

Model-Based Concurrency, Part Two

Kenneth M. Anderson

University of Colorado, Boulder

CSCI 5828 — Lecture 13 — 02/23/2010

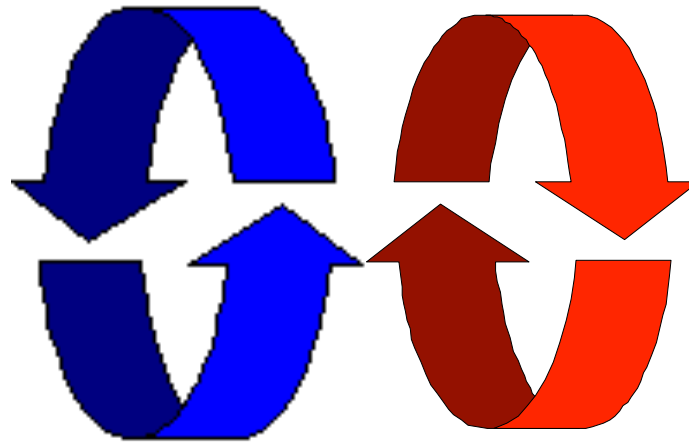
© University of Colorado, 2010

Credit Where Credit is Due

2

- ▶ The remainder of the slides in this presentation were created by Magee and Kramer for Chapter 3 of the Concurrency textbook

Concurrent Execution



Concurrency: concurrent execution

Concurrent execution

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

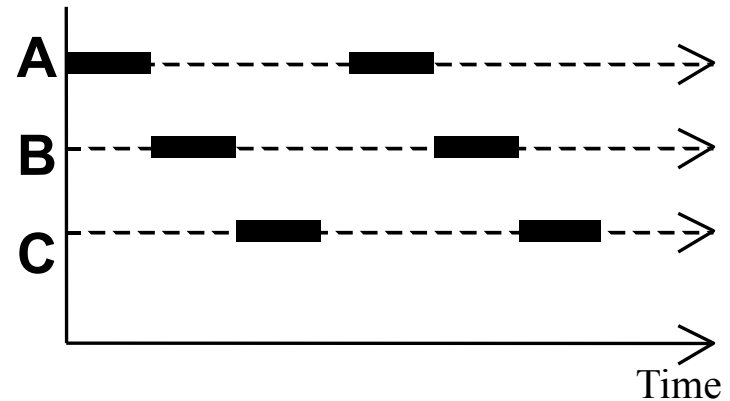
Models: **parallel composition** of asynchronous processes
- interleaving
interaction - shared actions
process labeling, and action relabeling and hiding
structure diagrams

Practice: Multithreaded Java programs

Definitions

◆ **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 Modeling 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 preservation of each process order)
- ◆ What is the result?
 - provides a general model independent 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) .
```

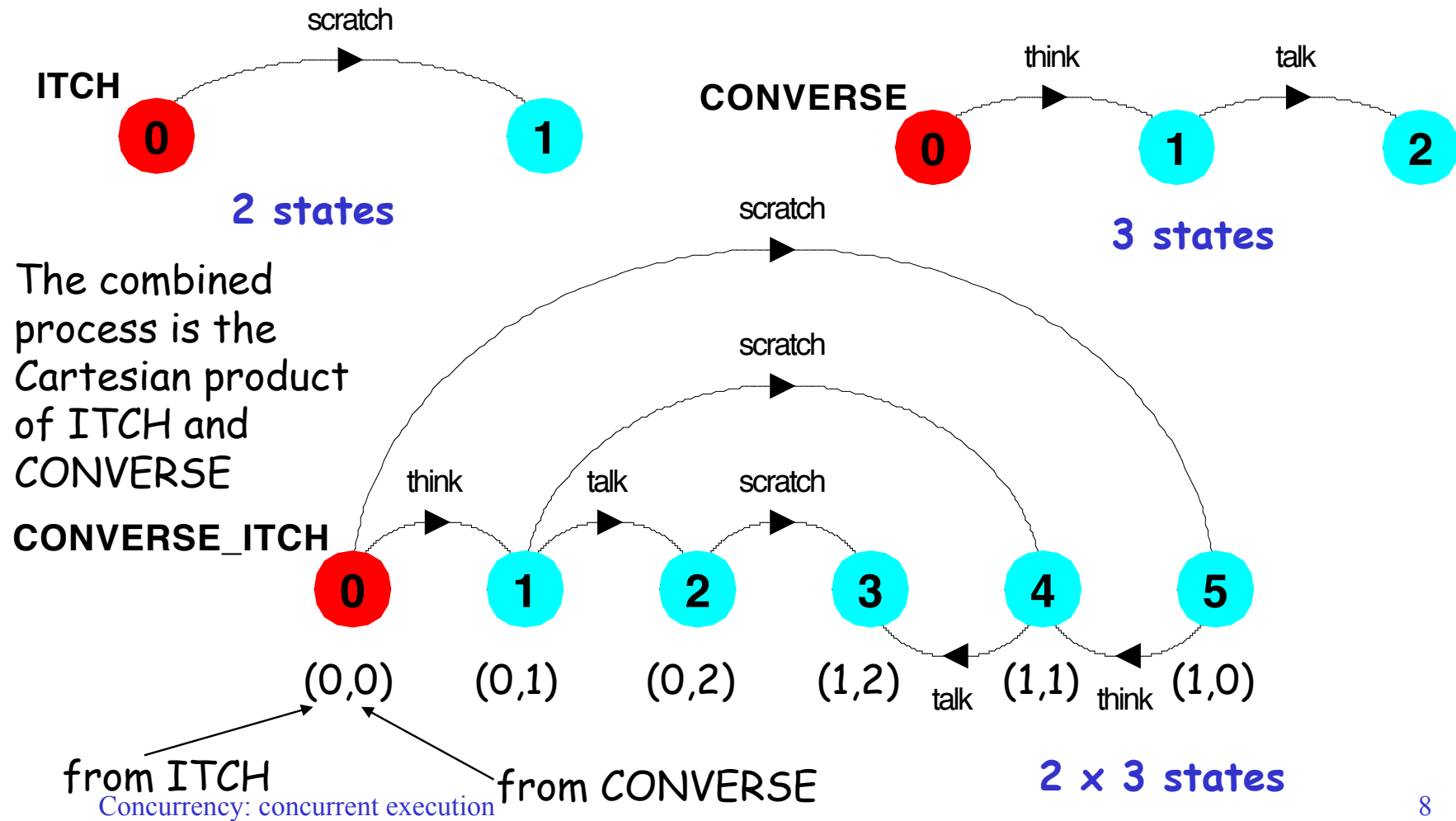
```
|| CONVERSE_ITCH = (ITCH || CONVERSE) .
```

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: $(P \parallel Q) = (Q \parallel P)$

Associative: $(P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R)$
 $= (P \parallel Q \parallel R).$

Clock radio example:

$\text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}).$

$\text{RADIO} = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}).$

$\parallel \text{CLOCK_RADIO} = (\text{CLOCK} \parallel \text{RADIO}).$

LTS? Traces? Number of states?

modeling interaction - 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 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.

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

MAKER
synchronizes
with USER
when **ready**.

LTS? Traces? Number of states?

Concurrency: concurrent execution

Non-disjoint
alphabets

shared alphabets affect the Cartesian product

- ◆ BILL = (play \rightarrow meet \rightarrow STOP).
- ◆ BEN = (work \rightarrow meet \rightarrow STOP).
- ◆ Each process has three states (initial, after first action, after second action)
- ◆ Cartesian product should produce 9 states (3 x 3)
- ◆ But LTS contains only 5 states! Why?
- ◆ Due to rules governing shared actions
- ◆ Full Cartesian Product: (initial, initial), (initial, work), (initial, meet), (play, initial), (play, work), (play, meet), (meet, initial), (meet, work), (meet, meet)
- ◆ But due to rules governing shared actions, the red tuples are not permitted;
Both processes must be ready to transition to the state after the “meet” action at the same time

modeling interaction - handshake

A handshake is an action acknowledged by another:

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

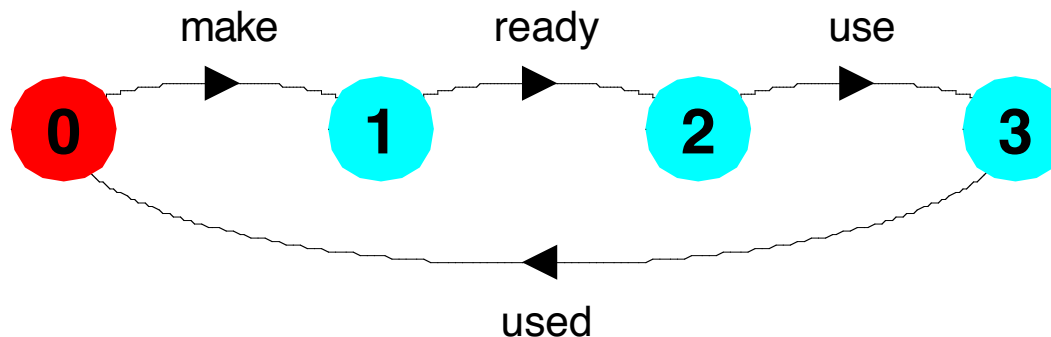
```
USERv2 = (ready->use->used->USERv2) .
```

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

3 states

3 states

3 × 3
states?



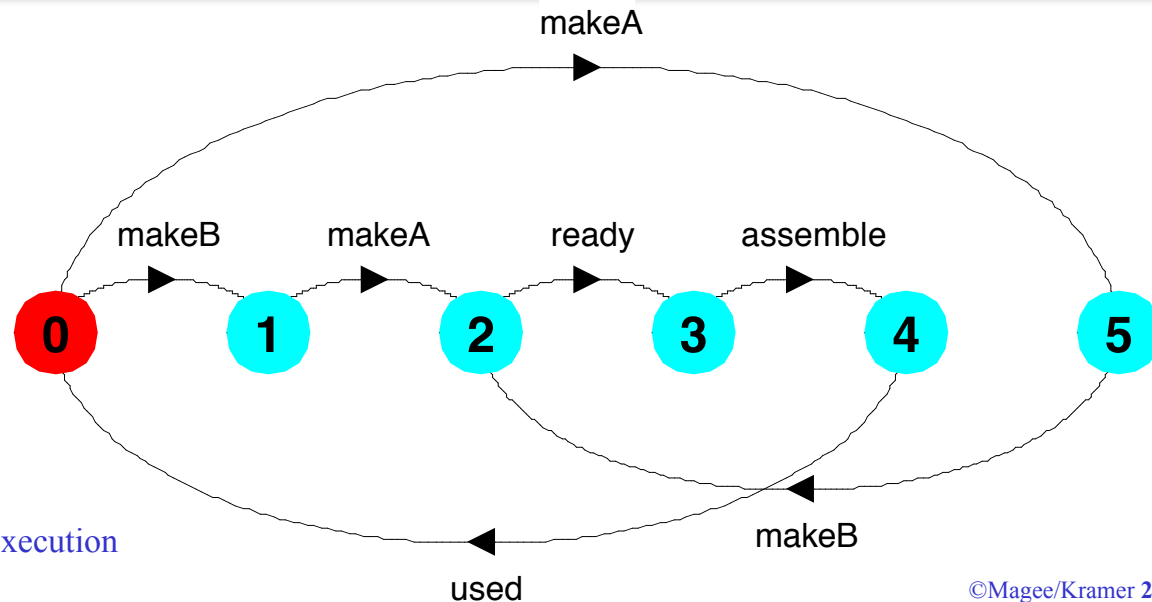
4 states

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) .
```



Concurrency: concurrent execution

13

composite processes

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

$$|| \text{MAKERS} = (\text{MAKE_A} || \text{MAKE_B}) .$$
$$|| \text{FACTORY} = (\text{MAKERS} || \text{ASSEMBLE}) .$$

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.

$$|| \text{FACTORY} = (\text{MAKE_A} || \text{MAKE_B} || \text{ASSEMBLE}) .$$

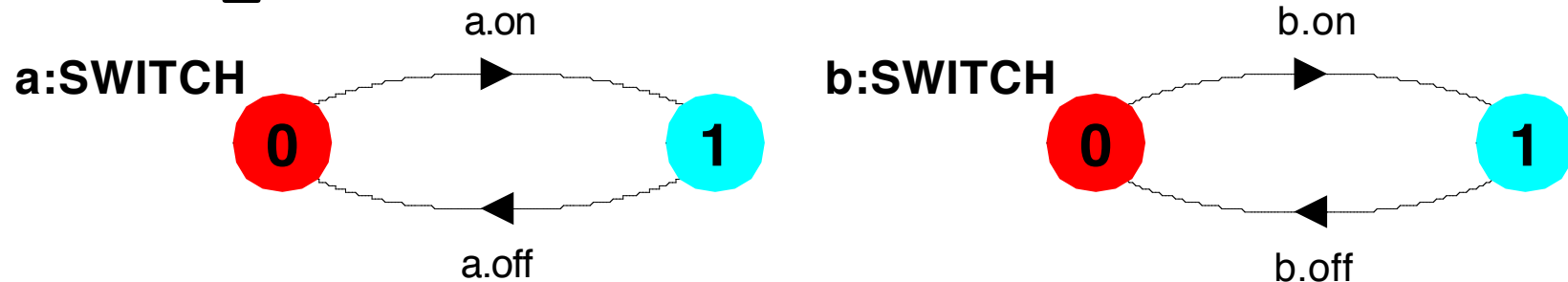
process instances and labeling

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

Two **instances** of a switch process:

$\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}) .$

$|| \text{TWO_SWITCH} = (a:\text{SWITCH} || b:\text{SWITCH}) .$



An array of **instances** of the switch process:

$|| \text{SWITCHES } (N=3) = (\text{forall}[i:1..N] \text{ } s[i]:\text{SWITCH}) .$

$|| \text{SWITCHES } (N=3) = (s[i:1..N]:\text{SWITCH}) .$

process labeling by a set of prefix labels

$\{a_1, \dots, a_x\} :: P$ replaces every action label n in the alphabet of P with the labels $a_1.n, \dots, a_x.n$. Further, every transition $(n \rightarrow X)$ in the definition of P is replaced with the transitions $(\{a_1.n, \dots, a_x.n\} \rightarrow X)$.

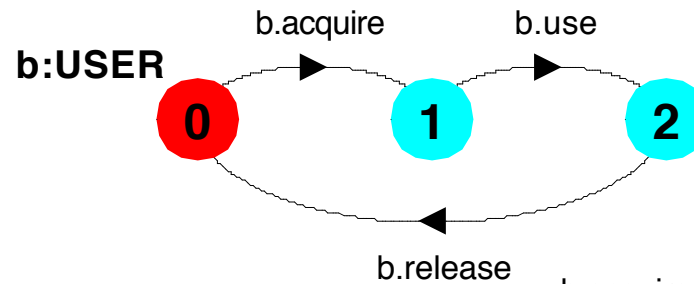
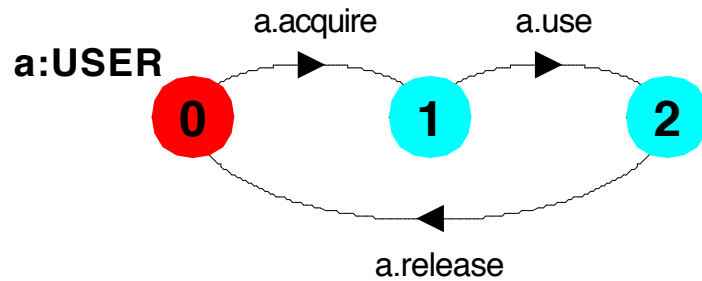
Process prefixing is useful for modeling **shared** resources:

$\text{RESOURCE} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}) .$

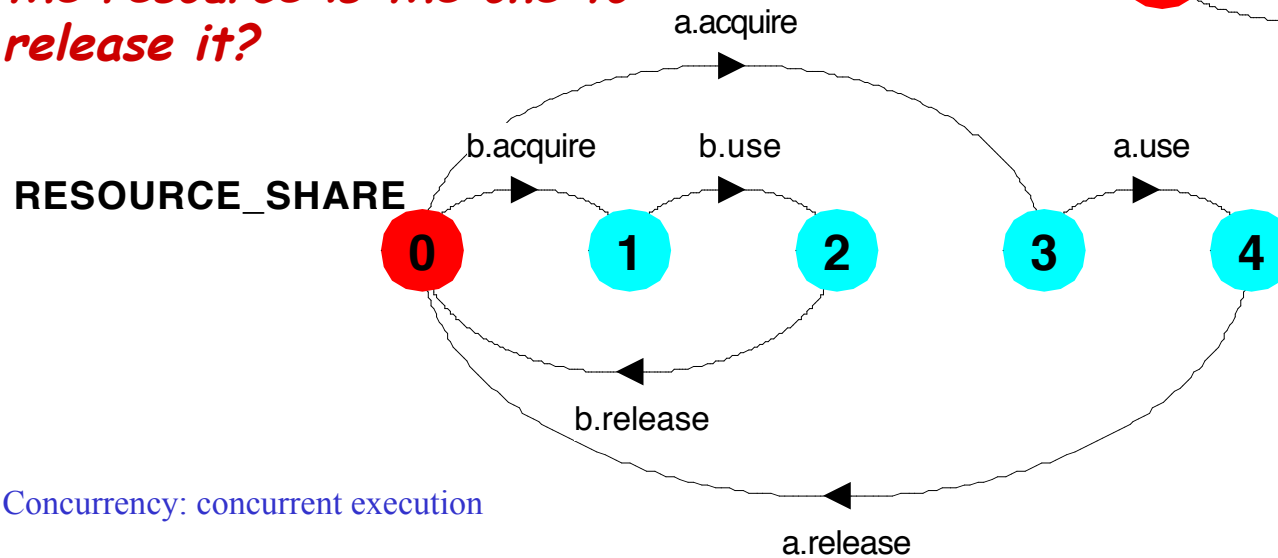
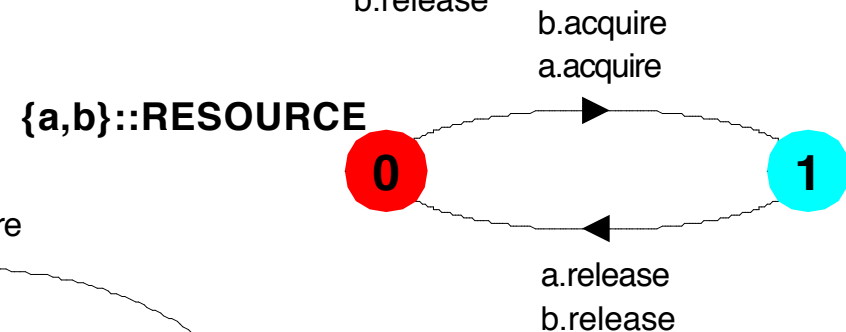
$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) .$

$|| \text{RESOURCE_SHARE} = (\text{a} : \text{USER} \quad || \quad \text{b} : \text{USER} \\ || \quad \{\text{a}, \text{b}\} :: \text{RESOURCE}) .$

process prefix labels for shared resources



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



Concurrency: concurrent execution

17

action relabeling

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

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

Relabeling to ensure that composed processes synchronize on particular actions.

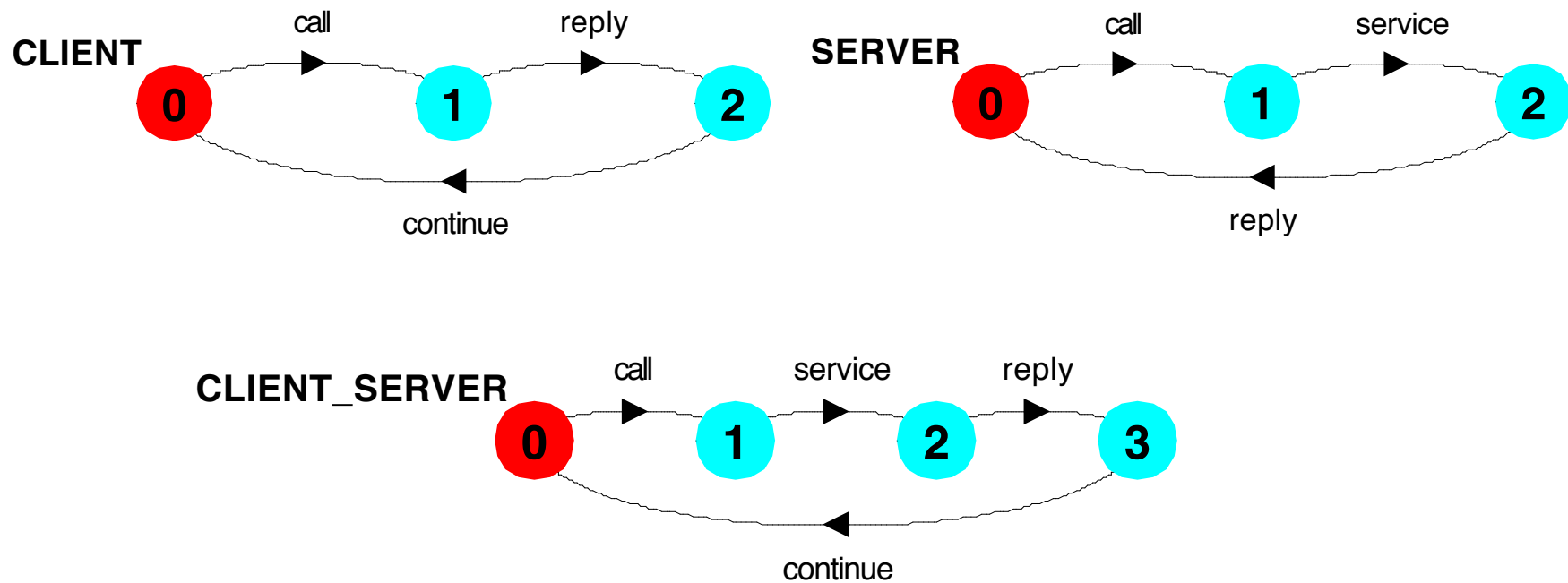
CLIENT = (**call**->**wait**->continue->CLIENT) .

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

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

action relabeling

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



action relabeling - 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

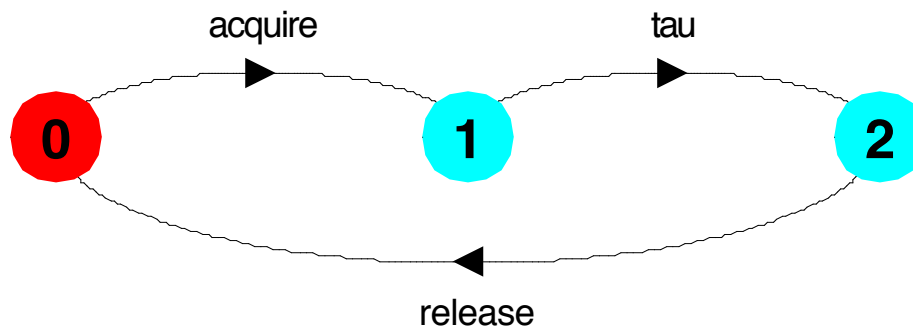
When applied to a process P , the hiding operator $\backslash\{a1..ax\}$ removes the action names $a1..ax$ from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled τ . Silent actions in different processes are not shared.

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

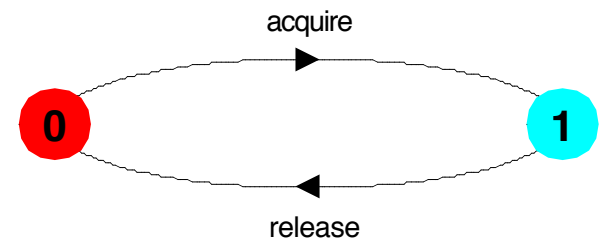
When applied to a process P , the interface operator $@\{a1..ax\}$ hides all actions in the alphabet of P not labeled in the set $a1..ax$.

action hiding

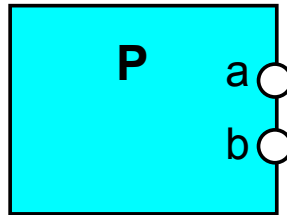
The following definitions are equivalent:

$$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \setminus \{\text{use}\}.$$
$$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) @ \{\text{acquire}, \text{release}\}.$$


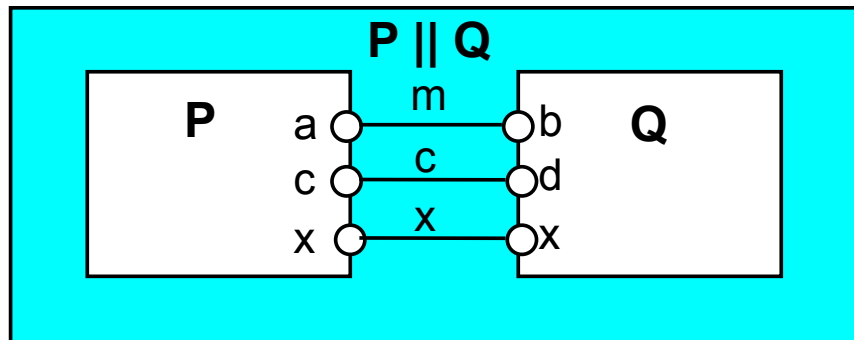
Minimization removes hidden tau actions to produce an LTS with equivalent observable behavior.



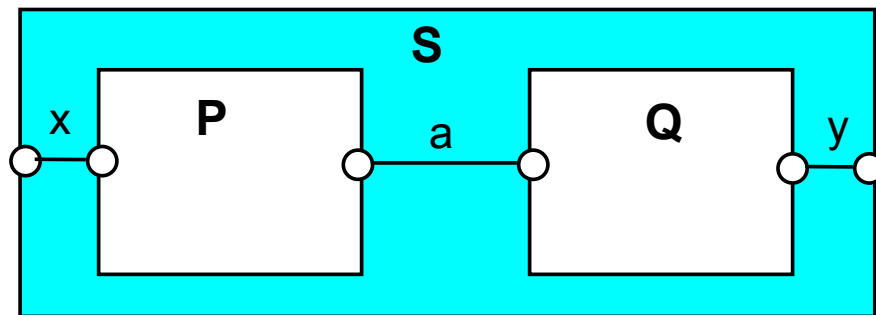
structure diagrams – systems as interacting processes



Process *P* with
alphabet $\{a,b\}$.



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



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

Concurrency: concurrent execution

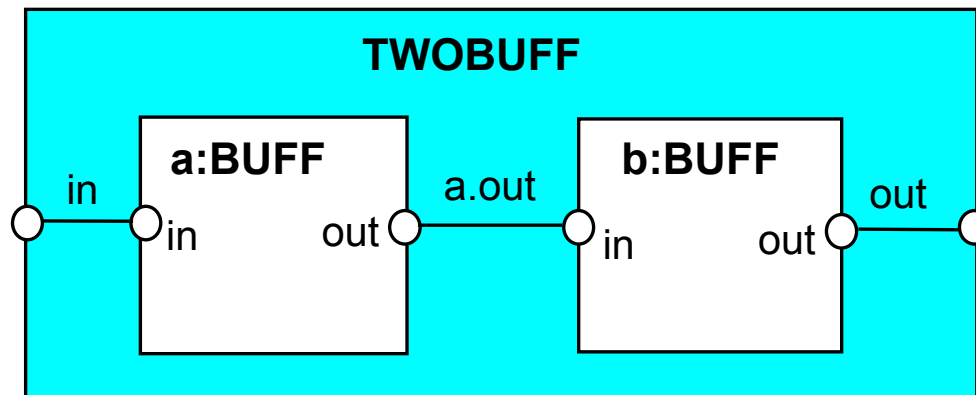
structure diagrams

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

range $T = 0..3$

$\text{BUFF} = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF}) .$

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



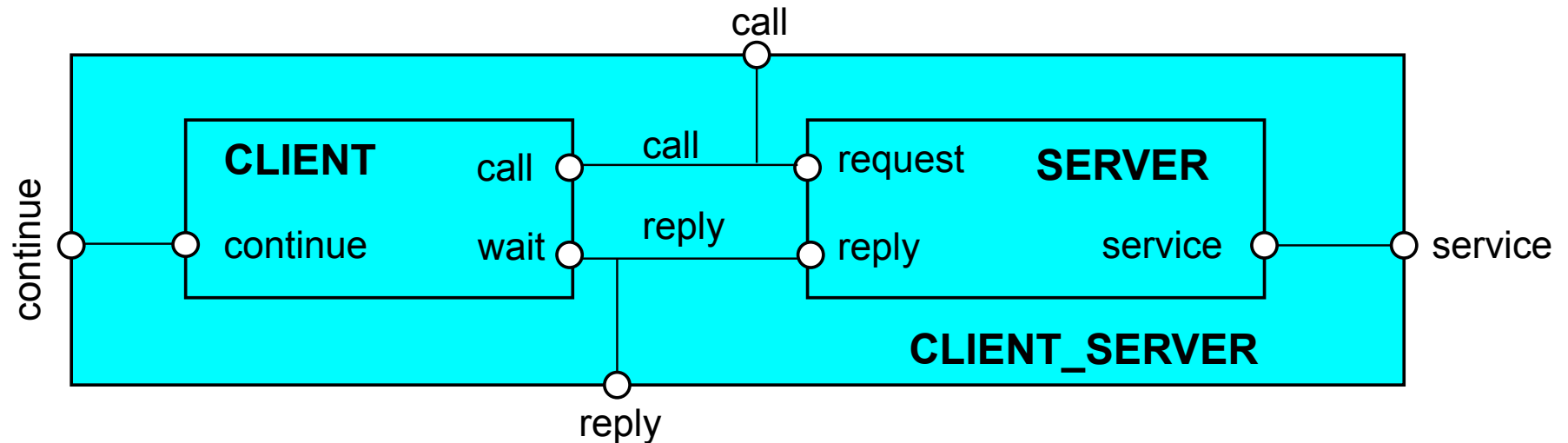
structure diagrams

Structure diagram for CLIENT_SERVER ?

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

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

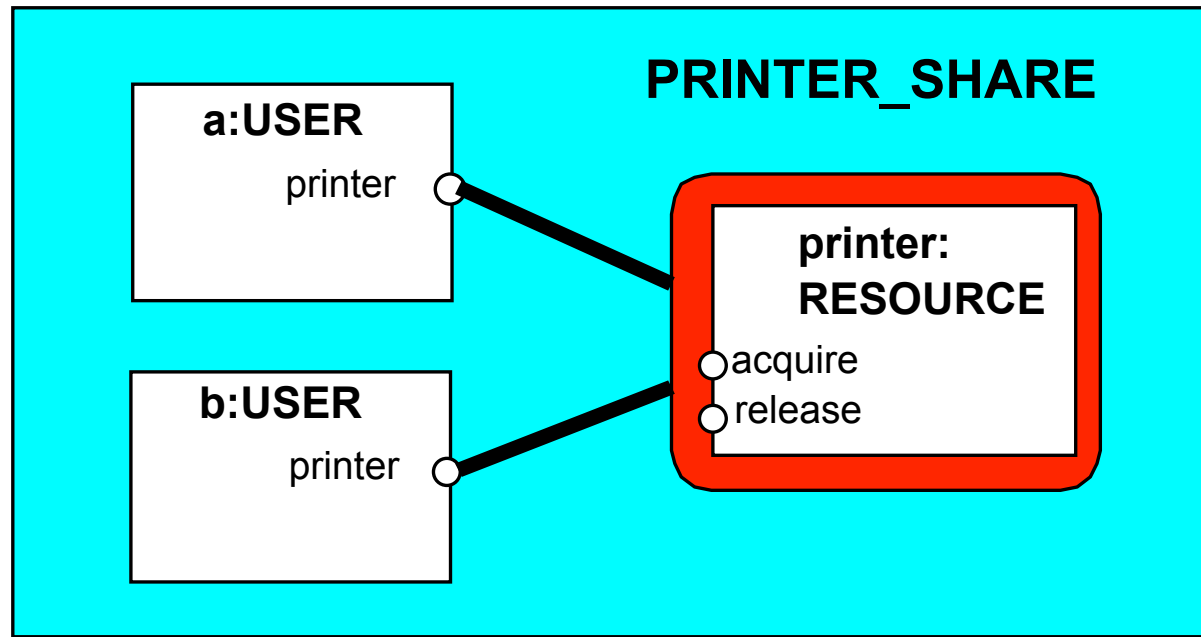
|| CLIENT_SERVER = (CLIENT || SERVER)
/ {call/request, reply/wait} .



Concurrency: concurrent execution

25

structure diagrams - resource sharing



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

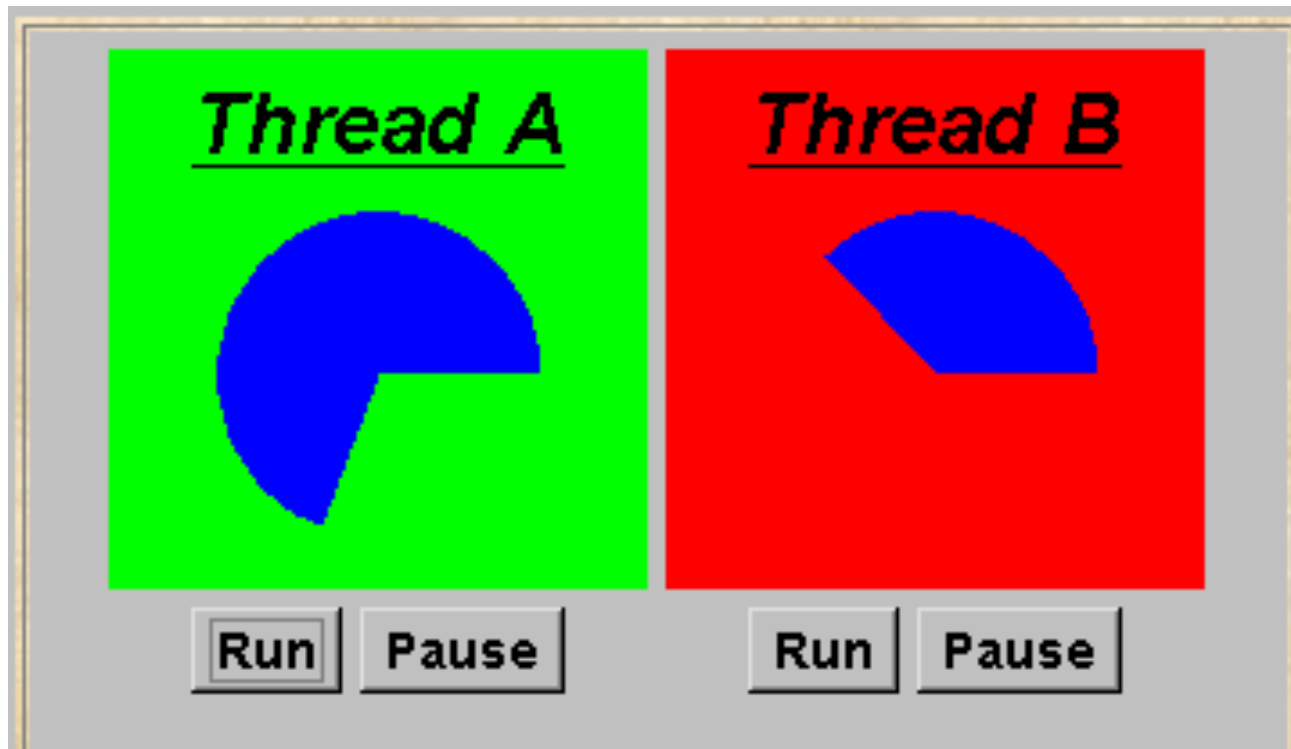
```
||PRINTER_SHARE
= (a:USER||b:USER||{a,b}::printer:RESOURCE) .
```

Concurrency: concurrent execution

26

3.2 Multi-threaded Programs in Java

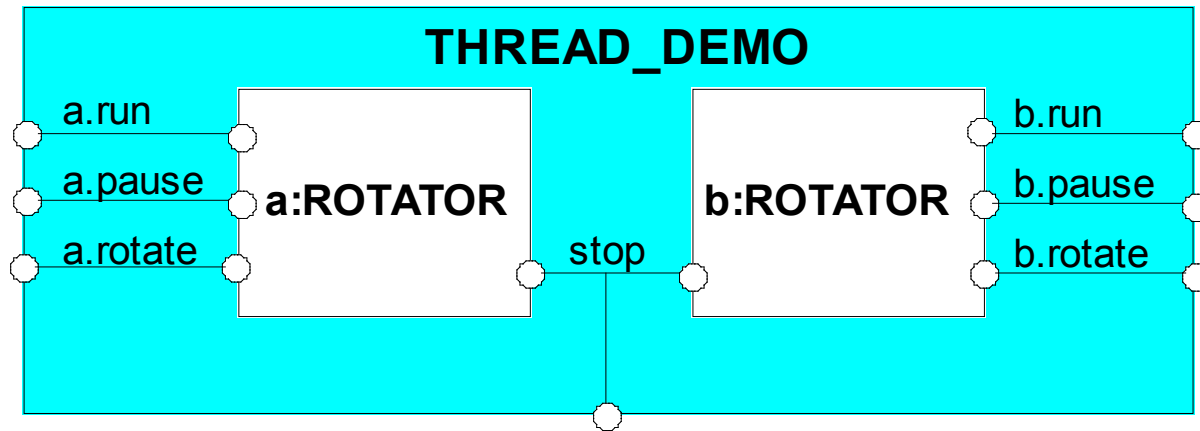
Concurrency in Java occurs when more than one thread is alive. ThreadDemo has two threads which rotate displays.



Concurrency: concurrent execution

27

ThreadDemo model



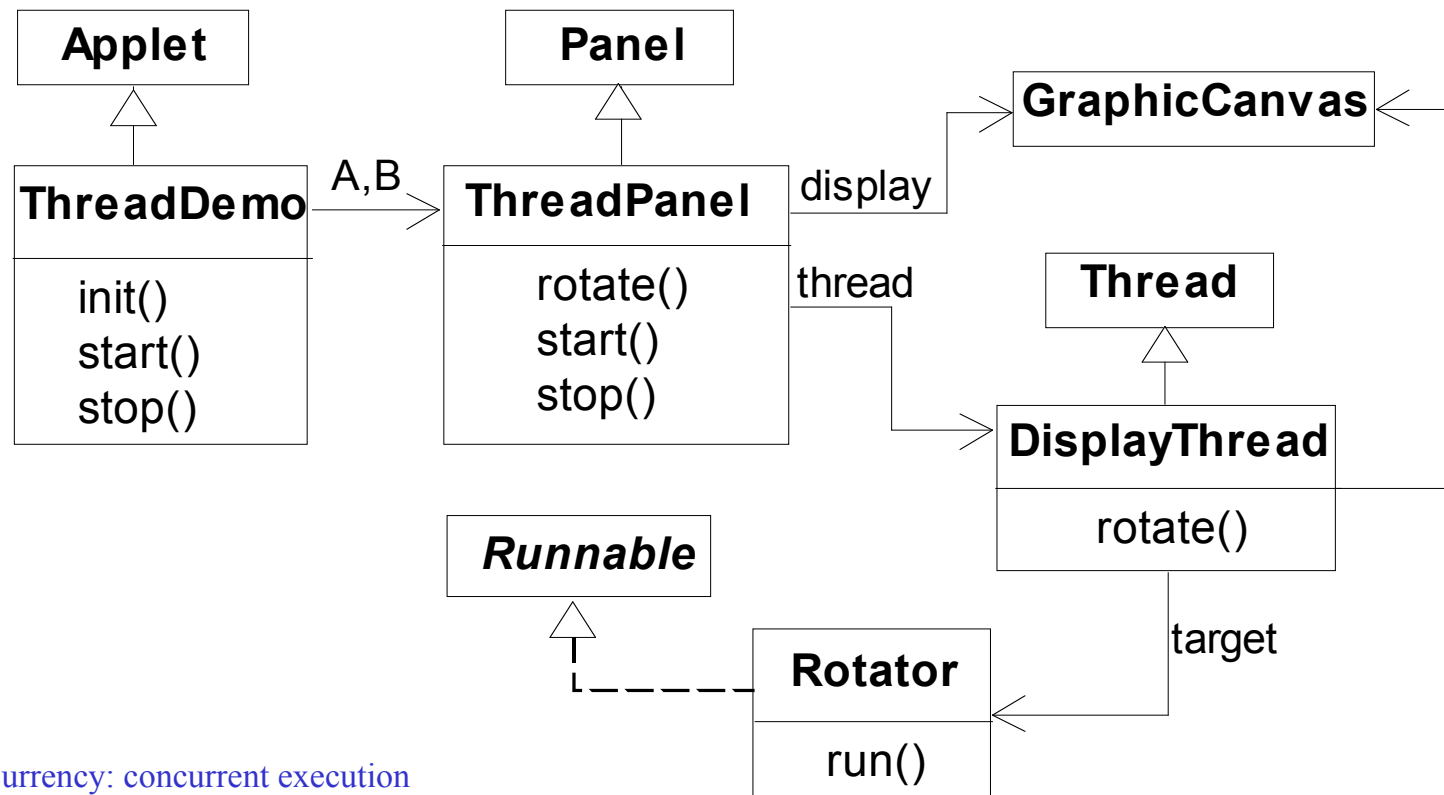
```
ROTATOR = PAUSED ,
PAUSED  = (run->RUN | pause->PAUSED
           | interrupt->STOP) ,
RUN      = (pause->PAUSED | {run,rotate}->RUN
           | interrupt->STOP) .

||THREAD_DEMO = (a:ROTATOR || b:ROTATOR)
/{stop/{a,b}.interrupt}.
```

*Interpret
run,
pause,
interrupt
as inputs,
rotate as
an output.*

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 labeling and action relabeling and hiding.
- **Structure diagrams**

◆ Practice

- **Multiple threads in Java.**

Concurrency: concurrent execution