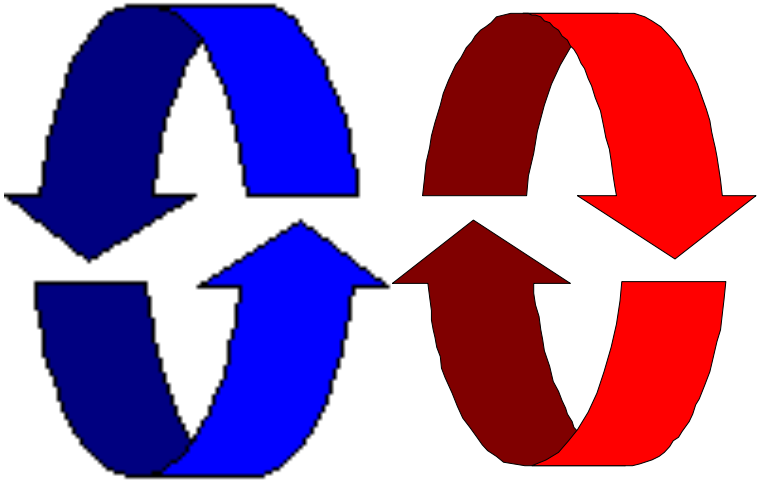
Chapter 3

Concurrent Execution



Concurrent Execution

Concepts: processes - concurrent execution, interleaving.
process interaction.

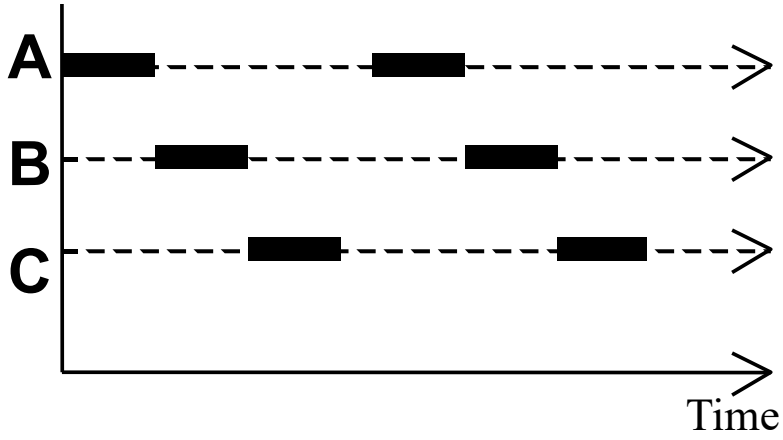
Models: parallel composition of asynchronous processes
- action interleaving
interaction - shared actions (synchronous)
process labelling, action re-labelling, hiding,
structure diagrams

Practice: Multithreaded Java programs.

Definitions

- ***(Pseudo) Concurrency***
  - ***Logically*** simultaneous processing.

  Does not imply multiple processing elements (PEs). Requires interleaved execution on **a single PE**.



- ***Parallelism***
  - ***Physically*** simultaneous processing.

  Involves multiple PEs and/or independent device operations.

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.

3.1 Modelling Concurrency

- How should we model process execution speed?
  - arbitrary speed

    (we abstract away time)

- How do we model concurrency?
  - arbitrary **relative order** of actions from **different processes**

    (**interleaving** but preserve **each** process's **order**)

- What is the result?
  - provides a general model independency of scheduling

    (**asynchronous** model of execution)

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.

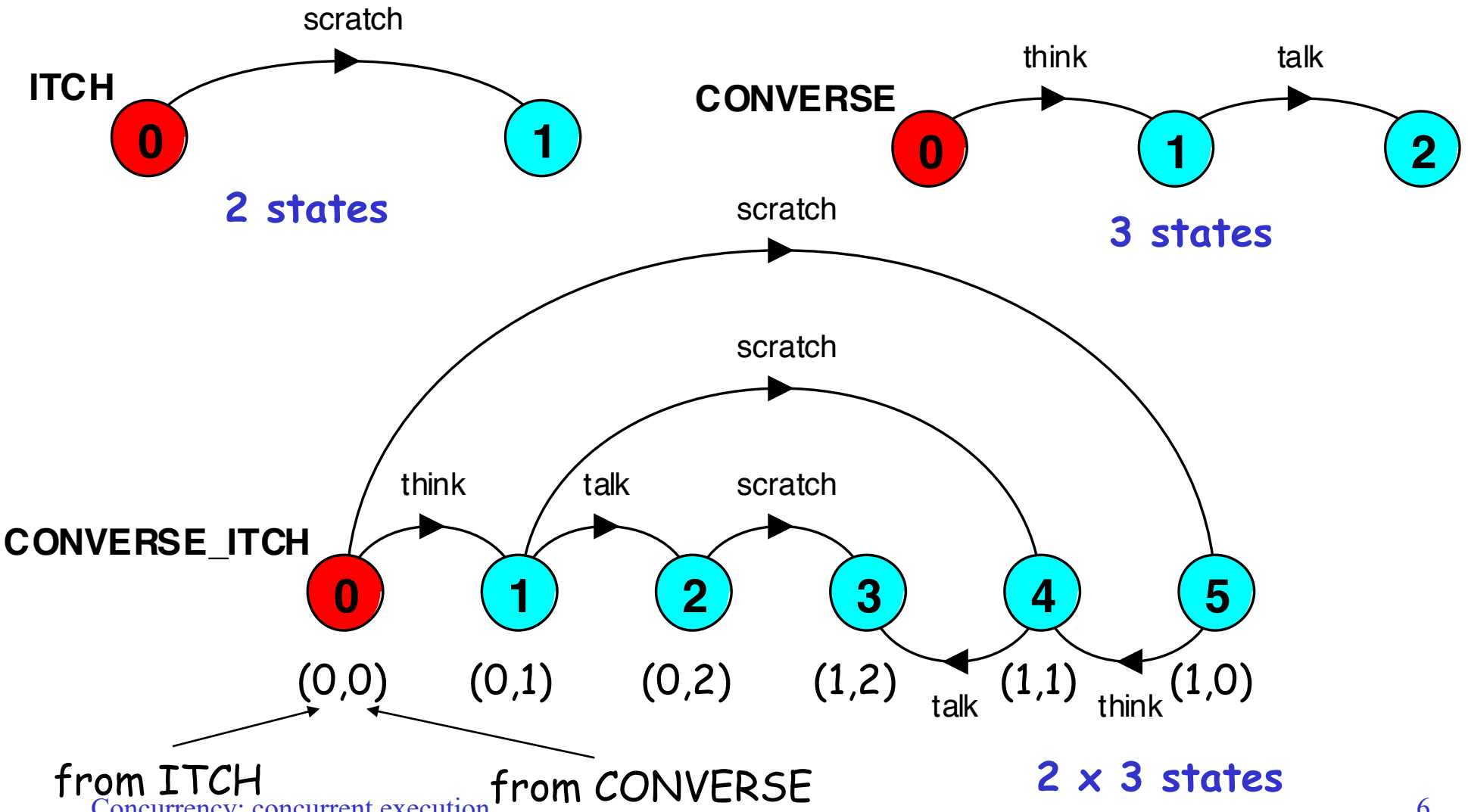
```
ITCH    = (scratch->STOP).
CONVERSE = (think->talk->STOP).

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

Disjoint alphabets

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

Possible traces as a result of action interleaving.

Parallel Composition – Action Interleaving



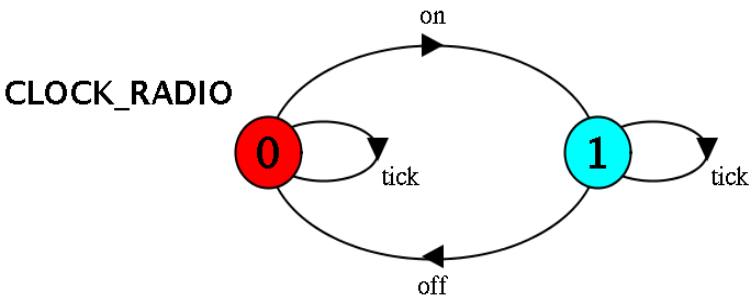
Parallel Composition – Algebraic Laws

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).
```



Modeling Interaction – Shared Actions

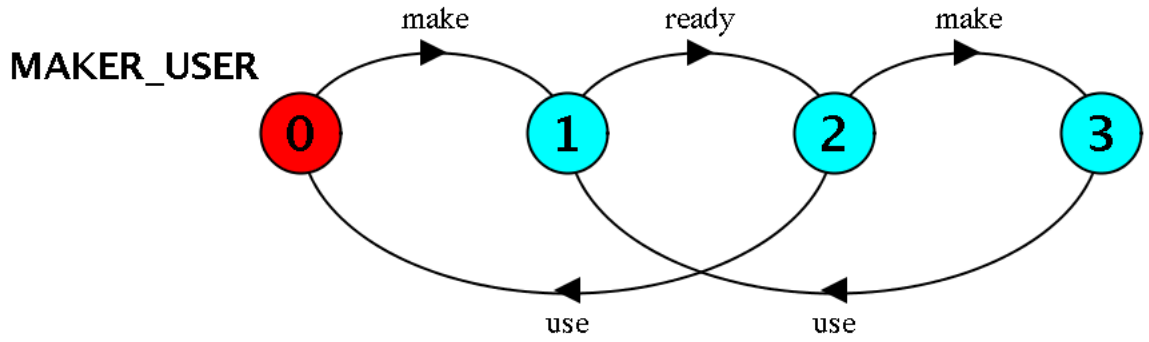
- If processes in a composition have **actions in common**, these actions are said to be ***shared***.
- Shared actions is the way that process **interaction** is modelled. While unshared actions may be arbitrarily interleaved, a ***shared action*** **must be executed at the same time** by all processes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

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

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

MAKER **synchronizes** with **USER** when **ready**.

Modeling Interaction – Shared Actions

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

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



make→ready→make→use→ready→ ...

Modeling Interaction - Handshake

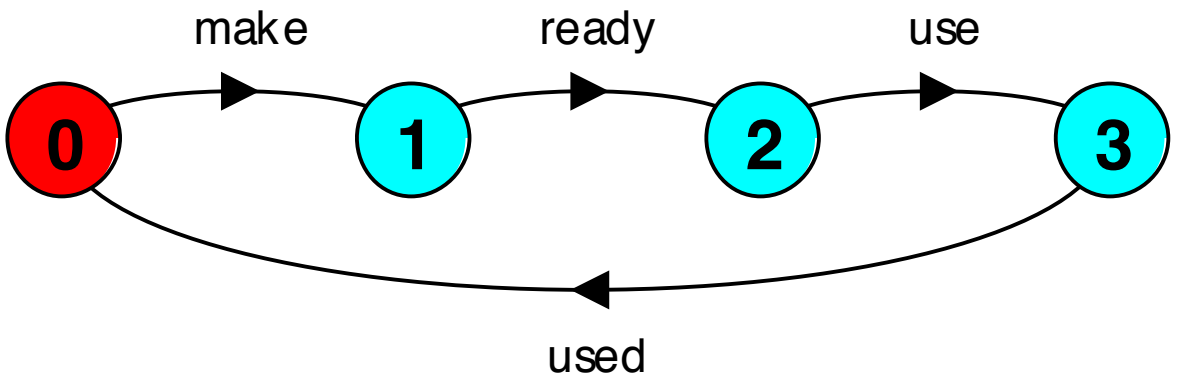
A handshake is an action (e.g., used) acknowledged by another:

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

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

3 states

3 states

3 x 3 states?



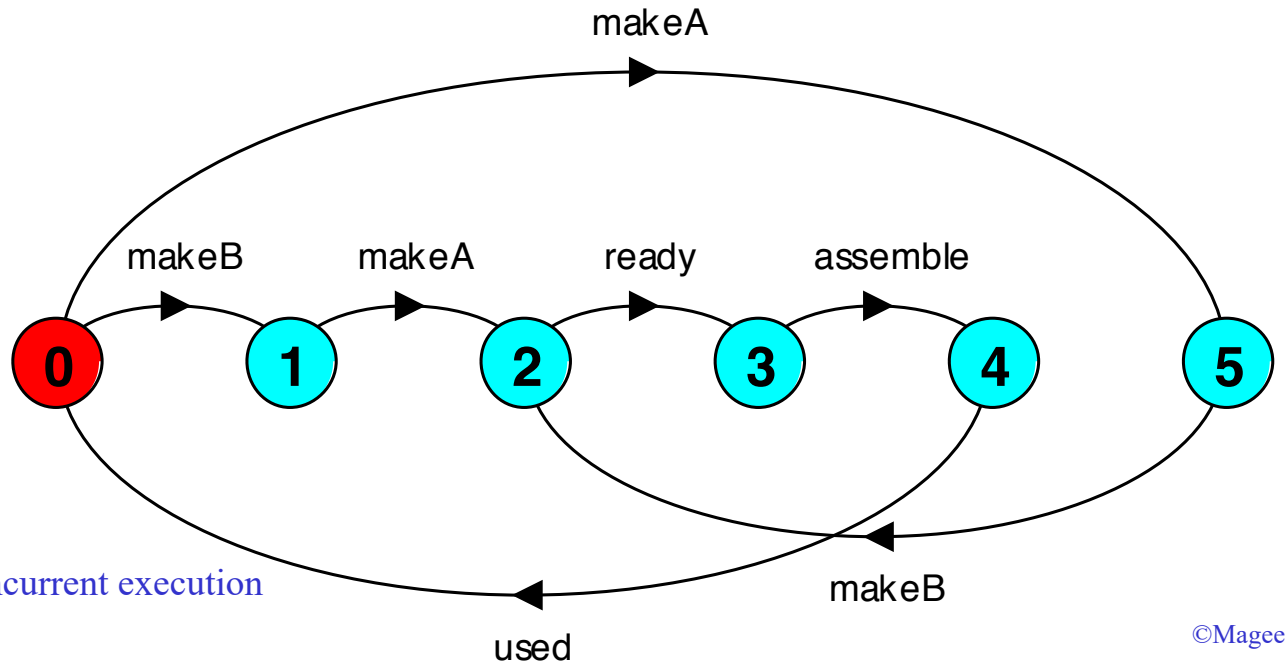
4 states

Interaction constrains the overall behaviour.

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).
```

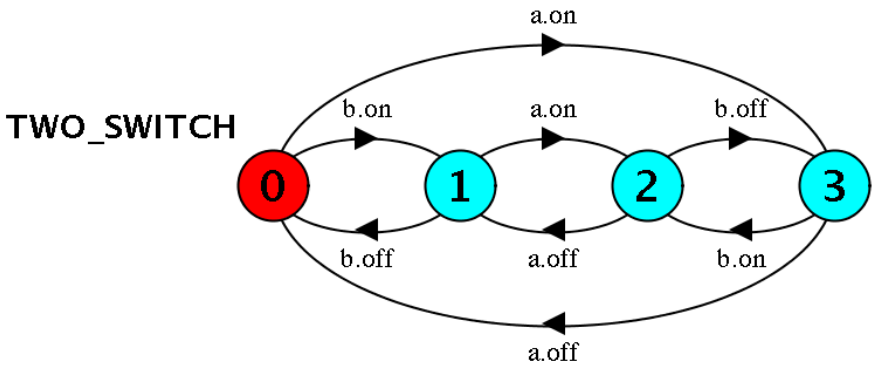
Substituting the definition for **MAKERS** in **FACTORY** and applying the **commutative** and **associative** laws for parallel composition results in the original definition for **FACTORY** in terms of **primitive processes**.

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

Process Instances and Labelling

- a:P: instance "a" of process P; prefixes each **action** in the alphabet of process P. (e.g., a.on)
- Two **instances** (**a** and **b**) of 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 Labelling by a Set of Instances

- {a1,..,ax}::P replaces every action label n in the alphabet of P process with the **labels** a1.n,..., ax.n.

  Note: a1, a2,...,ax: instances of process P

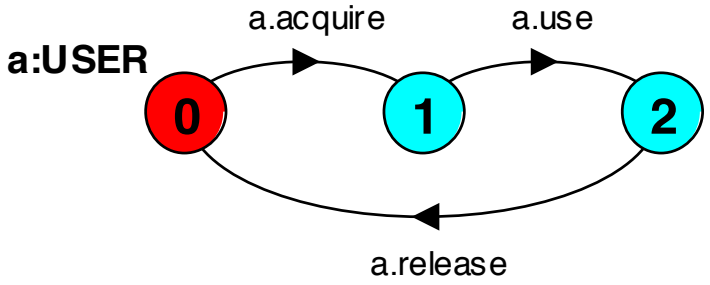
- Every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,...,ax.n} ->X).

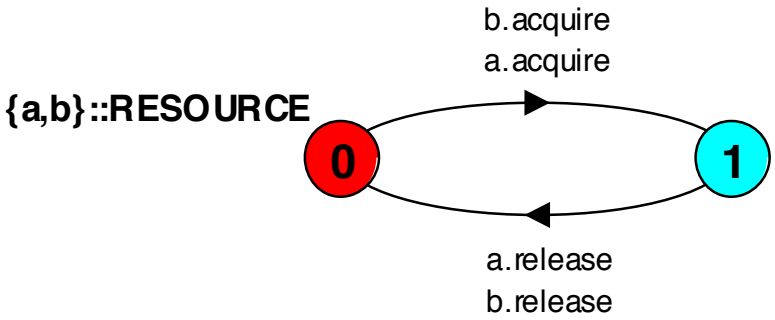
Process prefixing is useful for modelling **shared** resources:

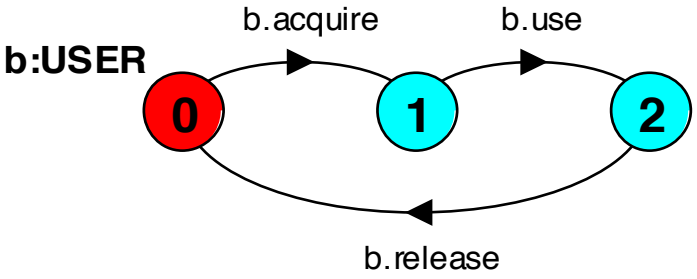
```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

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

Instances: a and b

Process Prefix Labels for Shared Resources

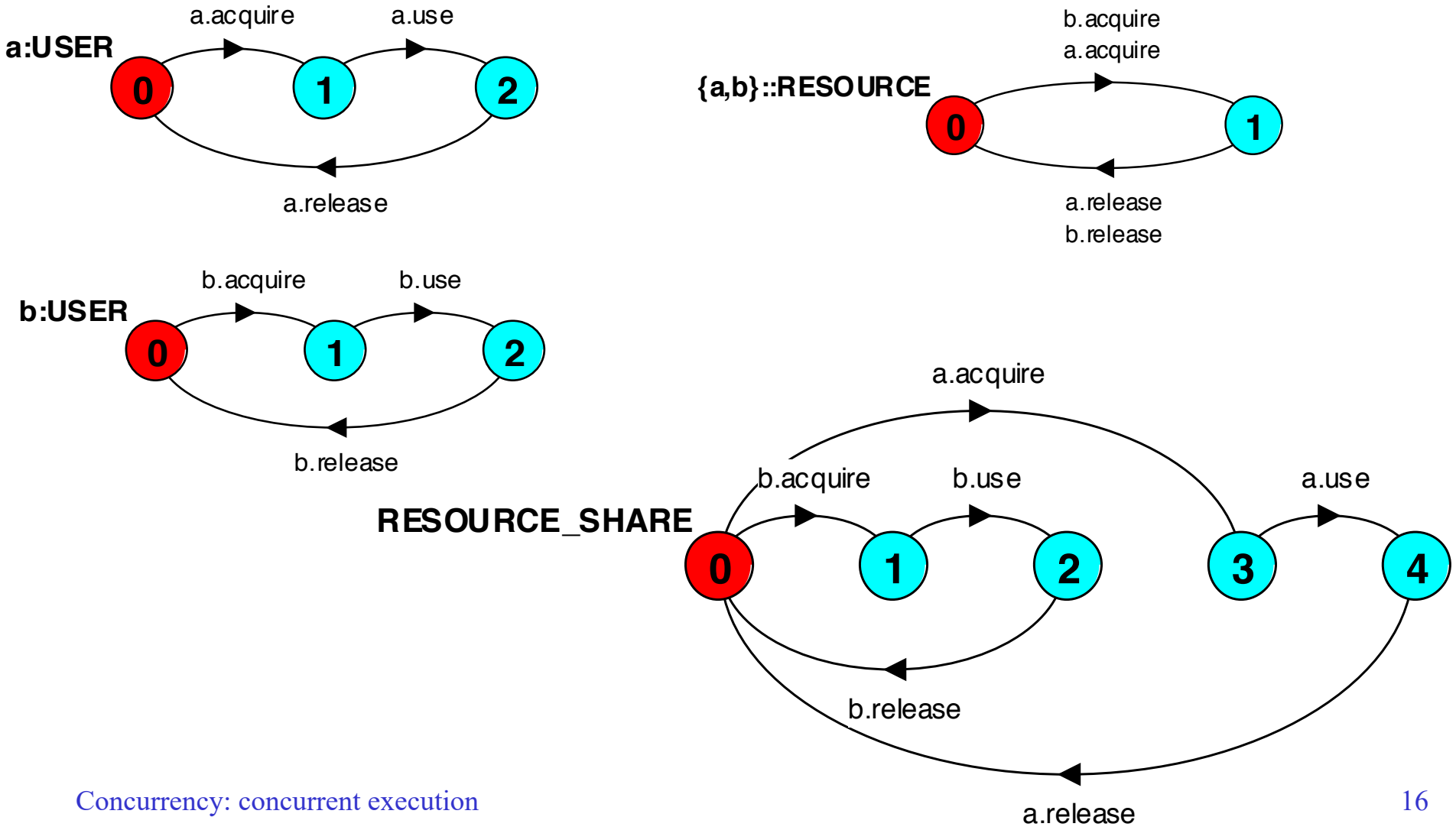






How does the model ensure that the user that acquires the resource is the one to release it?

Process Prefix Labels for Shared Resources



Action Re-Labelling

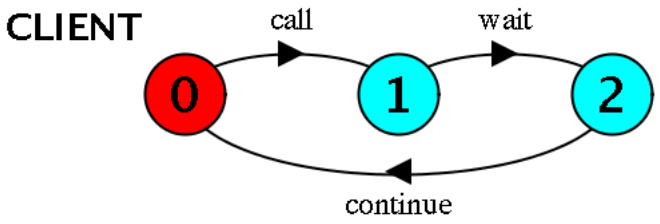
- Re-labelling functions are applied to processes to change the names of action labels.
- The general form of the re-labelling function is:
  /{*newlabel_1*/*oldlabel_1*, .. *newlabel_n*/*oldlabel_n*}.
- Relabelling to ensure that composed processes **synchronize** on particular actions.

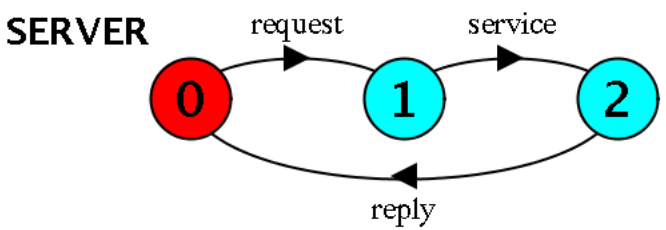
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

Note that both *newlabel* and *oldlabel* can be sets of labels.

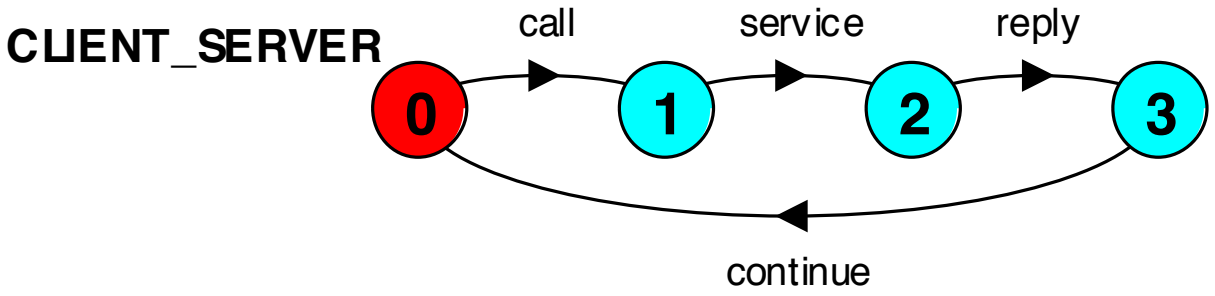
action re-labelling

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





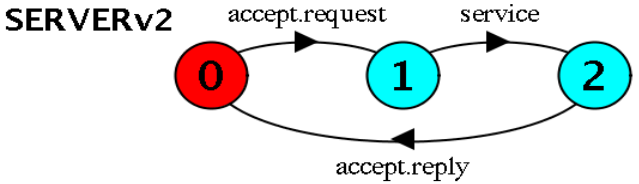

```
||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```

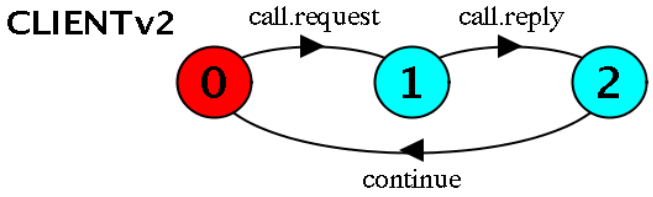


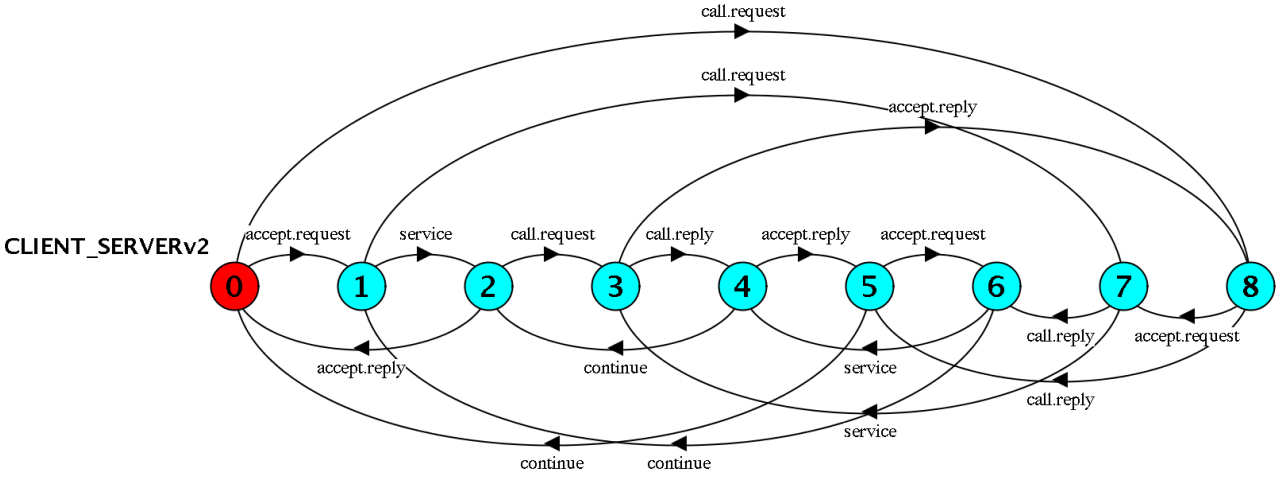
Action Re-Labelling – Prefix Labels

An alternative client server system: qualified or prefixed labels:

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

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2).
```



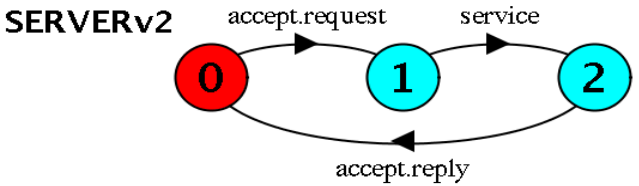


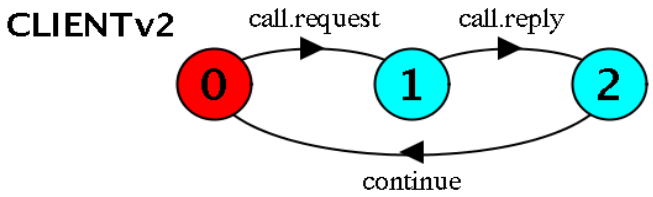


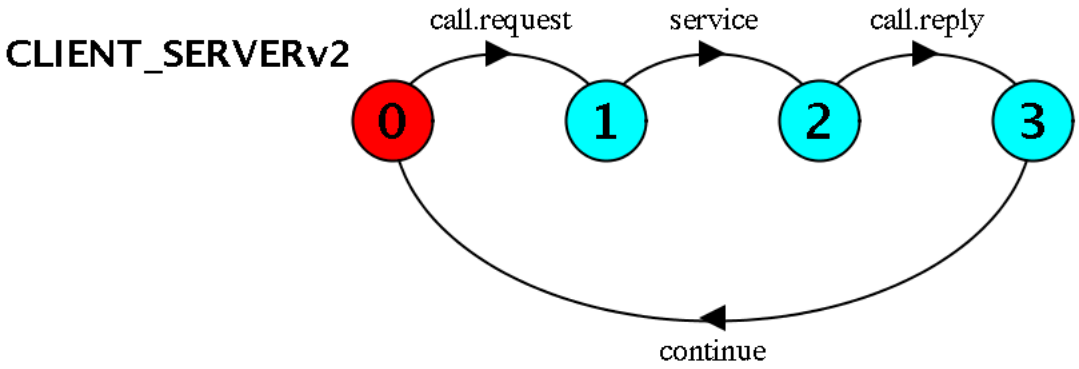
Action Re-Labelling – Prefix Labels

An alternative client server system: qualified or prefixed labels:

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

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







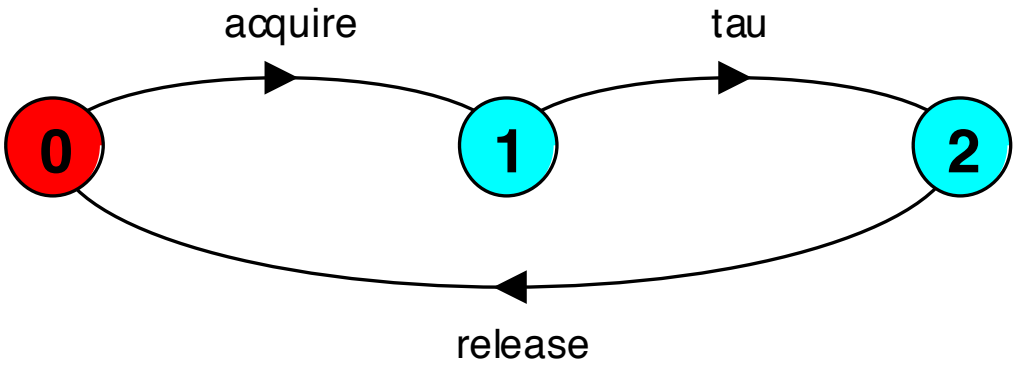
Action Hiding - Abstraction to Reduce Complexity

- When applied to a process P, the hiding operator \{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.
- Sometimes it is more convenient to specify the set of labels to be exposed....using @{a1..ax}

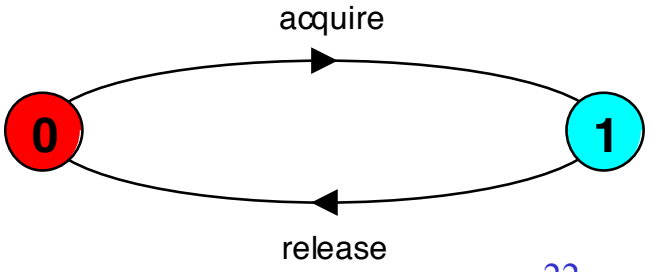
Action Hiding

The following definitions are equivalent:

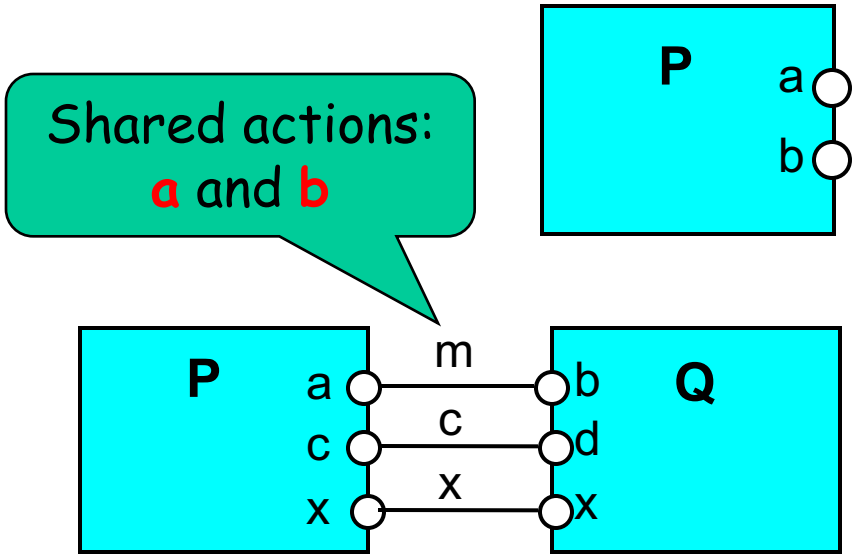
```
USER = (acquire->use->release->USER)
       \{use}.

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



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

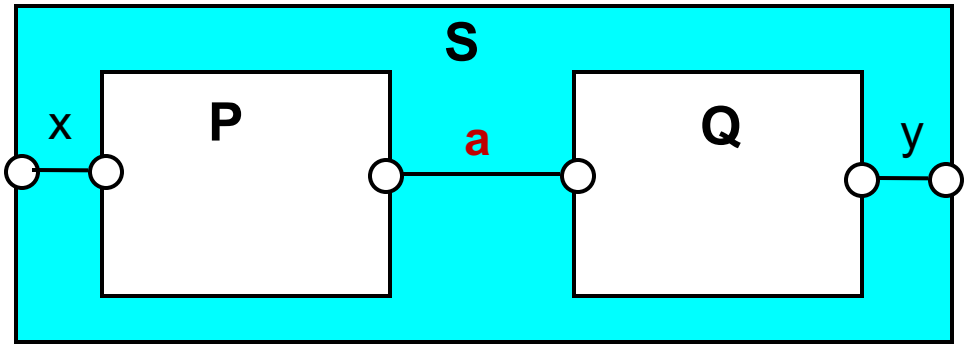


Structure Diagrams – Systems as Interacting Processes



Process P with alphabet {a,b}.

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



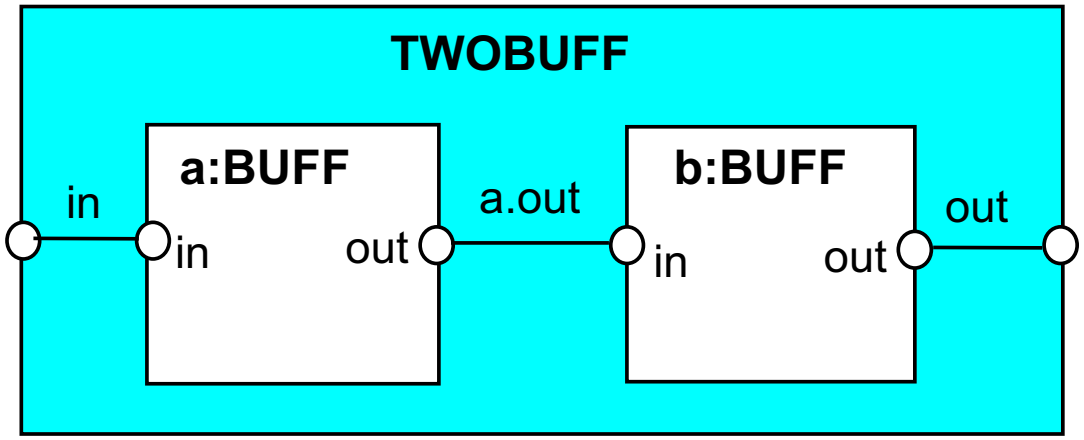
Composite process
||S = (P||Q) @ {x,y}

a : shared action between P and Q

Structure Diagrams

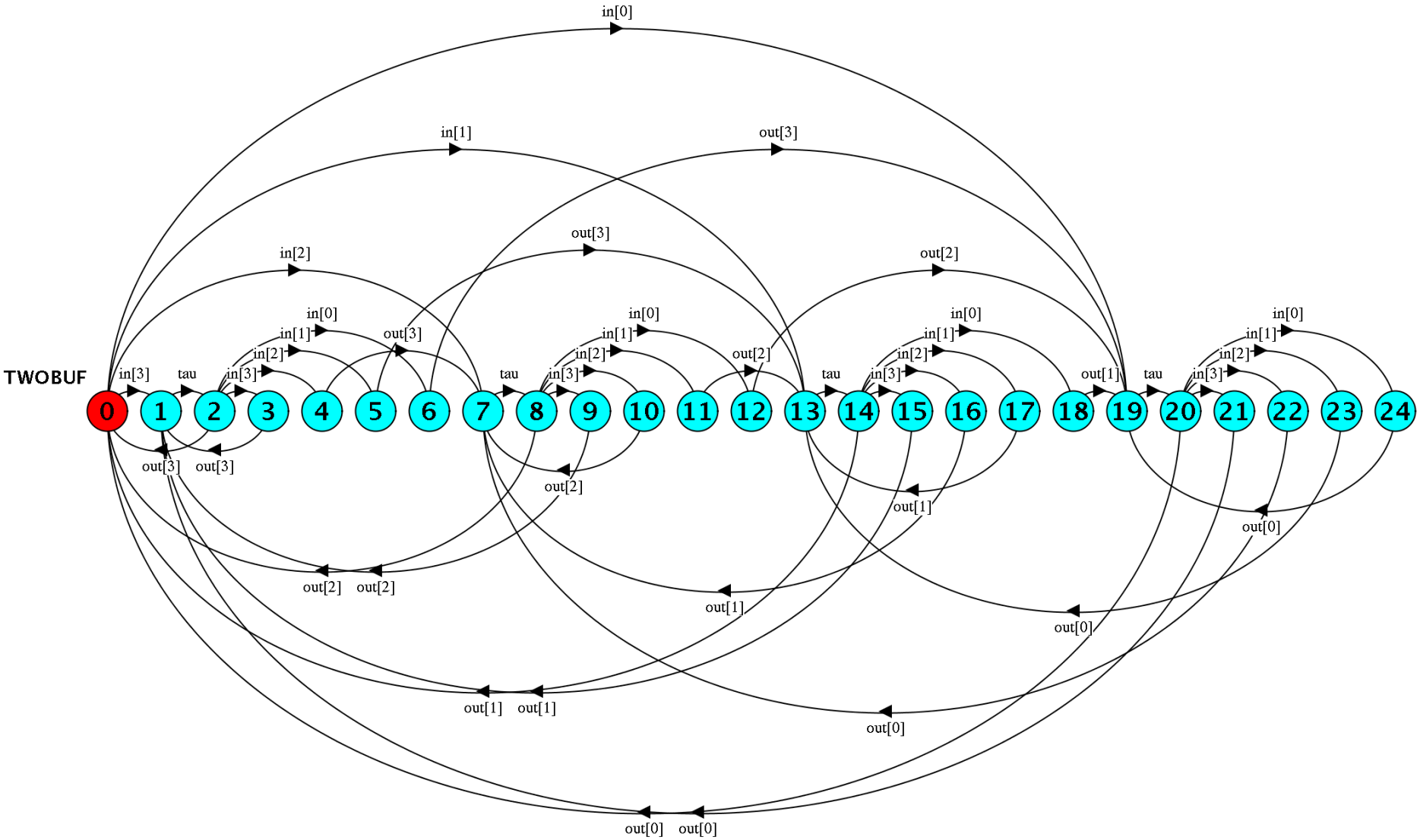
We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.

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

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

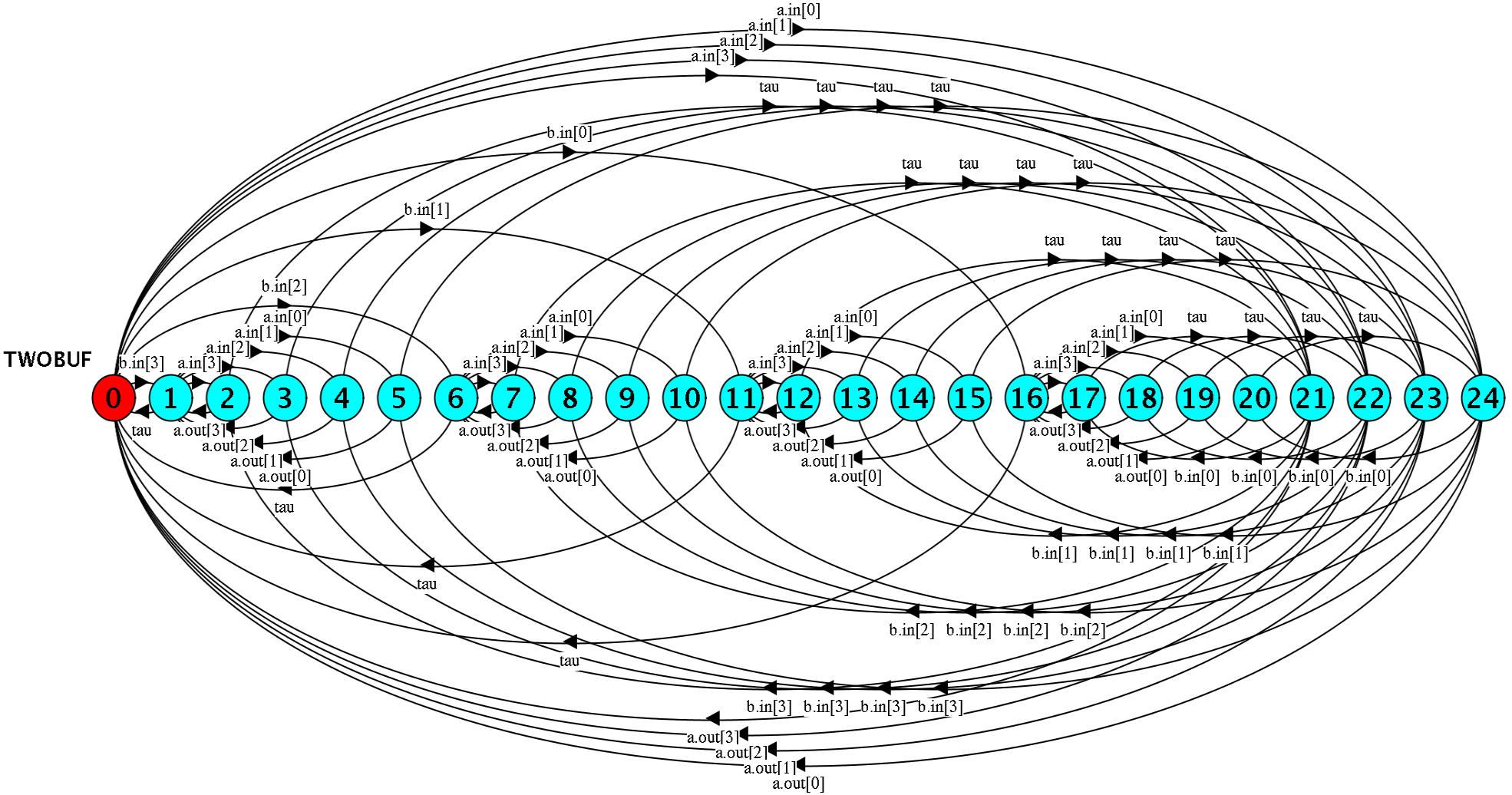



```
Labels in,
out, a.out
: in[i:T],
out[i:T],
a.out[i:T]
```

Structure Diagrams



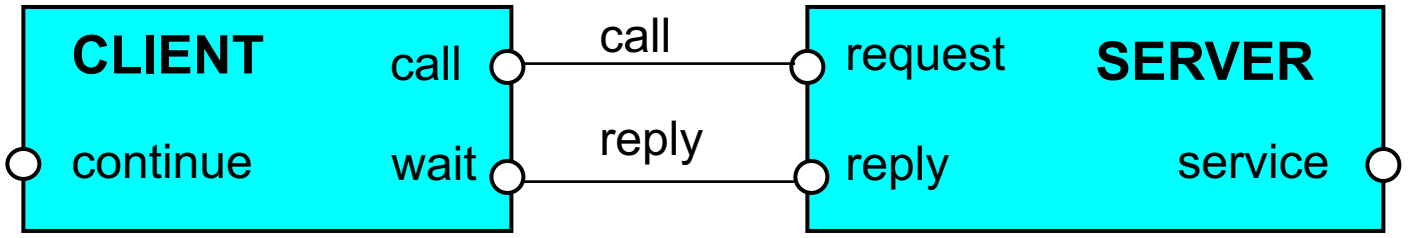
Structure Diagrams



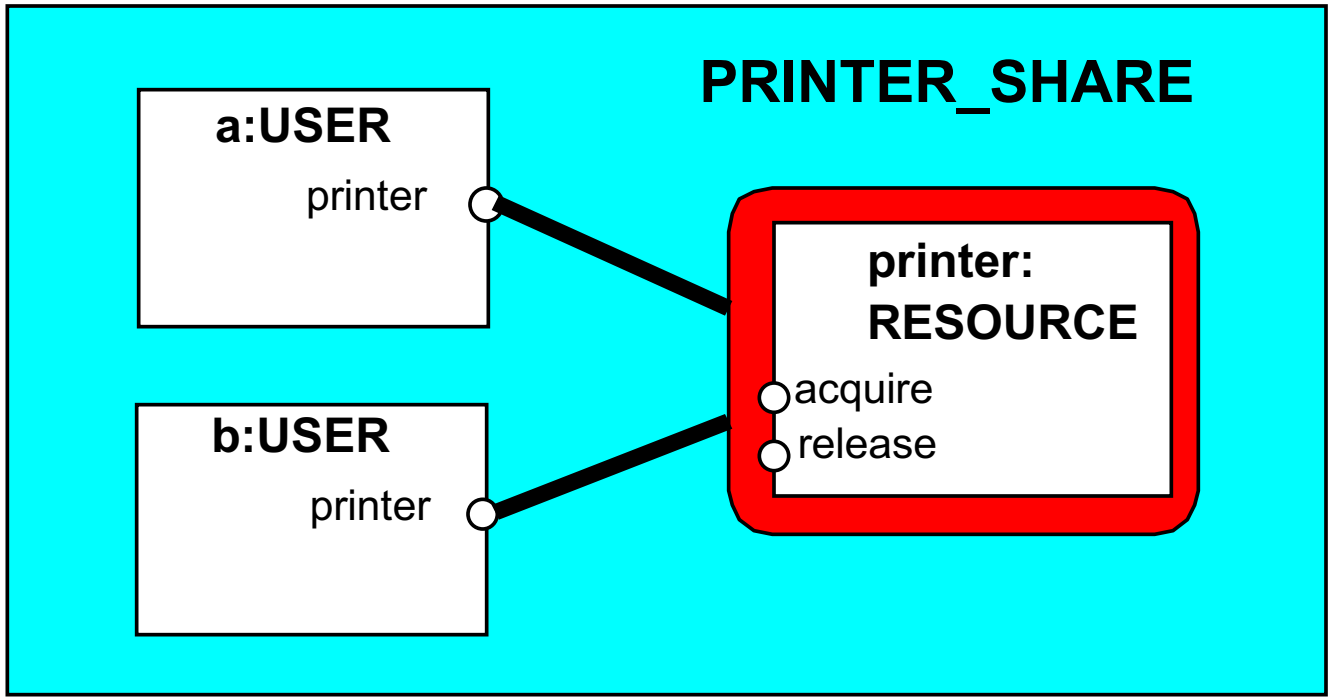
Structure Diagrams

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

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

Structure diagram for `CLIENT_SERVER` ?

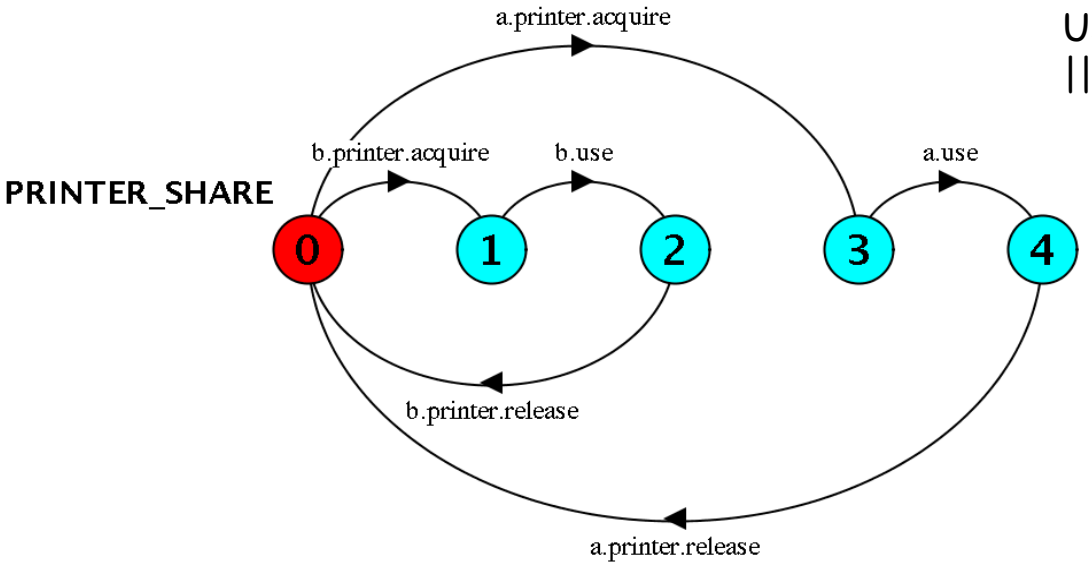


Structure Diagrams – Resource Sharing

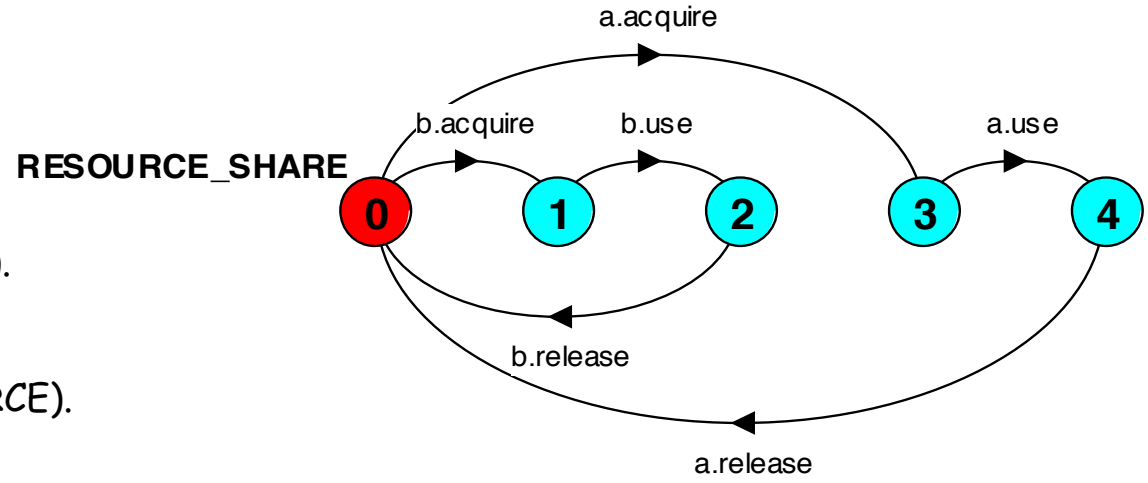



```
RESOURCE = (acquire->release->RESOURCE).
USER =     (printer.acquire->use
            ->printer.release->USER)\{use}.

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

Structure Diagrams – Resource Sharing



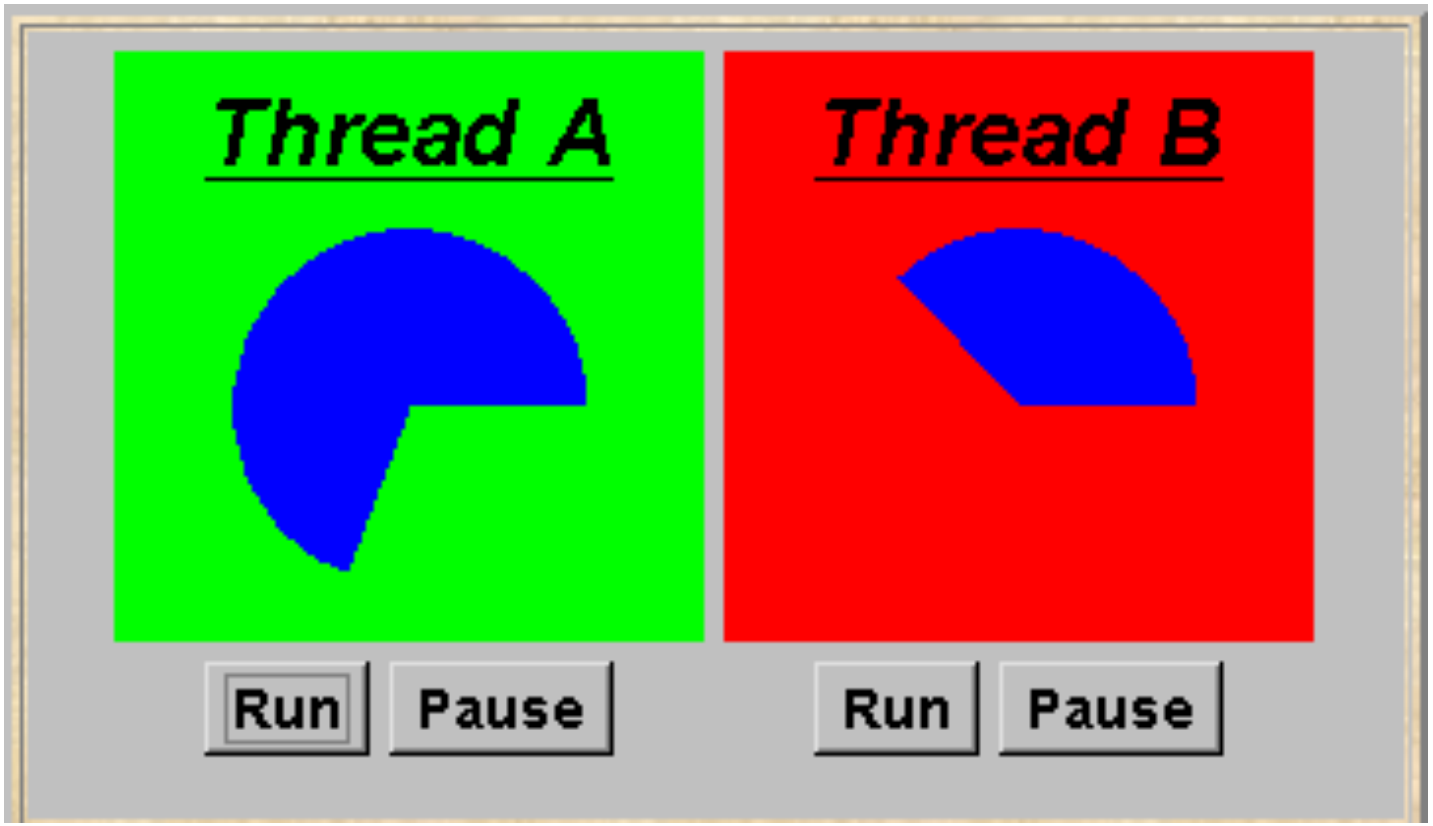

```
RESOURCE =  (acquire->release->RESOURCE).
USER =  (printer.acquire->use->printer.release->USER).
||PRINTER_SHARE =
    (a:USER||b:USER||{a,b}::printer:RESOURCE).
```



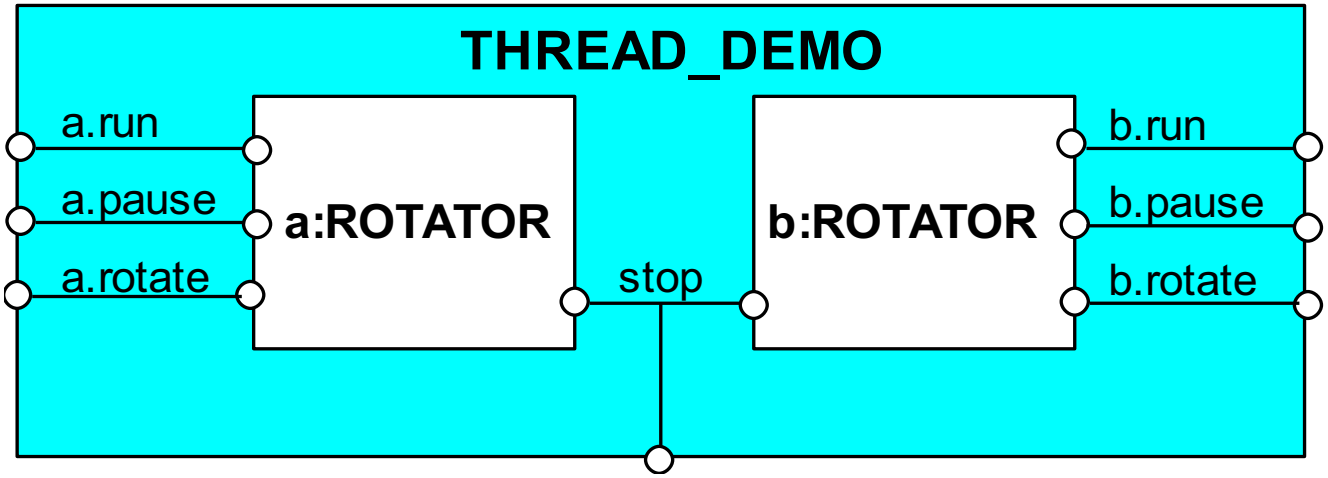

```
RESOURCE = (acquire->release->RESOURCE).
USER =     (acquire->use->release->USER).
||RESOURCE_SHARE =
            (a:USER||b:USER||{a,b}::RESOURCE).
```

3.2 Multi-threaded Programs in Java

Concurrency in Java occurs when more than one thread is alive.
ThreadDemo has **two threads** which rotate displays.



ThreadDemo model

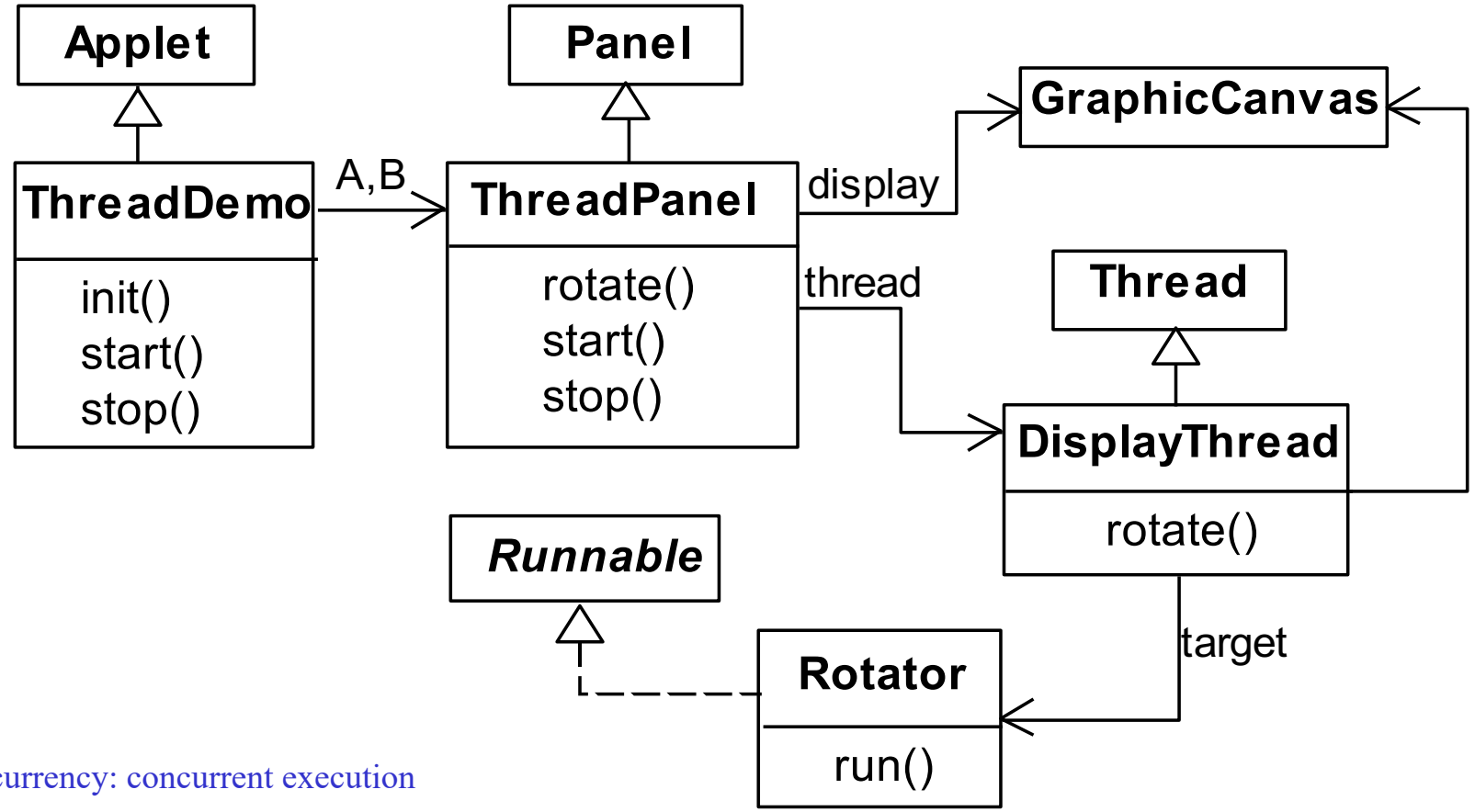



```
ROTATOR = PAUSED,
PAUSED  = (run->RUN | pause->PAUSED
          |interrupt->STOP),
RUN     = (pause->PAUSED |{run,rotate}->RUN
          |interrupt->STOP).

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)

 /{stop/{a,b}.interrupt}.
```

Input: run, pause, interrupt
Output: rotate

ThreadDemo implementation in Java - class diagram

ThreadDemo creates two **ThreadPanel** displays when initialized.
ThreadPanel manages the display and control buttons, and delegates calls to **rotate()** to **DisplayThread**. **Rotator** implements the **runnable** interface.



Rotator class

```
class Rotator implements Runnable {

  public void run() {
    try {
      while(true) ThreadPanel.rotate();
    } catch(InterruptedException e) {}
  }
}
```

Rotator implements the **runnable** interface, calling **ThreadPanel.rotate()** to move the display.

run() finishes if an exception is raised by **Thread.interrupt()**.

ThreadPanel class

```
public class ThreadPanel extends Panel {

  // construct display with title and segment  color c
  public ThreadPanel(String title, Color c) {…}

  // rotate display of currently running thread 6 degrees
  // return value not used in this example
  public static boolean rotate()
         throws InterruptedException {…}

  // create a new thread with target r and start it running
  public void start(Runnable r) {
        thread = new DisplayThread(canvas,r,…);
        thread.start();
    }

  // stop the thread using Thread.interrupt()
  public void stop() {thread.interrupt();}
}
```

`ThreadPanel` manages the display and control buttons for a thread.

Calls to **`rotate()`** are delegated to **`DisplayThread`**.

Threads are created by the **`start()`** method, and terminated by the **`stop()`** method.

ThreadDemo class

```
public class ThreadDemo extends Applet {
  ThreadPanel A; ThreadPanel B;

  public void init() {
    A = new ThreadPanel("Thread A",Color.blue);
    B = new ThreadPanel("Thread B",Color.blue);
    add(A); add(B);
  }

  public void start() {
    A.start(new Rotator());
    B.start(new Rotator());
  }

  public void stop() {
    A.stop();
    B.stop();
  }
}
```

ThreadDemo creates two **ThreadPanel** displays when initialized and two threads when started.

ThreadPanel is used extensively in later demonstration programs.

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.