said to be shared. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the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

Modeling Interaction - Example

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

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

3 states

3 states



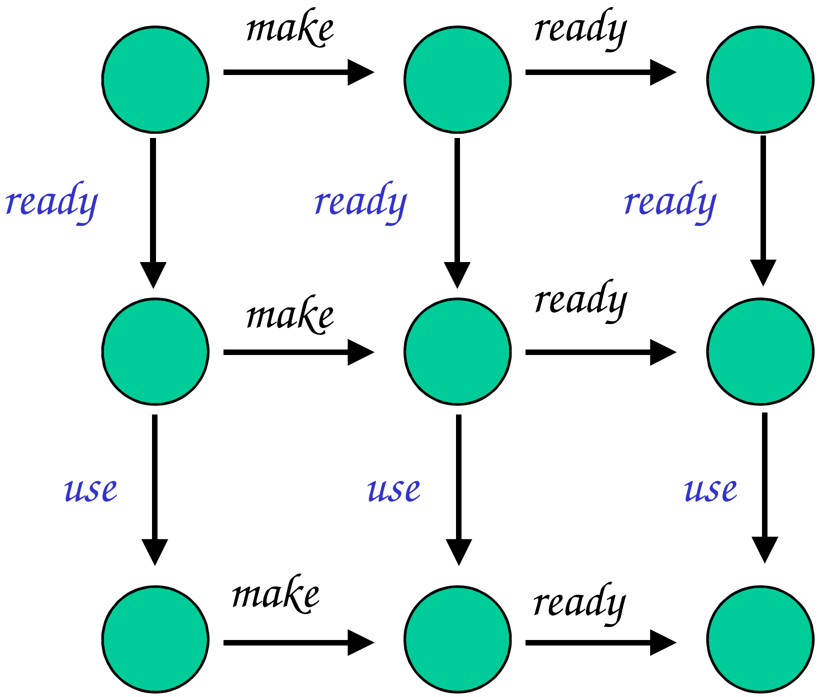
3 x 3 states?

Modeling Interaction - Example

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

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

3 states

3 states



3 x 3 states?

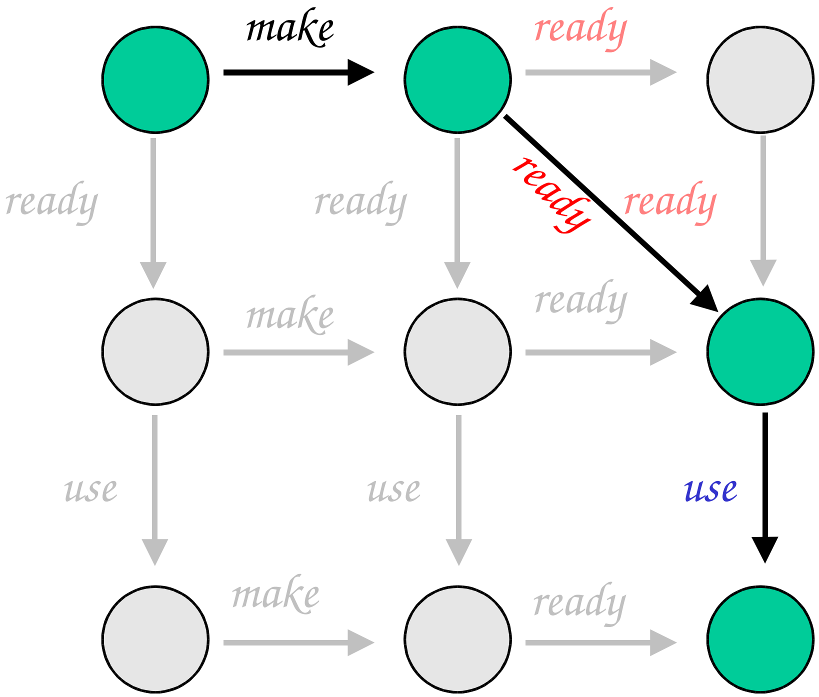
No…!

Modeling Interaction - Example

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

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

3 states

3 states

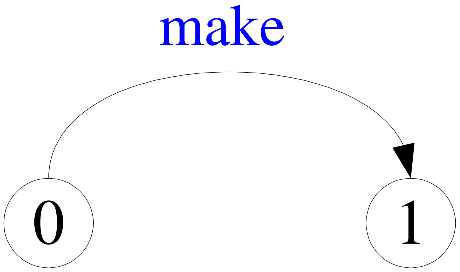


4 states!

Interaction constrains the overall behaviour.

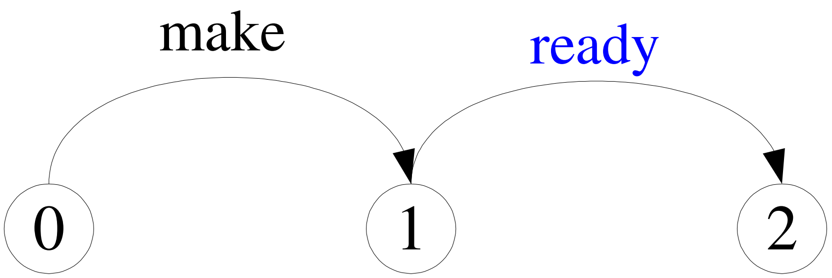
Modeling Interaction - Example

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

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



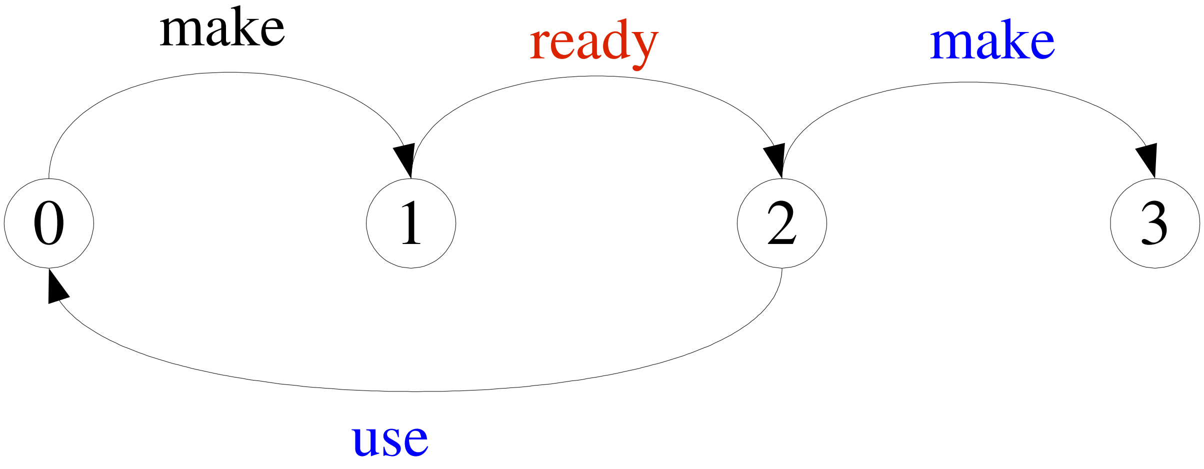
Modeling Interaction - Example

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

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



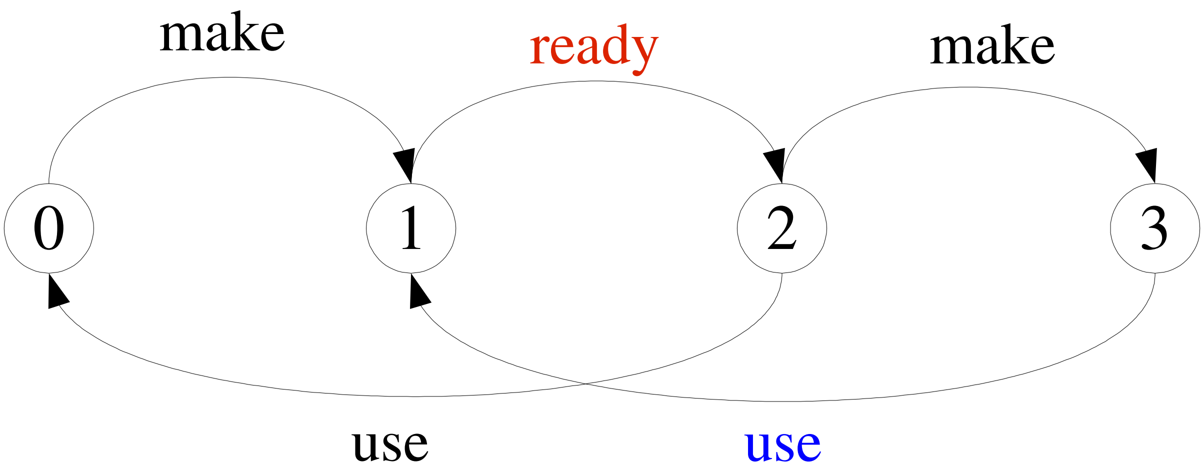
Modeling Interaction - Example

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

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



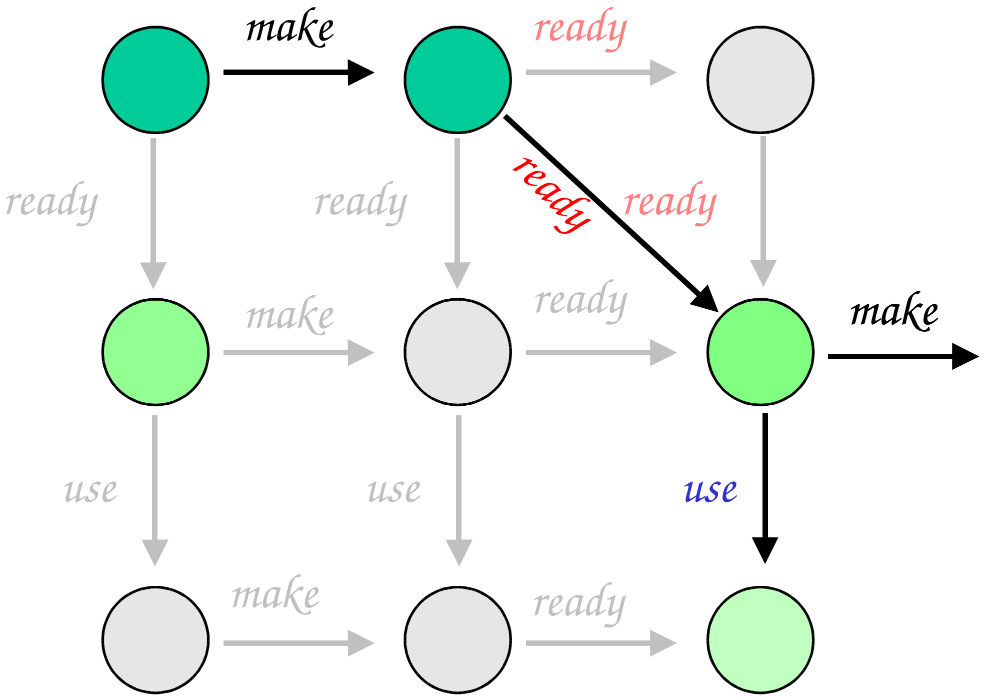
Modeling Interaction - Example

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

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



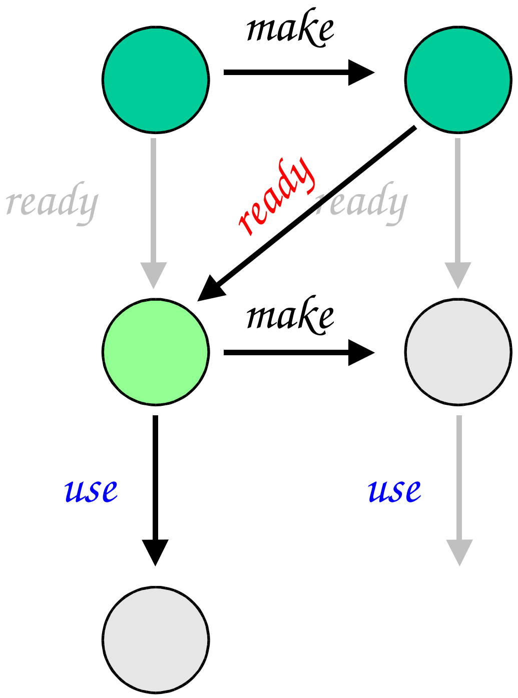
Modeling Interaction - Example

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

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



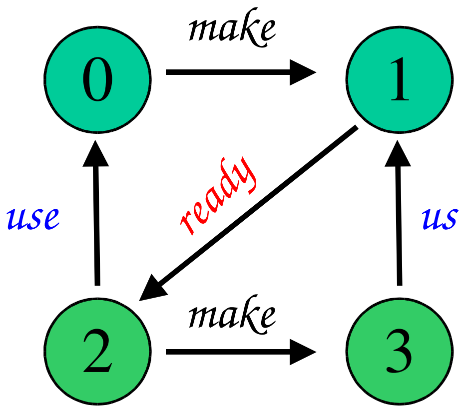
Modeling Interaction - Example

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

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



Modeling Interaction - Example

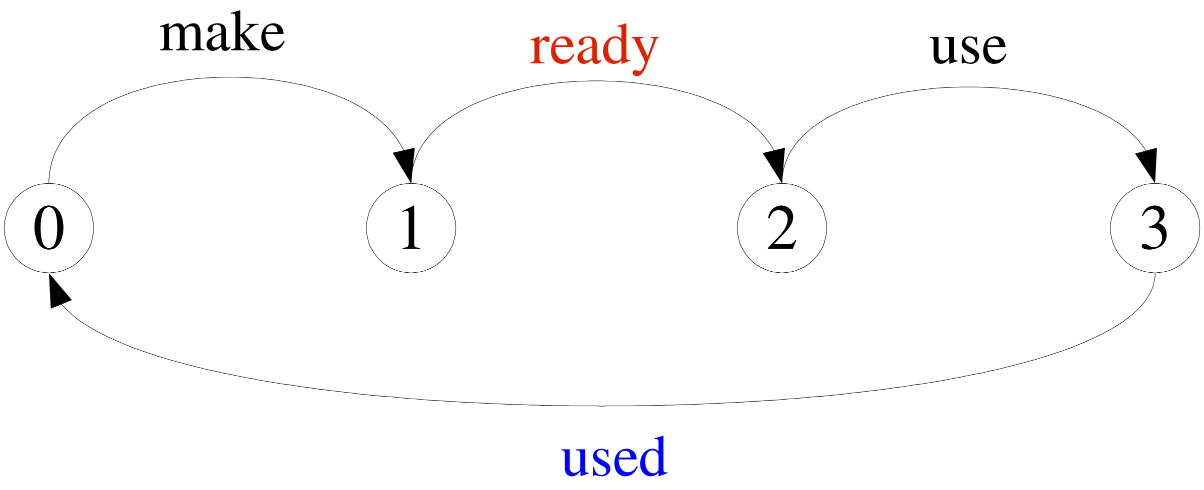
```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

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



Modeling Interaction - Handshake

A handshake is an action acknowledged by another:

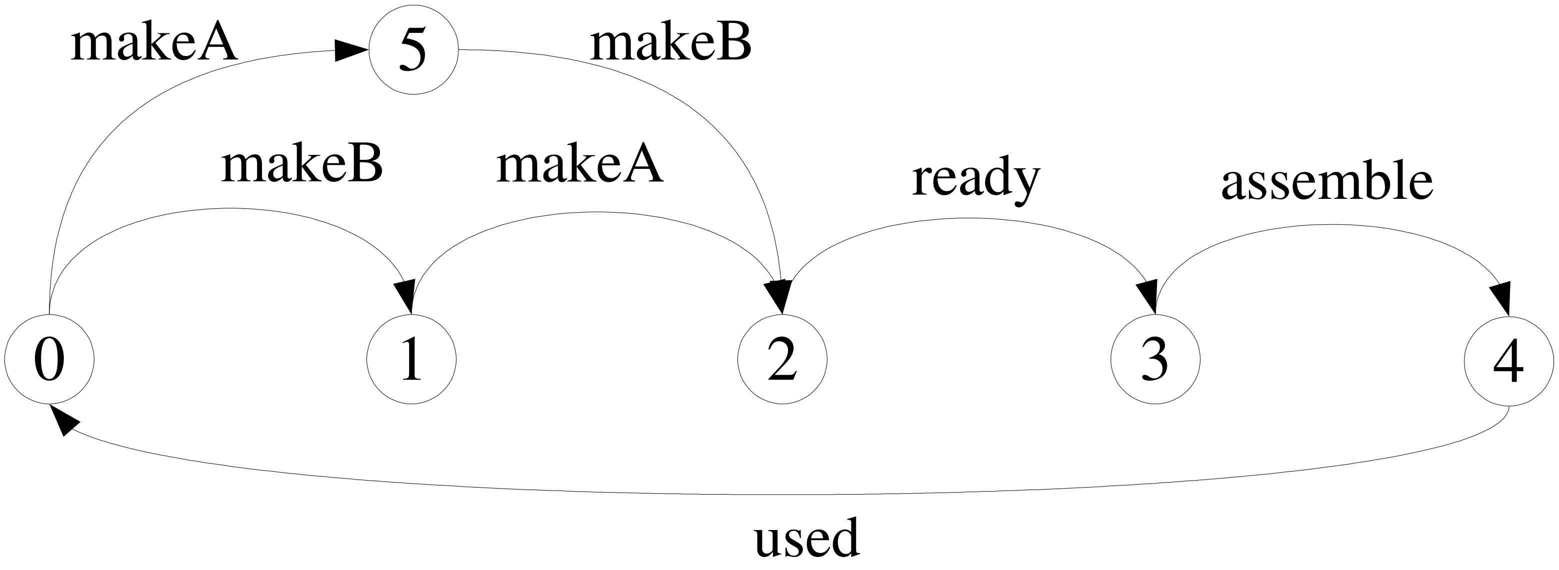
```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2  = (ready->use->used->USERv2).

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



Modeling 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).
```

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 Labeling

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).
```

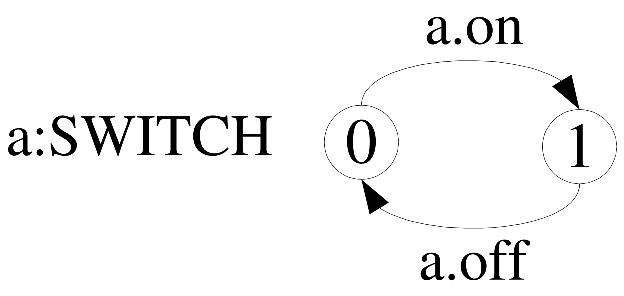
LTS? (a:SWITCH)

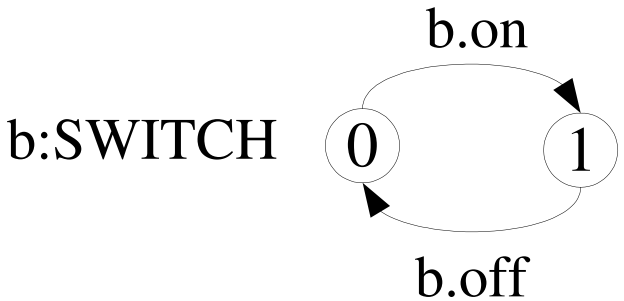
Process Labeling

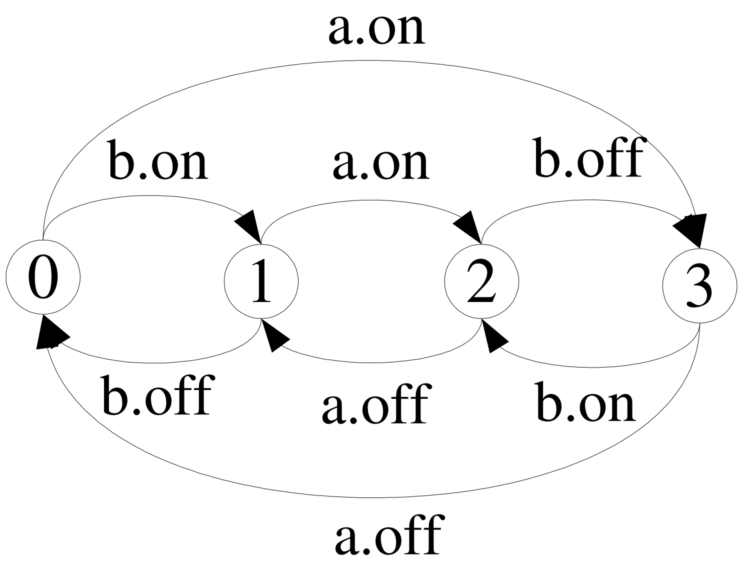
a:P *prefixes each action label in the alphabet of P with a.*

Two **instances** *of a switch process:*

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







	a	b
0:	off	off
1:	off	on
2:	on	on
3:	on	off

Process Labeling

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).
```

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 Labeling by a Set of Prefix Labels

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

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

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

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

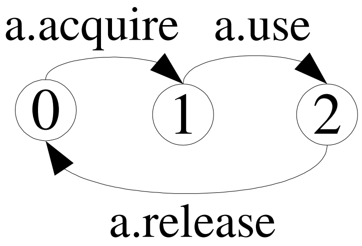
Process Prefix Labels for Shared Resources

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

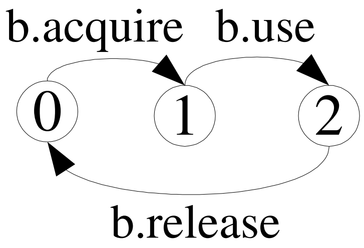
```
USER     = (acquire->use->release->USER).
```

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

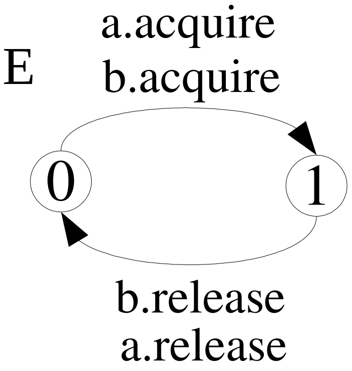
a:USER

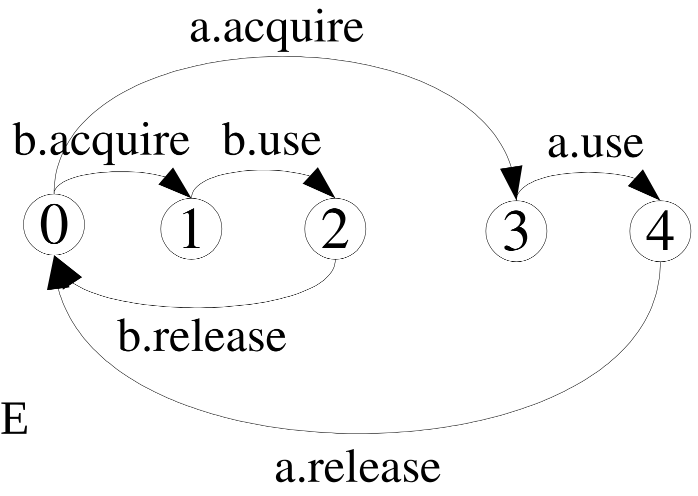


b:USER



{a,b}::RESOURCE





RESOURCE_SHARE

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

Action Relabeling

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

/{newlabel$_1$/oldlabel$_1$,… newlabel$_n$/oldlabel$_n$}.

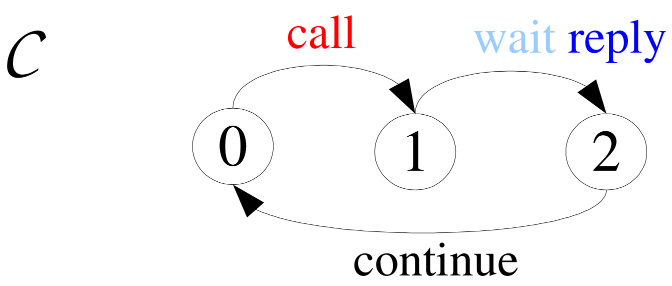
Relabeling to ensure that composed processes synchronize on particular actions:

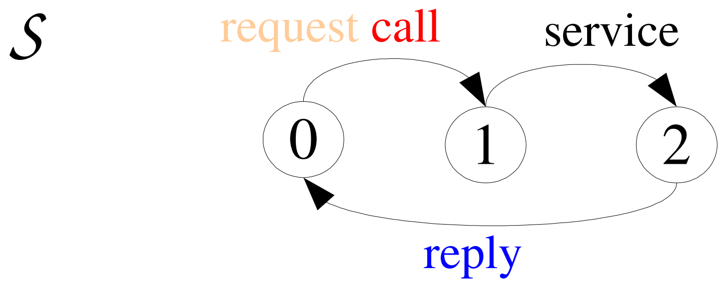
```
CLIENT = (call->wait->continue->CLIENT).
```

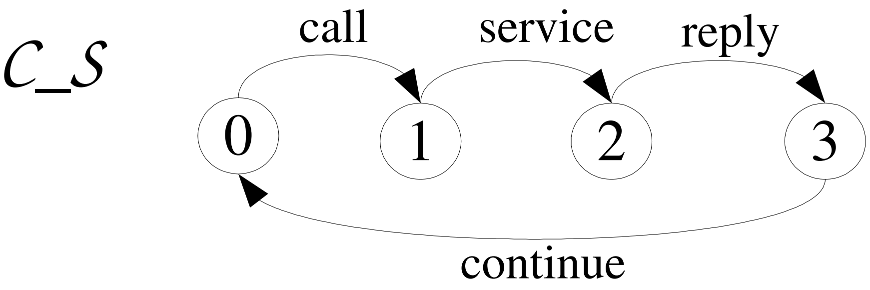
```
SERVER = (request->service->reply->SERVER).
```

Action Relabeling

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
C = (CLIENT /{reply/wait}).
S = (SERVER /{call/request}).
||C_S = (C || S).
```







Action Relabeling – Prefix Labels

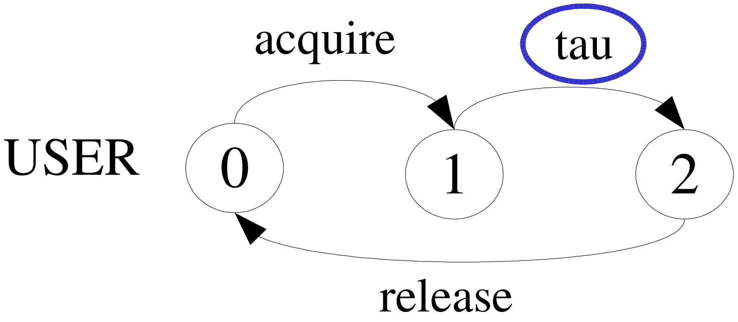
An alternative formulation of the client server system is described below using qualified or prefixed labels:

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

SERVERv2 = (accept.request

            ->service->accept.reply->SERVERv2).


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

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 labeled tau. Silent actions in different processes are not shared.*

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

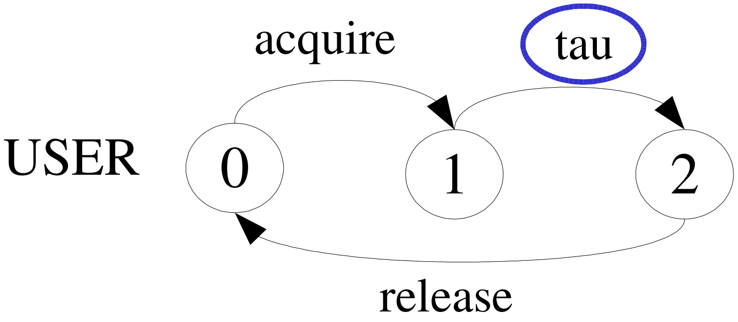


Action Hiding – Abstraction to Reduce Complexity

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

When applied to a process P, the interface operator **@{a1..ax}** *hides all actions in the alphabet of P not labeled in the set a1..ax.*

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

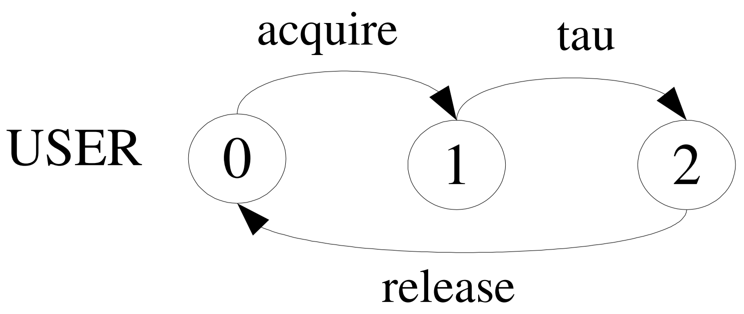


Action Hiding

The following definitions are equivalent:

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

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

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

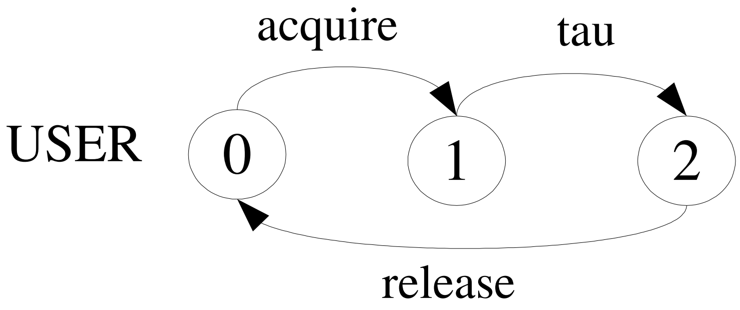


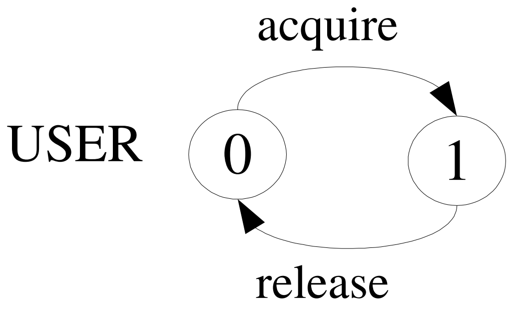
Action Hiding

The following definitions are equivalent:

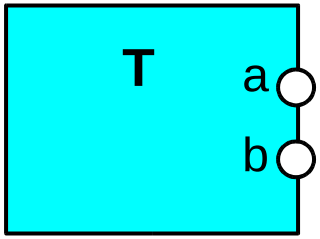
```
USER = (acquire->use->release->USER)
       \{use}.

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

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



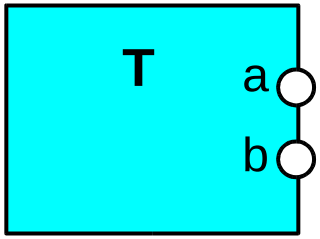


Structure Diagrams

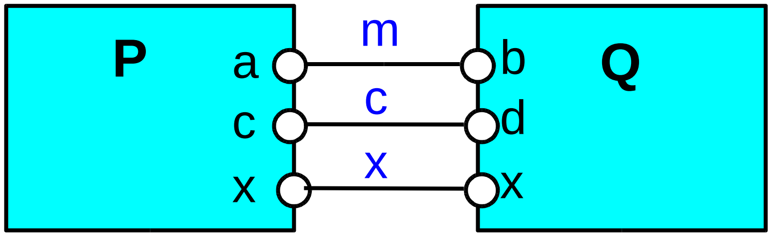


Process T with alphabet {a,b}.

Structure Diagrams



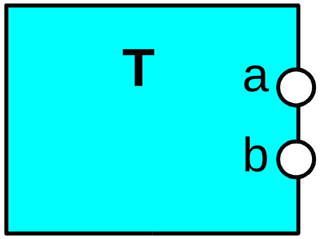
Process T with alphabet {a,b}.



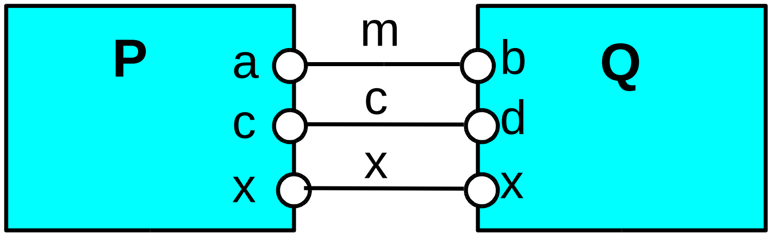
Parallel Composition

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

Structure Diagrams

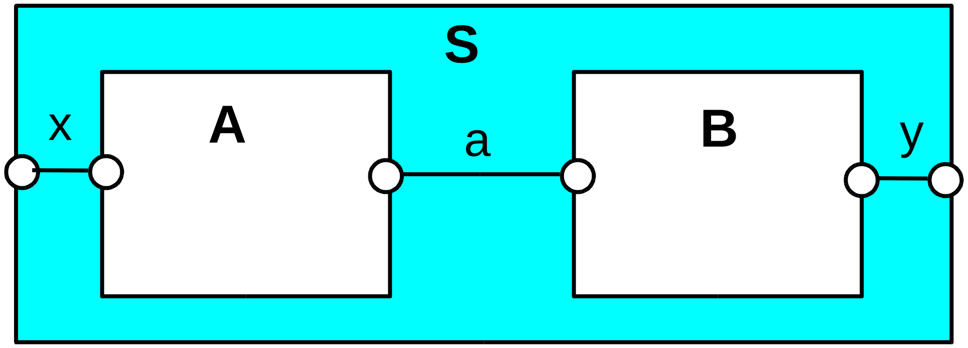


Process T with alphabet {a,b}.



Parallel Composition

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



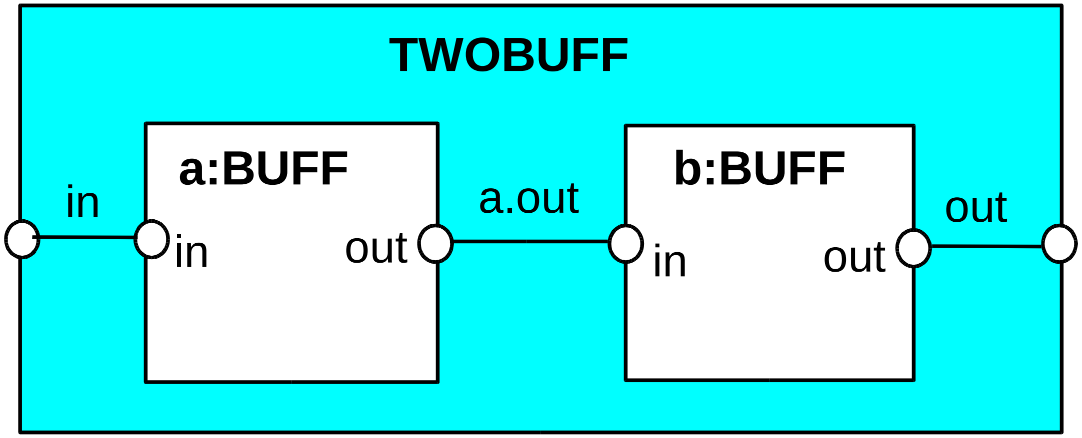
Composite process

||S = (A||B) @ {x,y}

Structure Diagrams

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

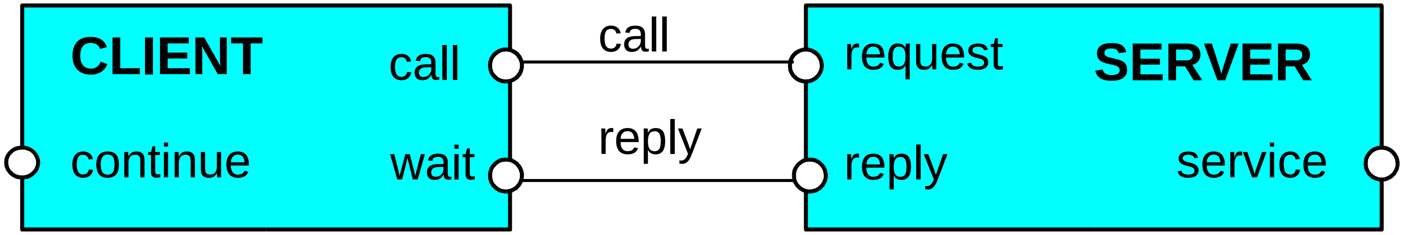
We use structure diagrams to capture the structure of a model expressed by the static combinators: parallel composition, relabeling and hiding.



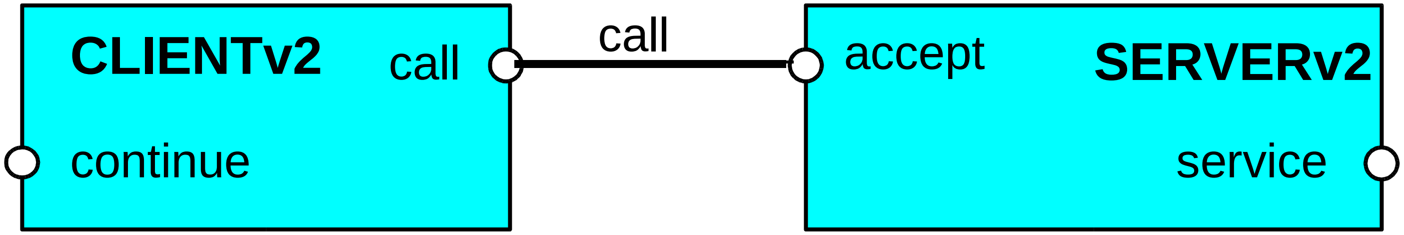
```
||TWOBUF =  (a:BUFF || b:BUFF)
                  /{in/a.in, a.out/b.in, out/b.out}
                     @{in,out}.
```

Structure Diagrams

Structure diagram for **`CLIENT_SERVER`** *?*



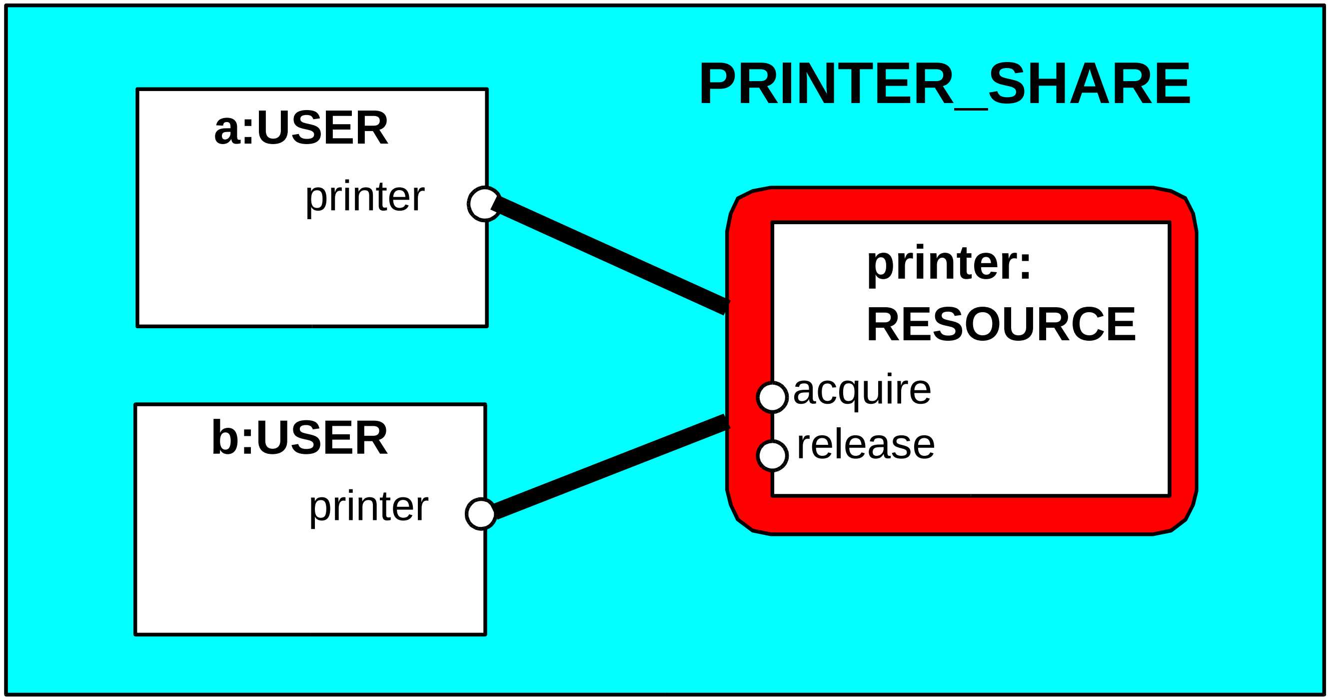
Structure diagram for **`CLIENT_SERVERv2`** *?*



Structure Diagrams – Resource Sharing

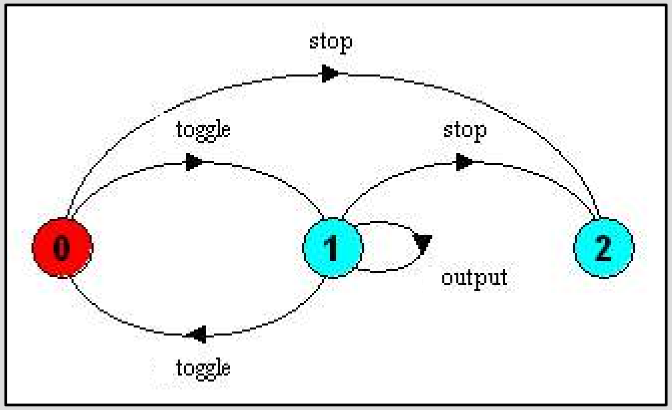
```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

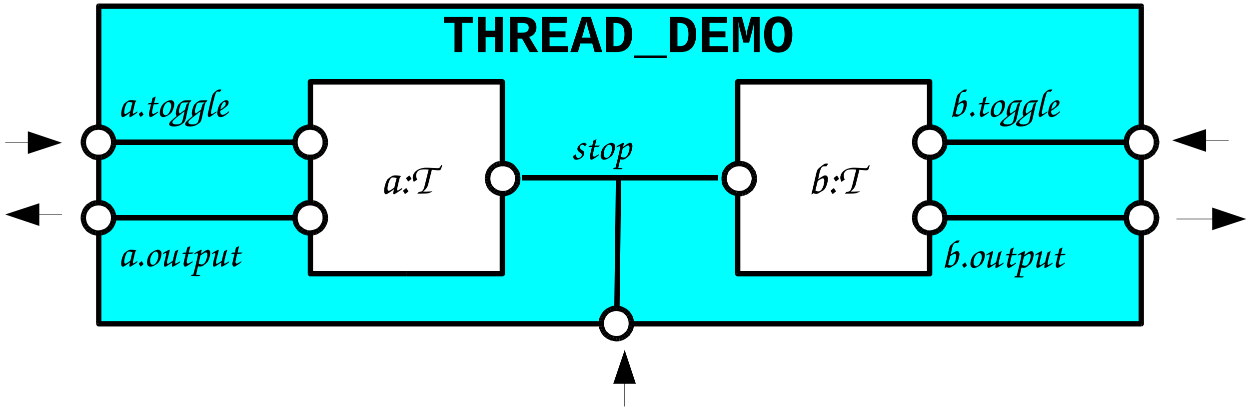
```
||PRINTER_SHARE =
   (a:USER || b:USER || {a,b}::printer:RESOURCE).
```



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() {
        on = false;
    }
    void toggle() { on = !on; }
    void abort()  { interrupt(); }
    private void output() { System.out.println(“output”); }
    public void run() {
        try {
            while (true) {
                if (on) output();
                sleep(1000);
            }
        } catch(InterruptedException _) {
            System.out.println(“Done!”);
        }
    }
}
```

ThreadDemo Code: MyThread

```
class MyThread extends Thread {
    private boolean on;
    MyThread() {
        on = false;
    }
    void toggle() { on = !on; }
    void abort()  { interrupt(); }
    private void output() { System.out.println(“output”); }
    public void run() {
        try {
            while (true) {
                if (on) output();
                sleep(1000);
            }
        } catch(InterruptedException _) {
            System.out.println(“Done!”);
        }
    }
}
```

Interpret:

toggle*, *abort

as inputs

output

as output

ThreadDemo Code: ThreadDemo

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

Summary

- **Concepts:** concurrent processes and process interaction.
- **Models:**
  - asynchronous *(arbitrary speed)* & interleaving *(arbitrary order)*
  - parallel composition *(finite state process with action interleaving)*
  - process interaction *(shared actions)*
  - process labeling, action relabeling, and hiding
  - structure diagrams
- **Practice:** multiple threads in Java.