Introduction to modelling in FSP (Finite State Processes) and its application for analysis of Java Mutex problems

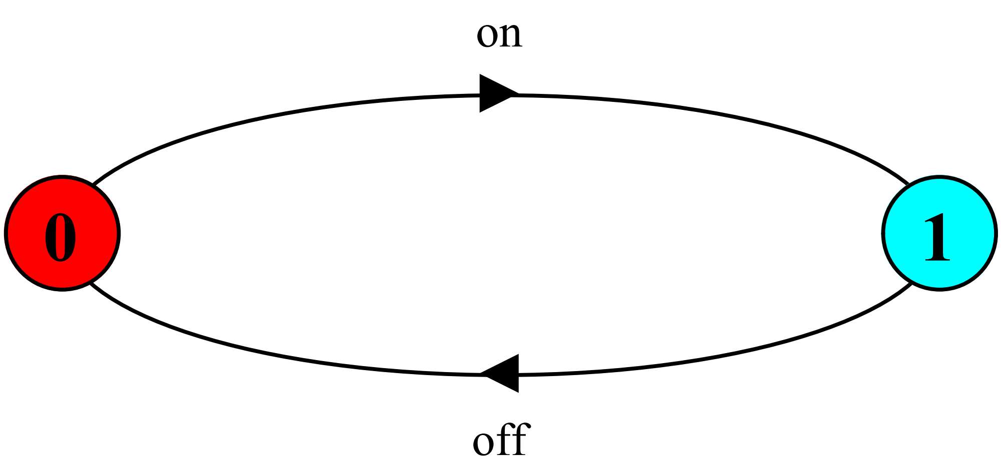
Modeling Processes

Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the ***LTSA*** analysis tool.

- LTS - graphical form
- FSP - algebraic (textual) form

modeling processes

A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.



a light switch
LTS

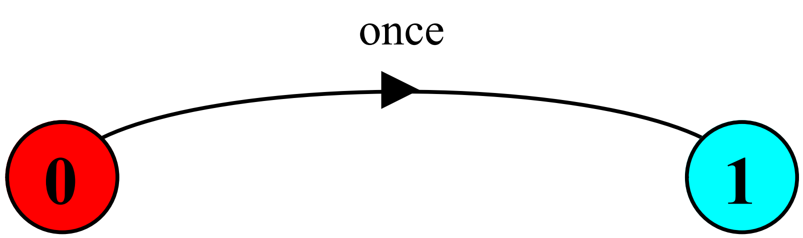
`on→off→on→off→on→off→ ……….`

a sequence of actions or *trace*

FSP - action prefix

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

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



ONESHOT state machine

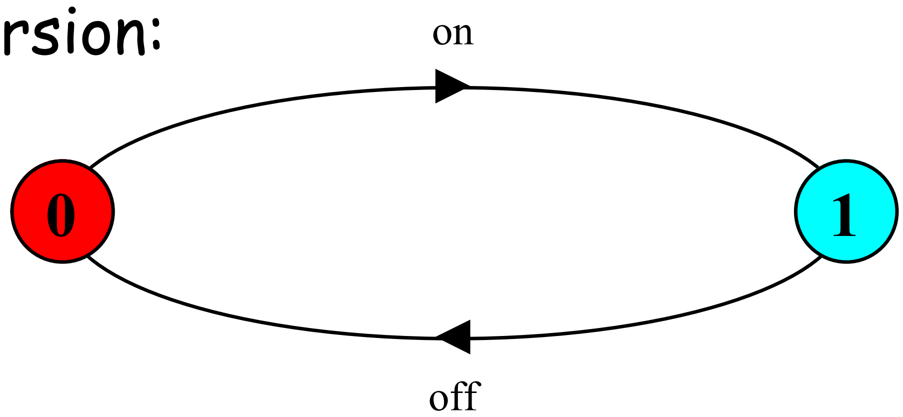
(terminating process)

Convention: actions begin with lowercase letters
PROCESSES begin with uppercase letters

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```



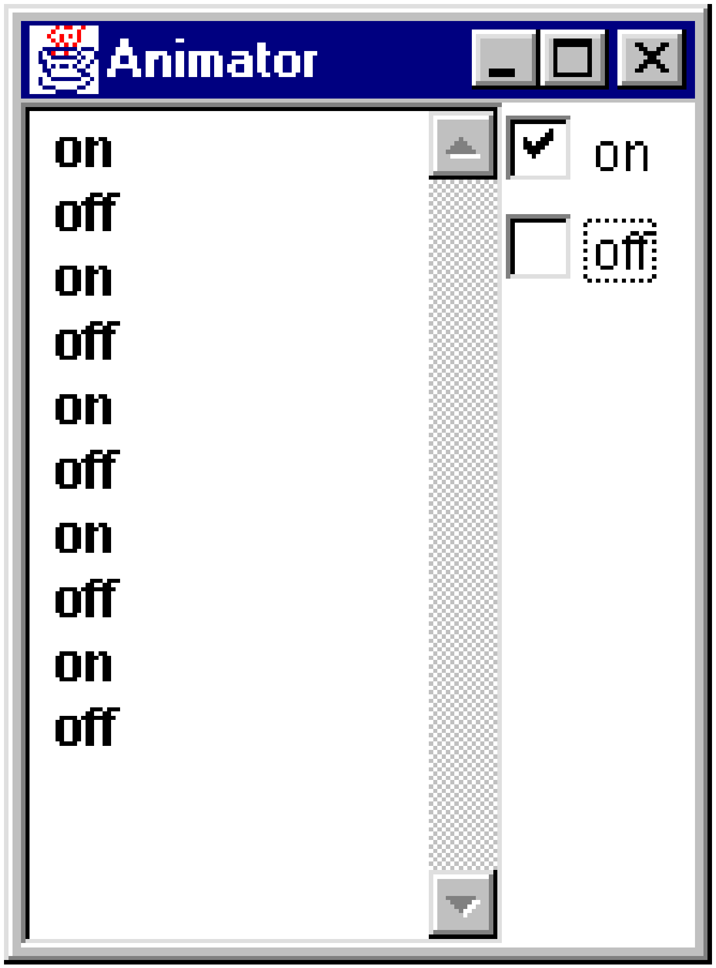
Substituting to get a more succinct definition:

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

And again:

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

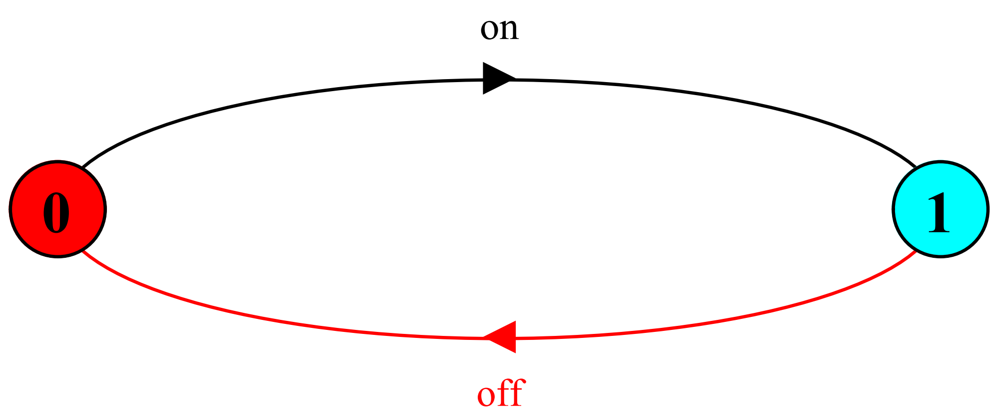
animation using LTSA



The *LTSA* animator can be used to produce a trace.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.

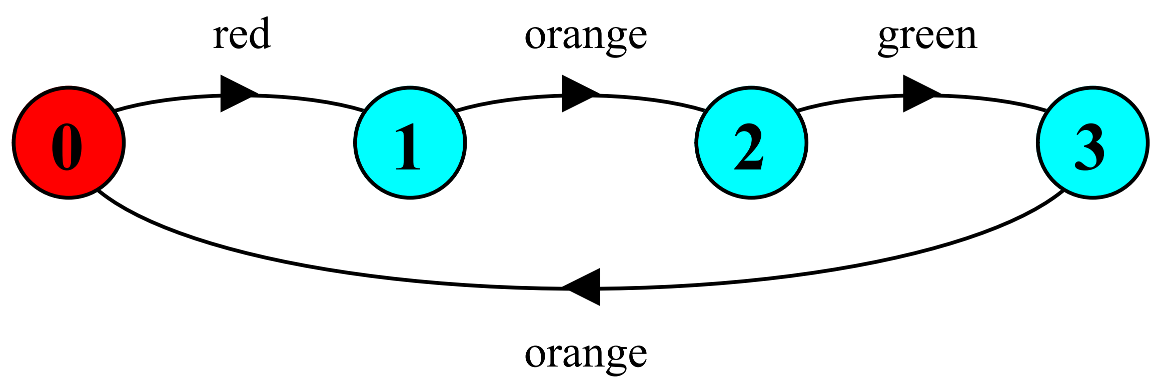


FSP - action prefix

FSP model of a traffic light :

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

LTS generated using *LTSA*:



Trace:

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

FSP - choice

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

Who or what makes the choice?

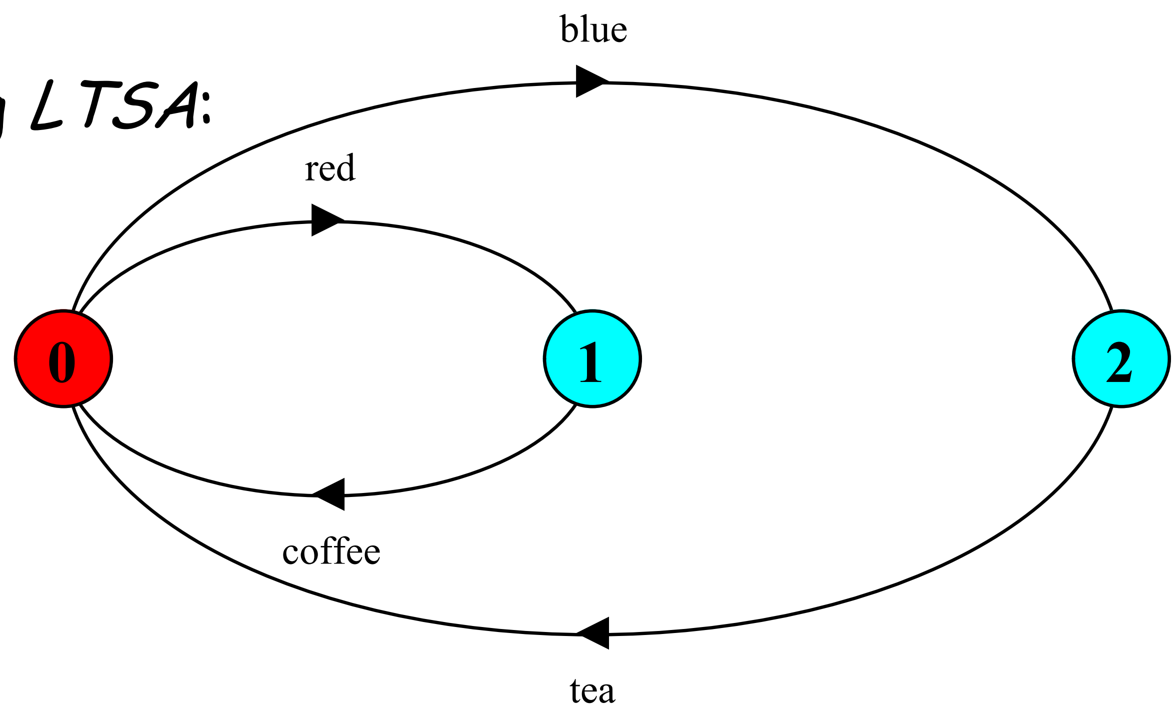
Is there a difference between input and output actions?

FSP - choice

FSP model of a drinks machine :

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

LTS generated using *LTSA*:



Possible traces?

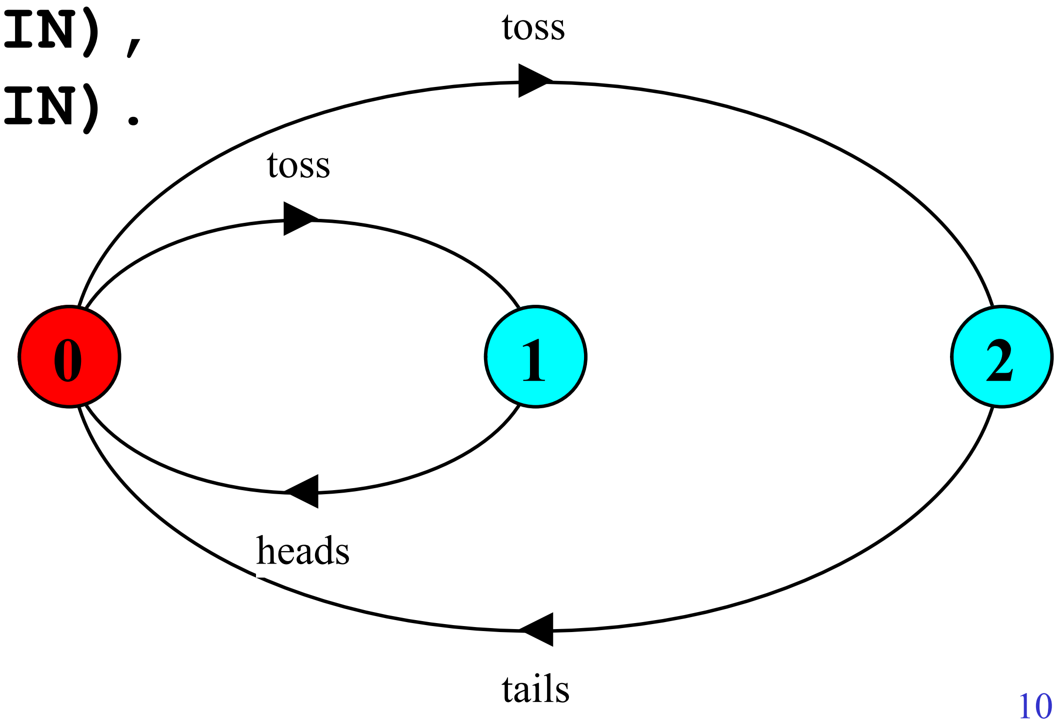
Non-deterministic choice

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

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

Tossing a coin.

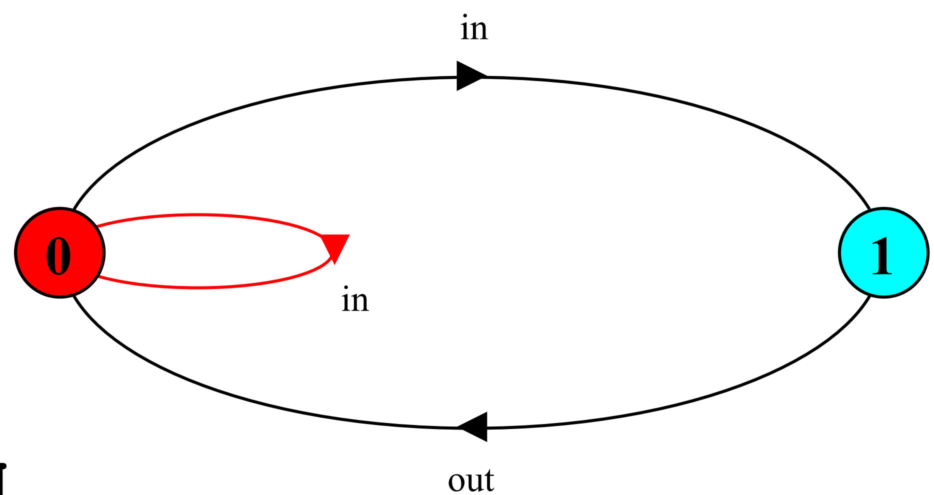
Possible traces?



Modeling failure

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

Use non-determinism...




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

FSP - indexed processes and actions

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

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

equivalent to

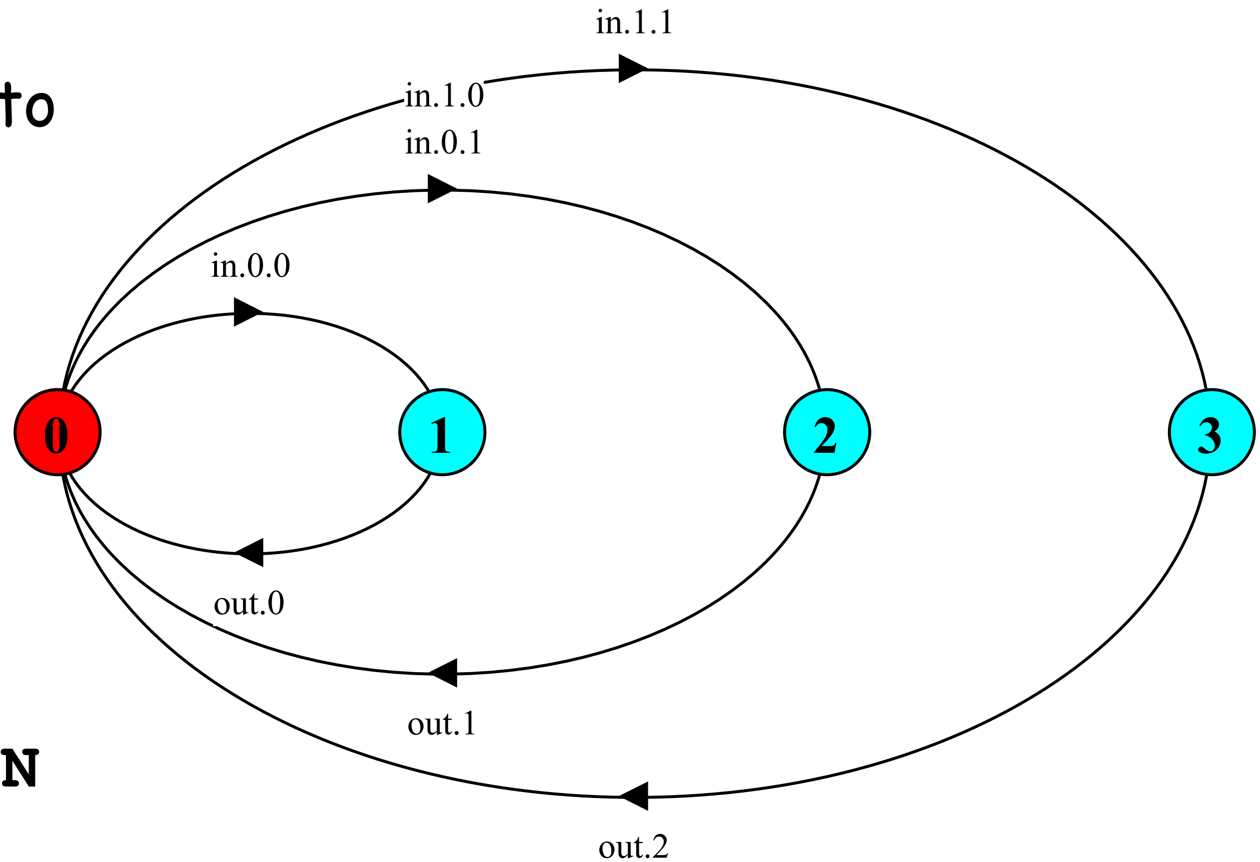
```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

or using a process parameter with default value:

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

FSP - constant & range declaration

index expressions to model calculation:

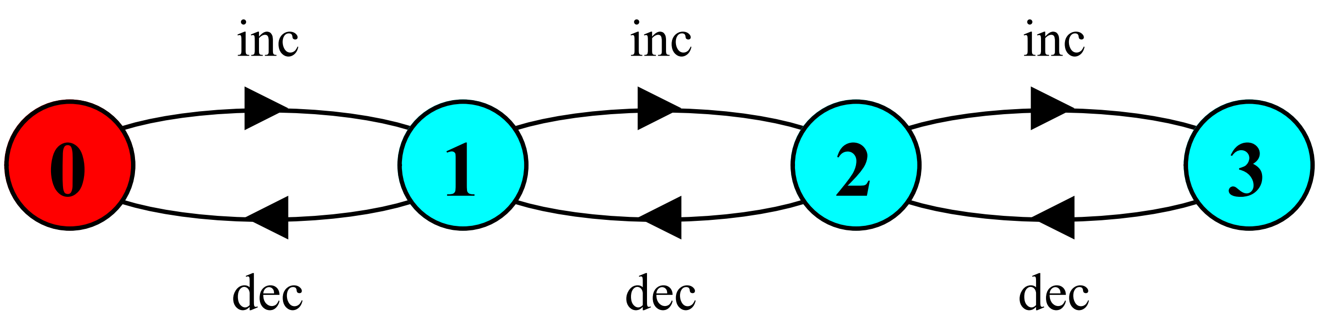



```
const N = 1
range T = 0..N
range R = 0..2*N

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

FSP - guarded actions

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

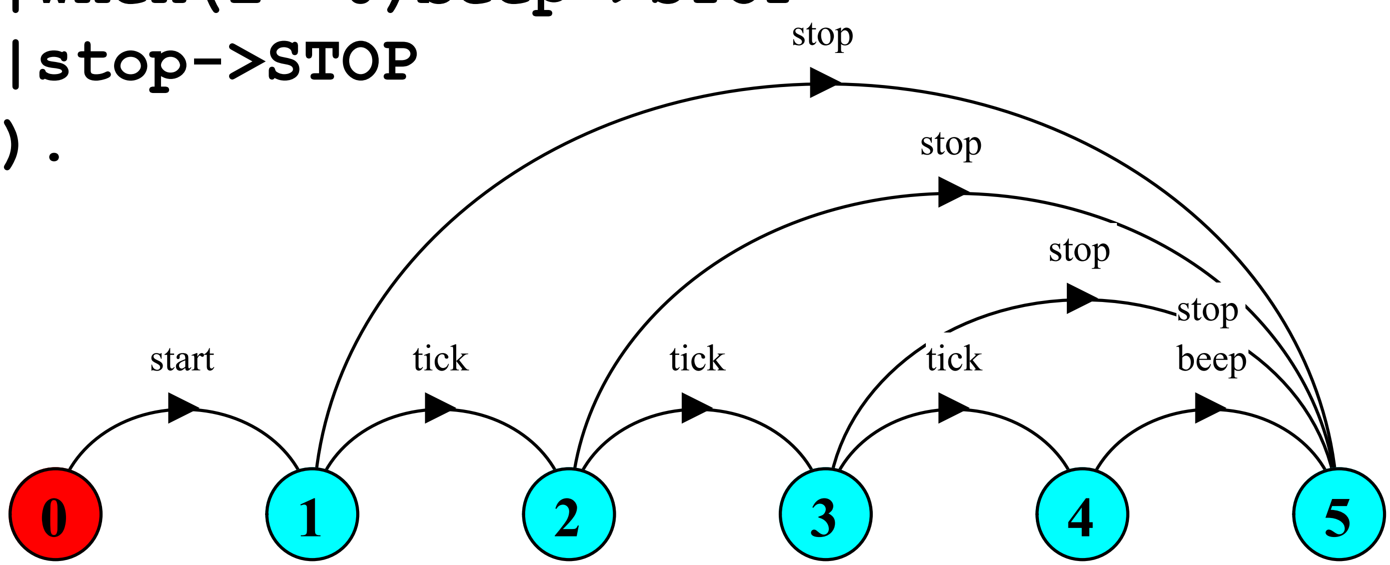
```
COUNT (N=3)      = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```



FSP - guarded actions

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

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



FSP - process alphabets

The alphabet of a process is the set of actions in which it can engage.

Alphabet extension can be used to extend the **implicit** alphabet of a process:

```
WRITER = (write[1]->write[3]->WRITER)
              +{write[0..3]}.
```

Alphabet of **WRITER** is the set **{write[0..3]}**

(we make use of alphabet extensions in later chapters)

parallel composition - action interleaving

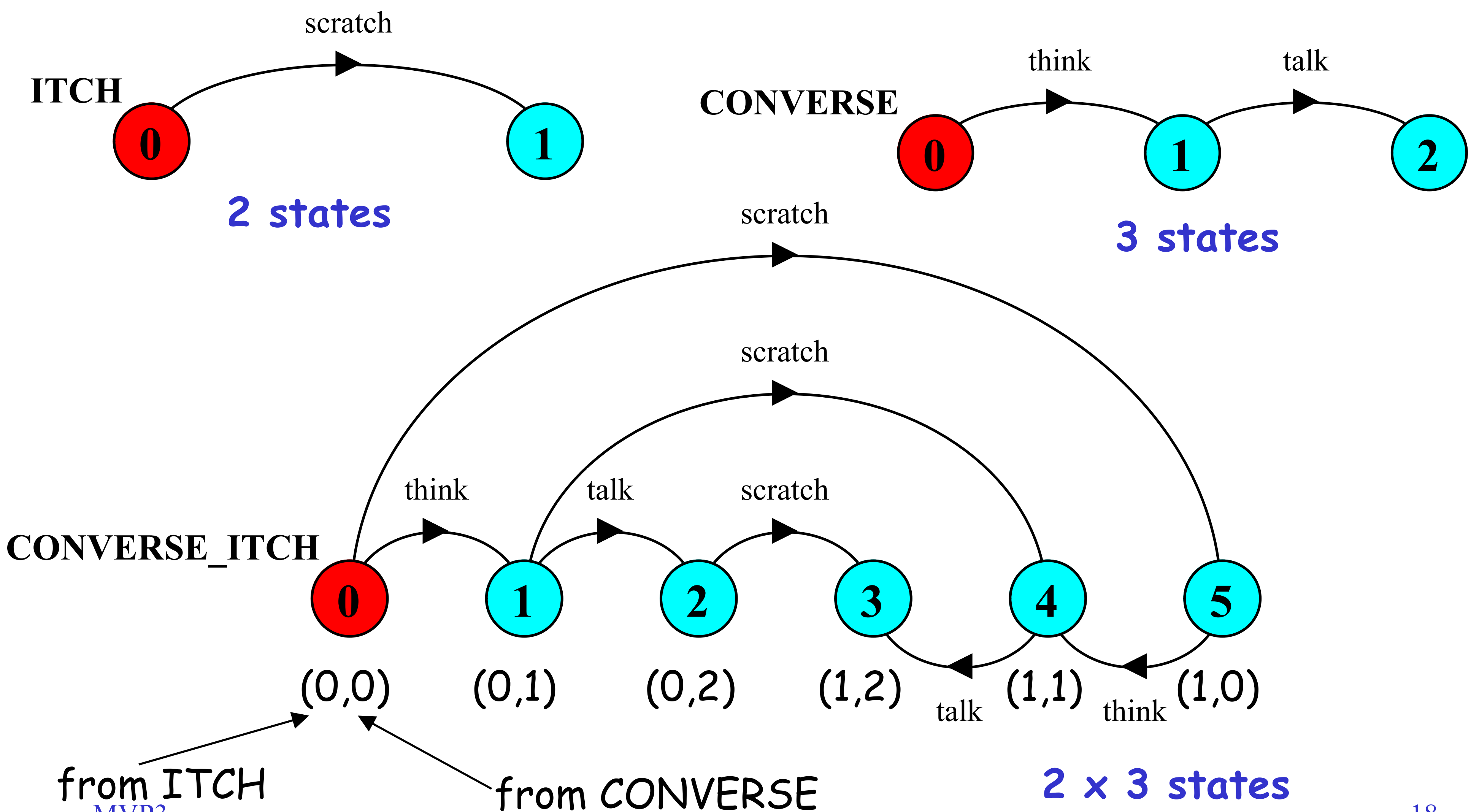
If P and Q are processes then (P||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

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

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

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).
```

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?

modeling interaction - handshake

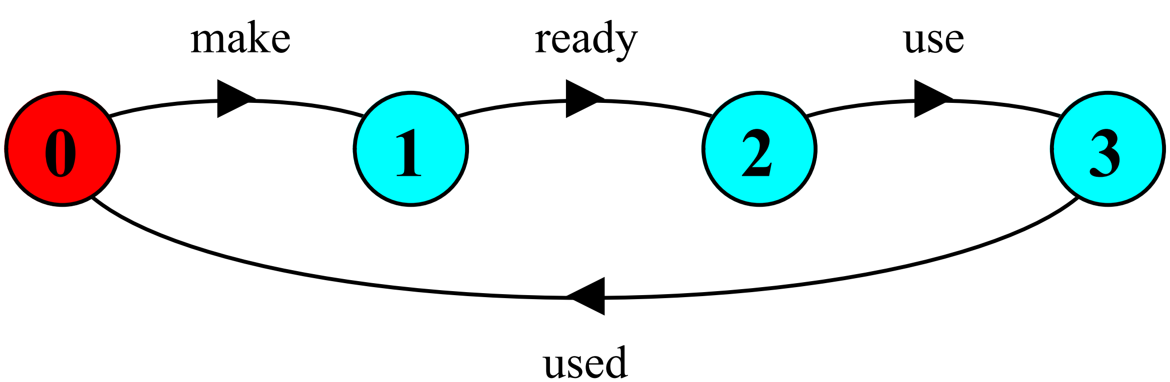
A handshake is an action acknowledged by another:

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

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

3 states

3 states

3 x 3 states?



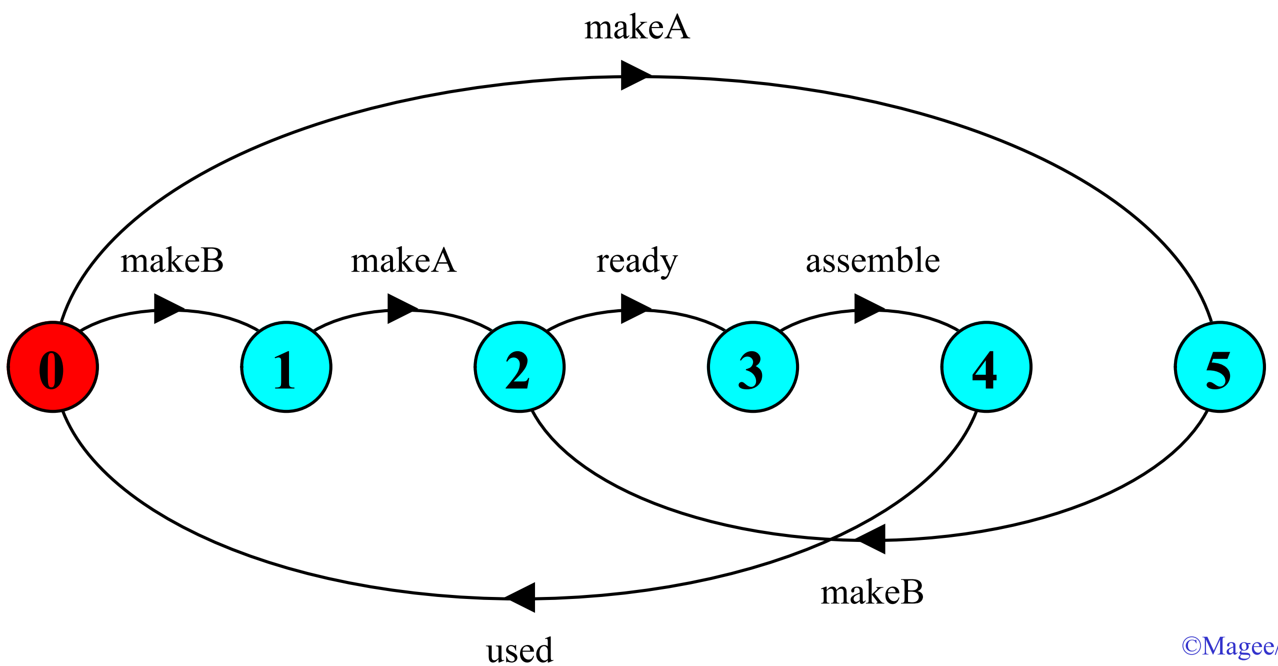
4 states

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.

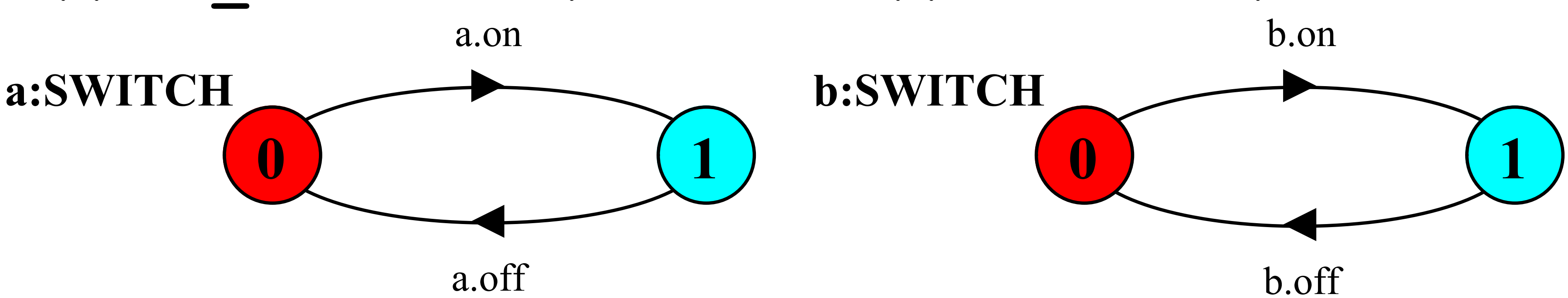
```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

process 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

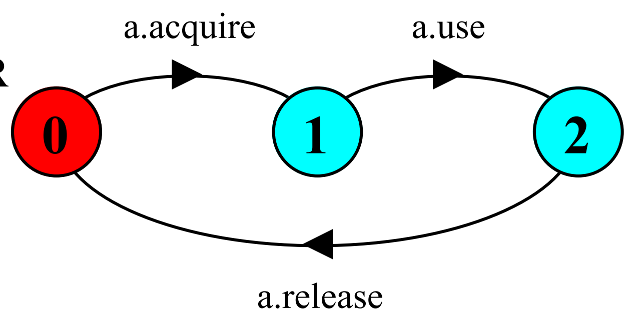
{a1,..,ax}::P replaces every action label n in the alphabet of P with the labels a1.n,…,ax.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,…,ax.n} ->X).

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

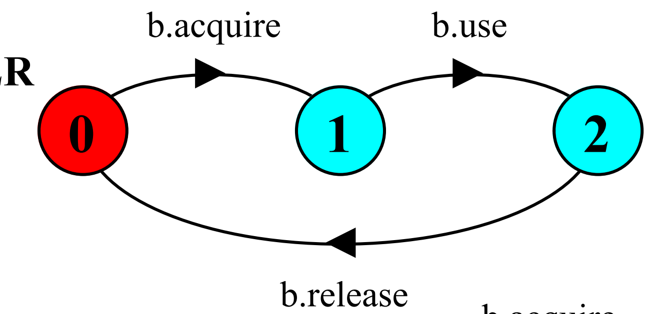
```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

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

process prefix labels for shared resources

a:USER

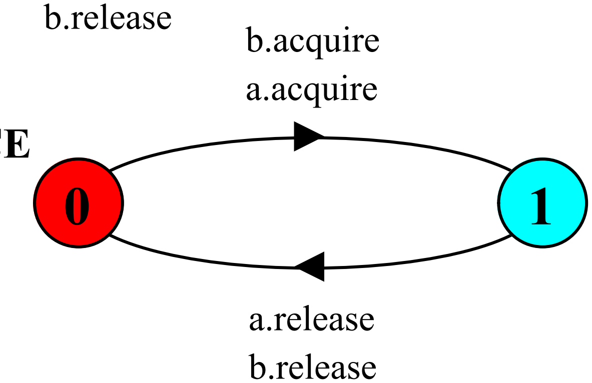


b:USER

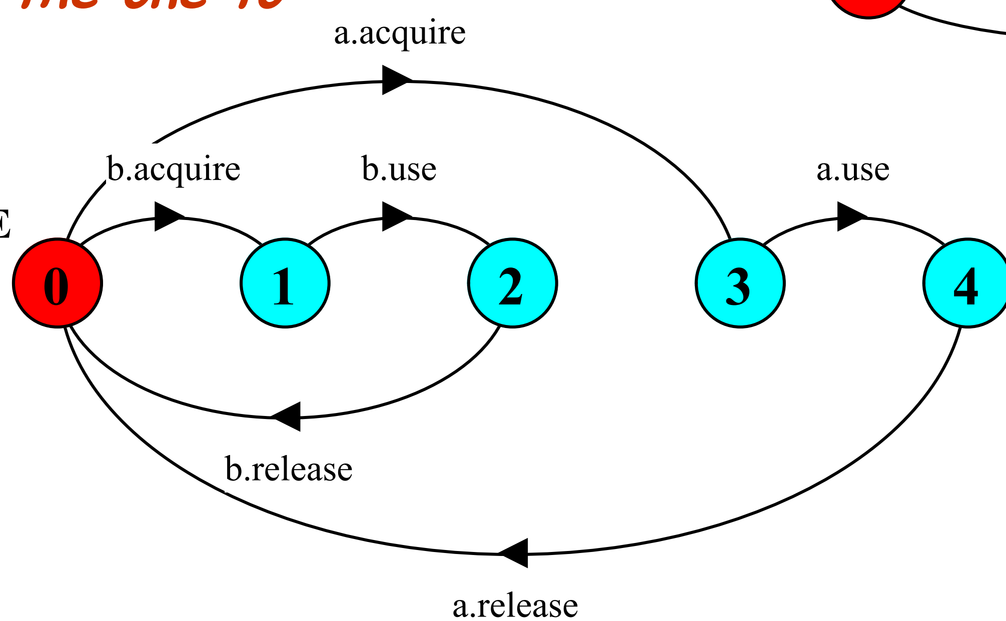


How does the model ensure that the user that acquires the resource is the one to release it?

{a,b}::RESOURCE



RESOURCE_SHARE



action relabeling

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

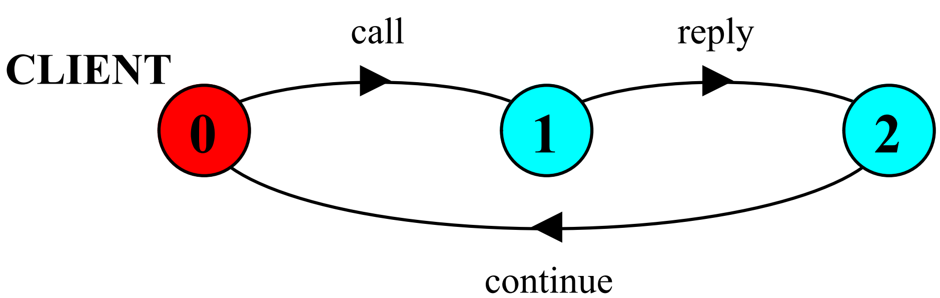
/{*newlabel_1*/*oldlabel_1*,… *newlabel_n*/*oldlabel_n*}.

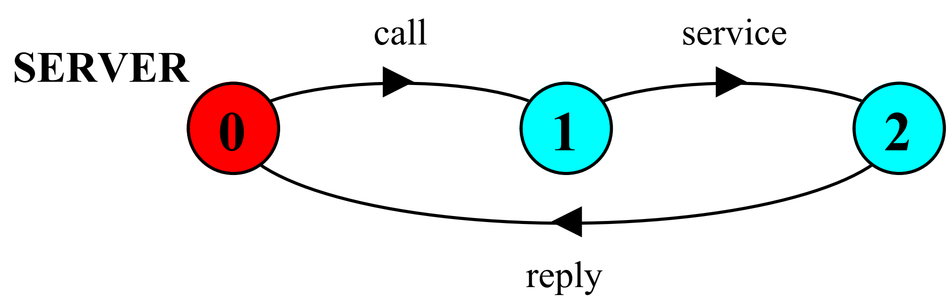
Relabeling to ensure that composed processes synchronize on particular actions.

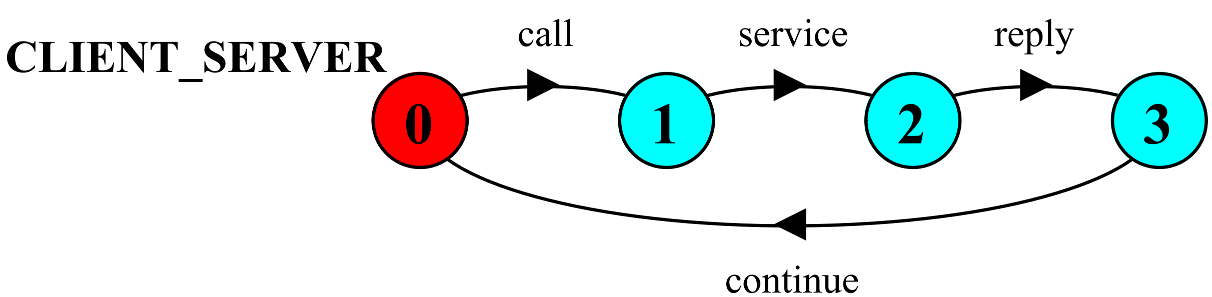
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

action relabeling

```
||CLIENT_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 \{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.

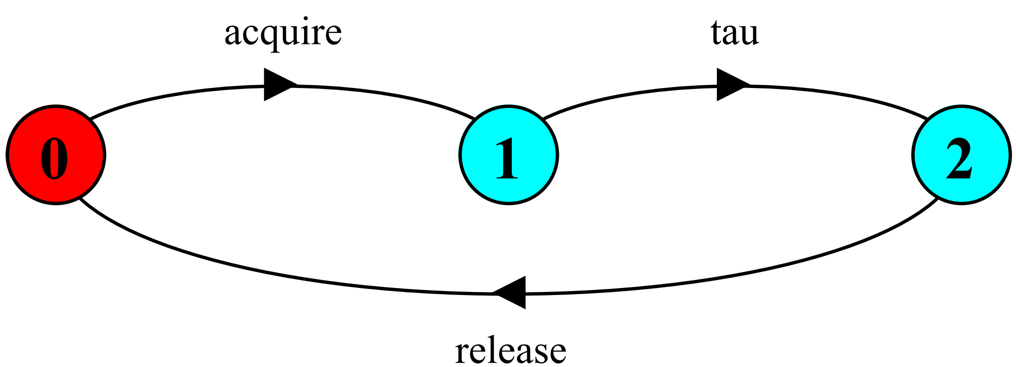
Sometimes it is more convenient to specify the set of labels to be exposed....

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.

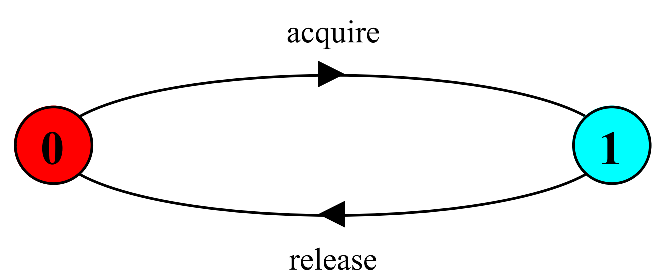
action hiding

The following definitions are equivalent:

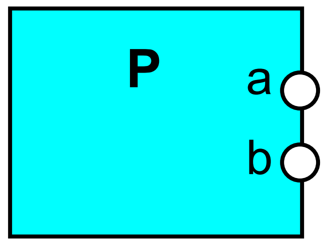
```
USER = (acquire->use->release->USER)
       \{use}.

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



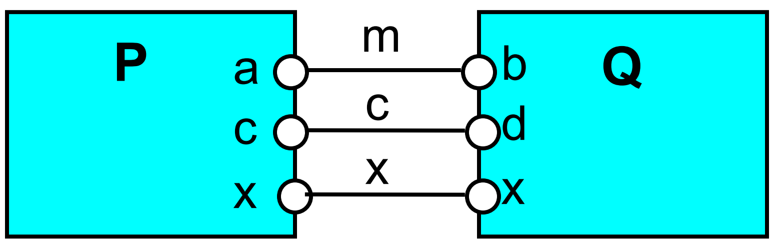
Minimization removes hidden **tau** actions to produce an LTS with equivalent observable behavior.



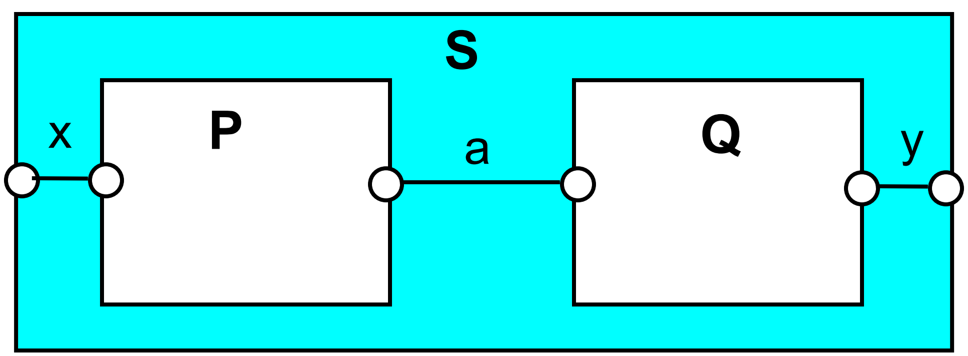
structure diagrams



Process P with alphabet {a,b}.



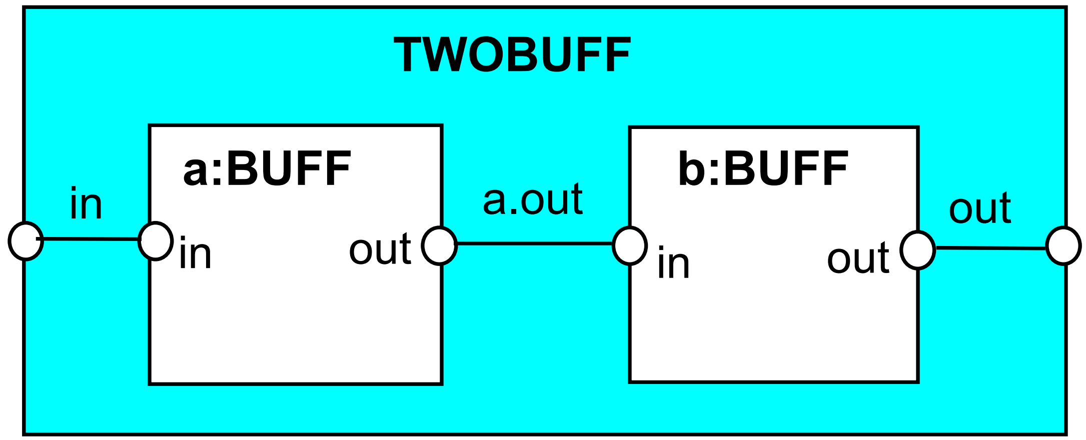
Parallel Composition
(P||Q) / {m/a,m/b,c/d}



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

structure diagrams

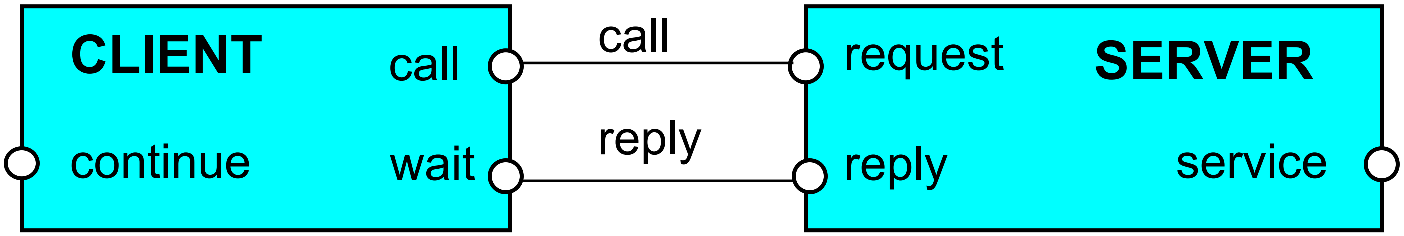
We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition*, *relabeling* and *hiding*.



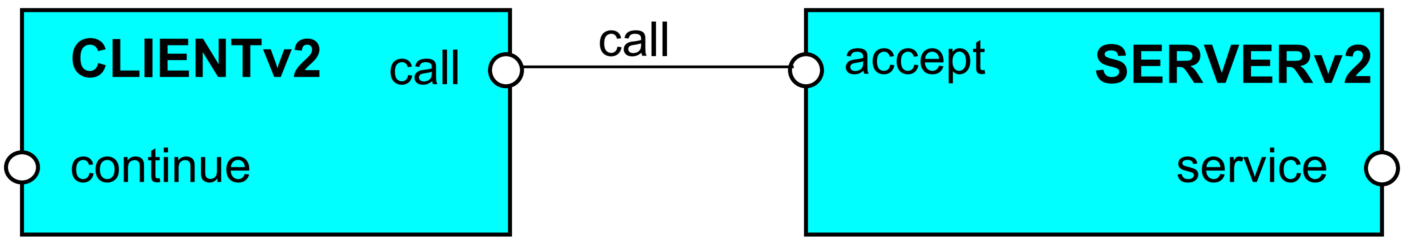

```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).

||TWOBUF = ?
```

structure diagrams

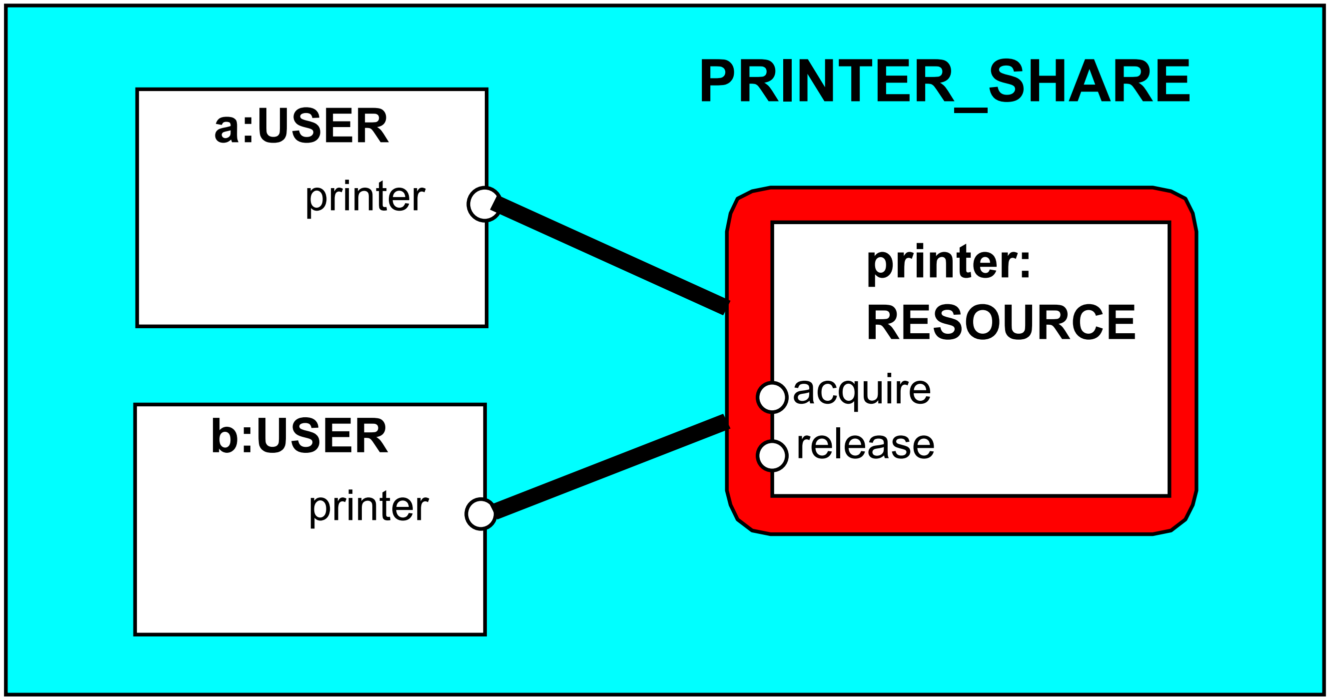
Structure diagram for `CLIENT_SERVER` ?



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).
```

Shared Objects & Mutual Exclusion

Concepts: process interference.
mutual exclusion.

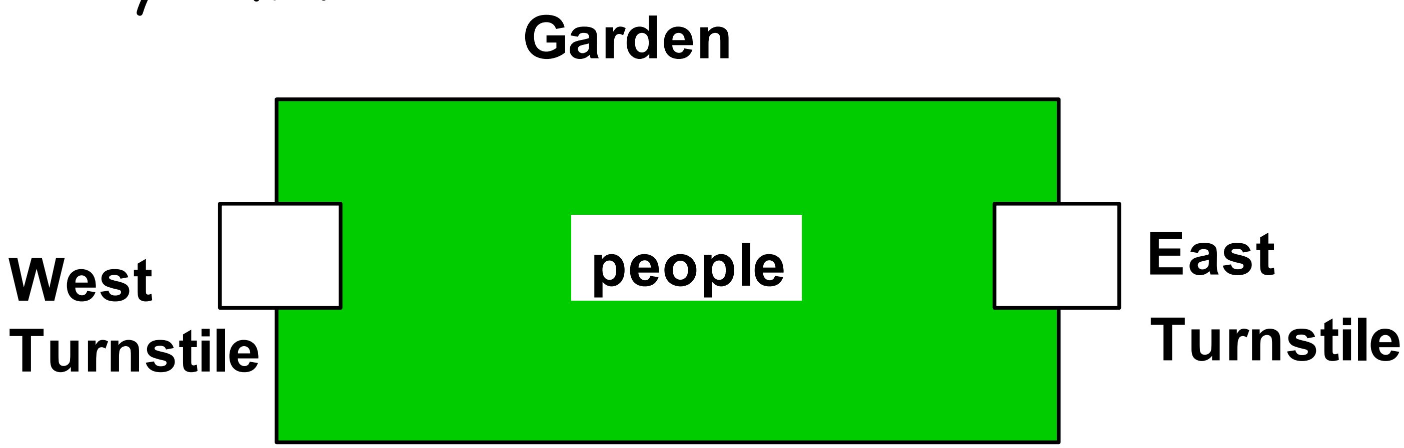
Models: model checking for interference
modeling mutual exclusion

Practice: thread interference in shared Java objects
mutual exclusion in Java
(synchronized objects/methods).

Interference

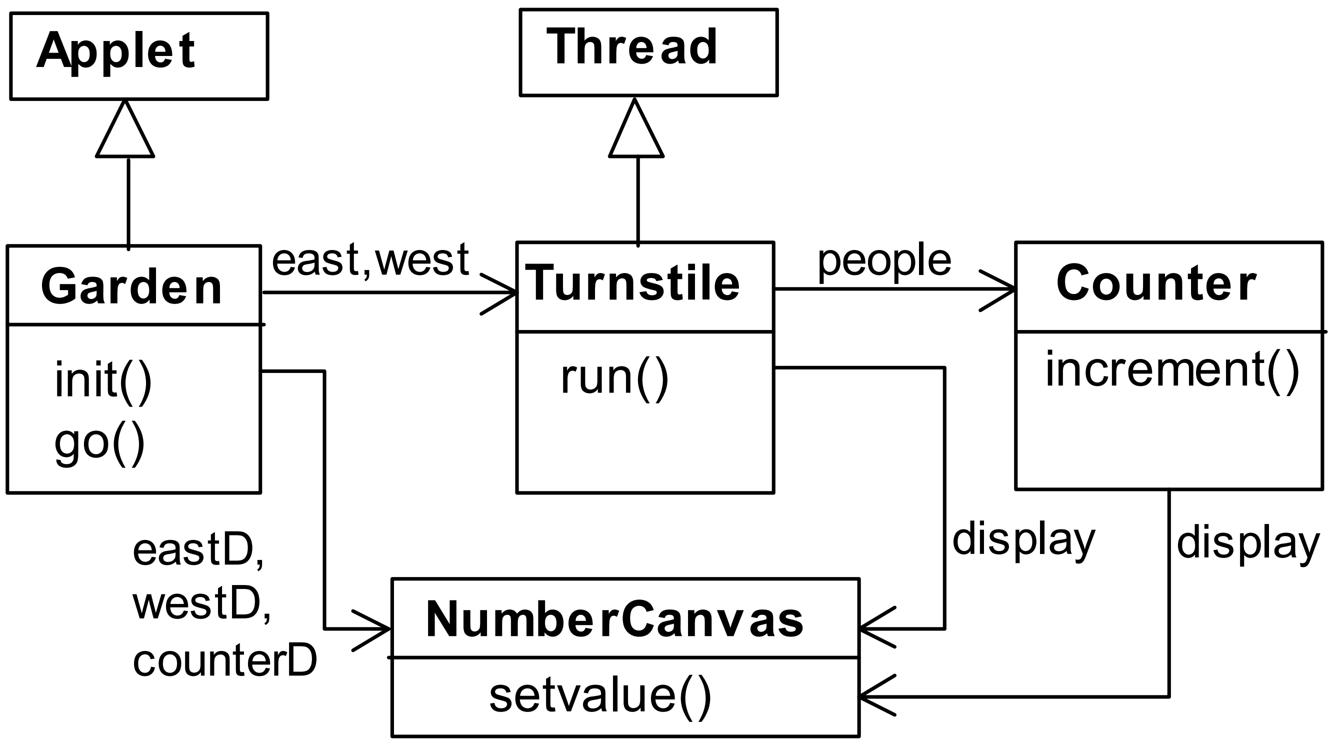
Ornamental garden problem:

People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.



The concurrent program consists of two concurrent threads and a shared counter object.

ornamental garden Program - class diagram



The **`Turnstile`** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **`increment()`** method of the counter object.

ornamental garden program

The **`Counter`** object and **`Turnstile`** threads are created by the **`go()`** method of the Garden applet:

```
private void go() {
  counter = new Counter(counterD);
  west = new Turnstile(westD,counter);
  east = new Turnstile(eastD,counter);
  west.start();
  east.start();
}
```

Note that **`counterD`**, **`westD`** and **`eastD`** are objects of **`NumberCanvas`** used in chapter 2.

Turnstile class

```
class Turnstile extends Thread {
  NumberCanvas display;
  Counter people;

  Turnstile(NumberCanvas n,Counter c)
    { display = n; people = c; }

  public void run() {
    try{
      display.setvalue(0);
      for (int i=1;i<=Garden.MAX;i++){
        Thread.sleep(500); //0.5 second between arrivals
        display.setvalue(i);
        people.increment();
      }
    } catch (InterruptedException e) {}
  }
}
```

The **`run()`** method exits and the thread terminates after **`Garden.MAX`** visitors have entered.

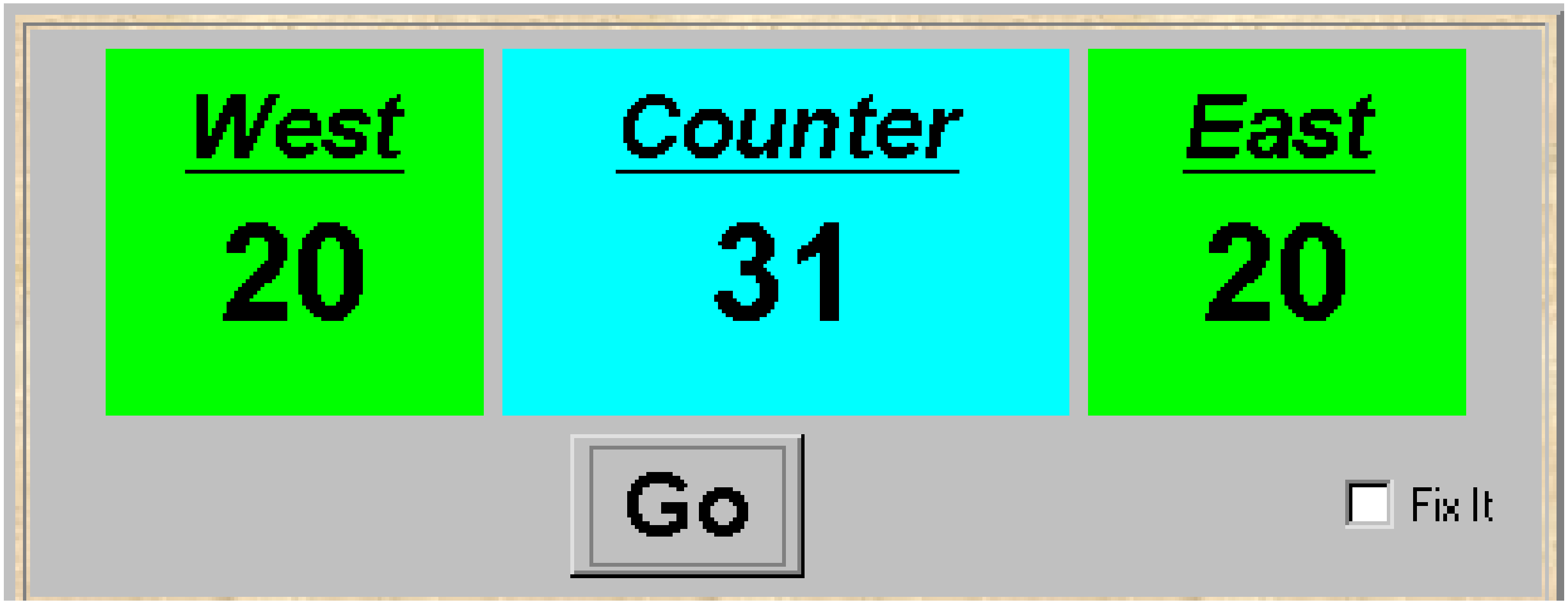
Counter class

```
class Counter {
  int value=0;
  NumberCanvas display;

  Counter(NumberCanvas n) {
    display=n;
    display.setvalue(value);
  }

  void increment() {
    int temp = value;    //read value
    Simulate.HWinterrupt();
    value=temp+1;        //write value
    display.setvalue(value);
  }
}
```

Hardware interrupts can occur at **arbitrary** times.

The **`counter`** simulates a hardware interrupt during an **`increment()`**, between reading and writing to the shared counter **`value`**. Interrupt randomly calls **`Thread.yield()`** to force a thread switch.

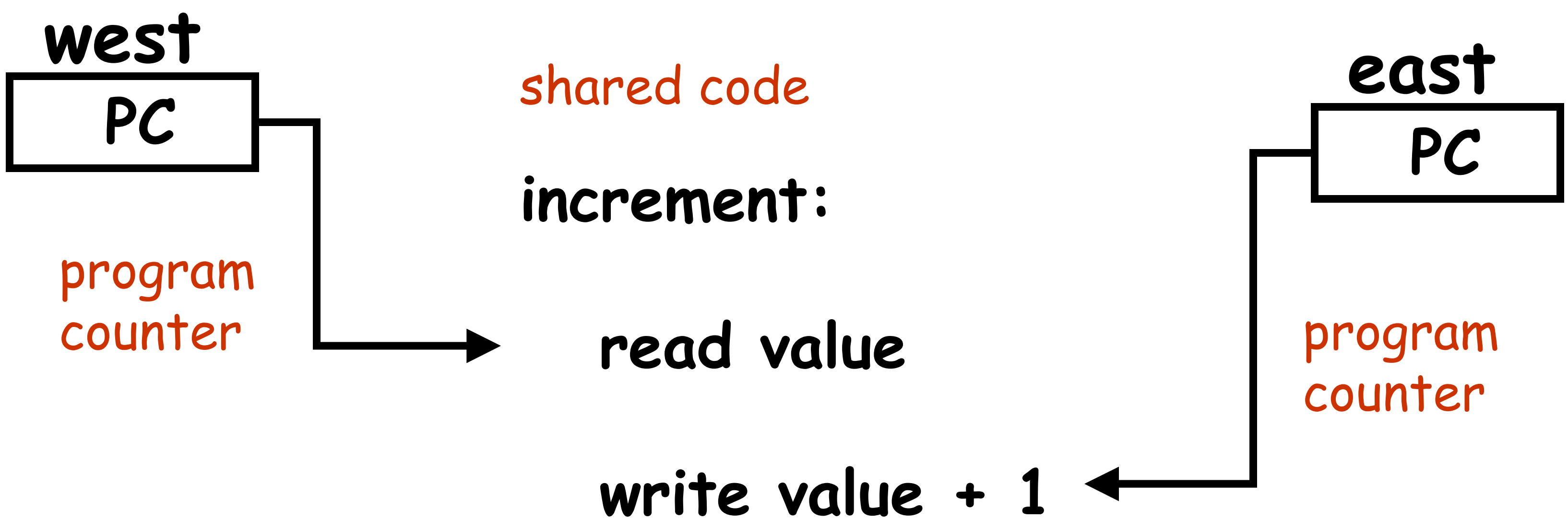
ornamental garden program - display



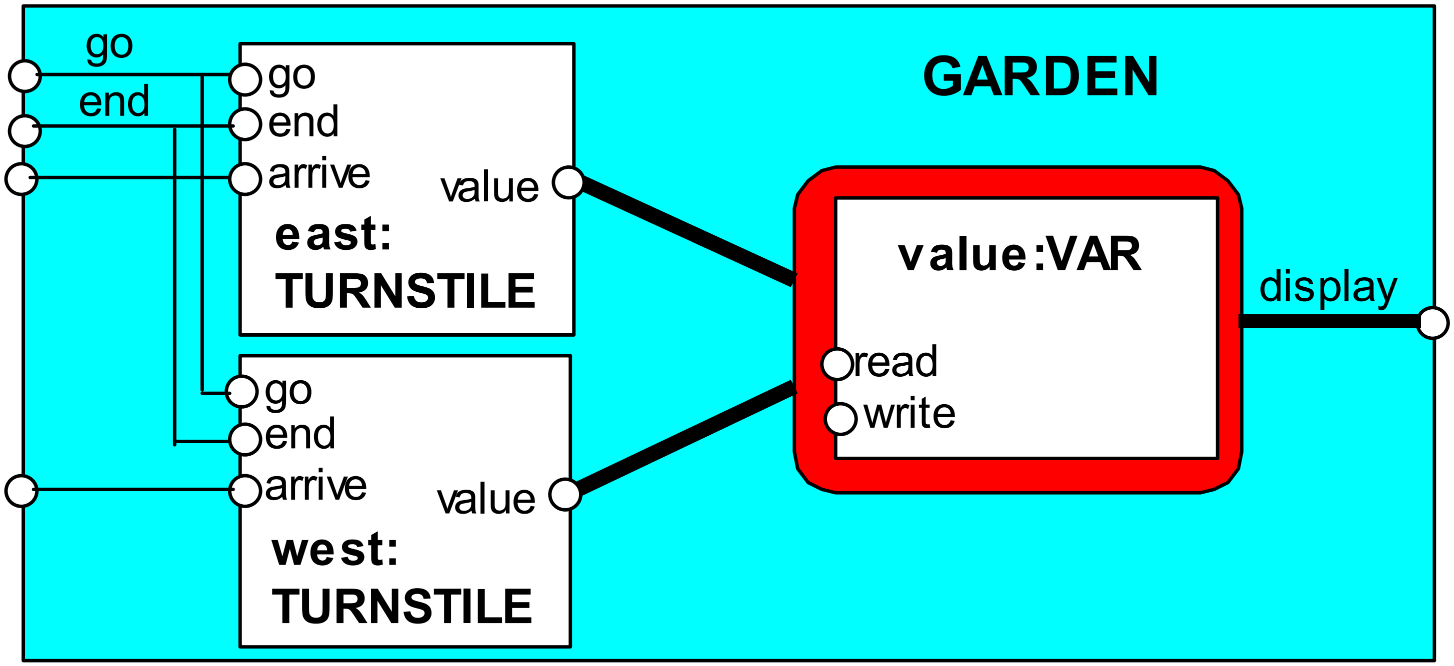
After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. ***Why?***

concurrent method activation

Java method activations are not atomic - thread objects **east** and **west** may be executing the code for the increment method at the same time.



ornamental garden Model



Process **VAR** models read and write access to the shared counter **value**.

Increment is modeled inside **TURNSTILE** since Java method activations are not atomic i.e. thread objects **east** and **west** may interleave their **read** and **write actions**.

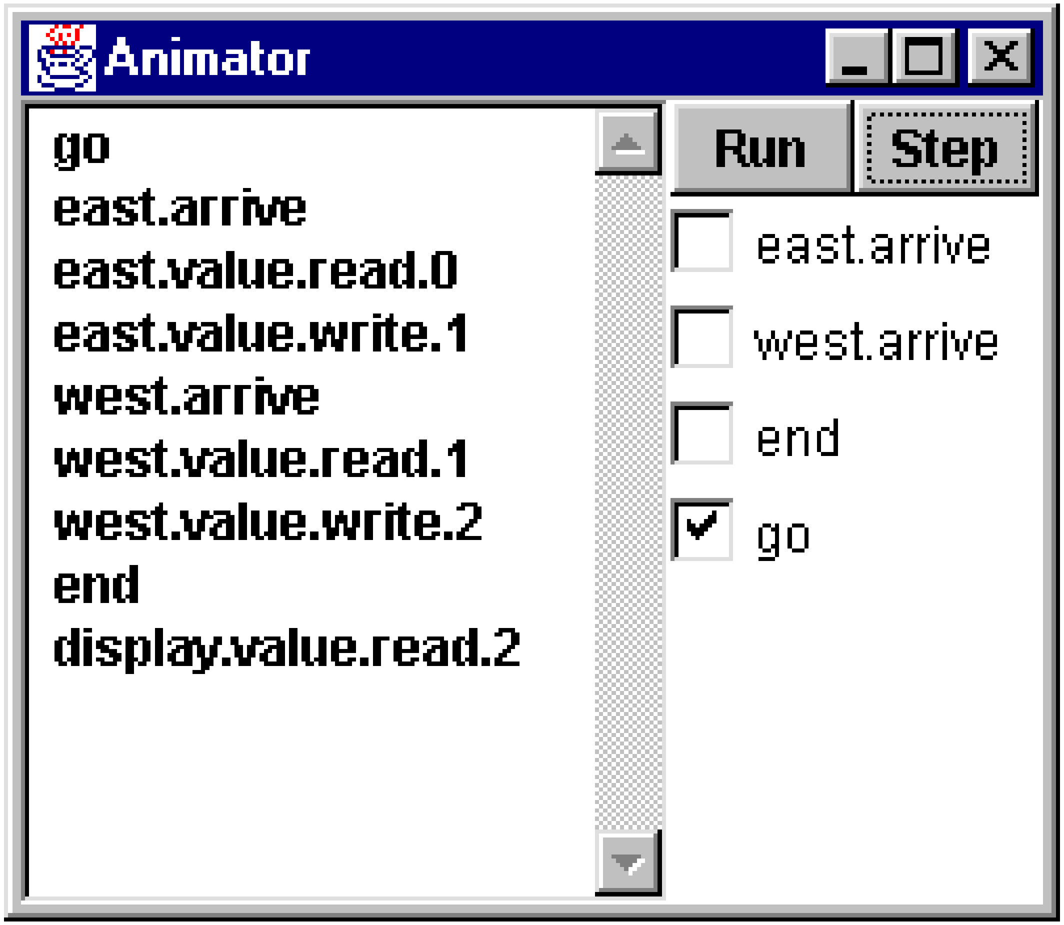
ornamental garden model

```
const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} }

VAR       = VAR[0],
VAR[u:T] = (read[u]    ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
             -> value.write[x+1]->RUN
            )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || { east,west,display} ::value:VAR)
            /{ go /{ east,west} .go,
              end/{ east,west} .end} .
```

The alphabet of process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The alphabet of **TURNSTILE** is extended with **VarAlpha** to ensure no unintended free actions in **VAR** ie. all actions in **VAR** must be controlled by a **TURNSTILE**.

checking for errors - animation



Scenario checking - use animation to produce a trace.

Is this trace correct?

checking for errors - exhaustive analysis

Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

```
TEST        = TEST[0],
TEST[v:T]   =
      (when (v<N){east.arrive,west.arrive}->TEST[v+1]
      |end->CHECK[v]
      ),
CHECK[v:T] =
     (display.value.read[u:T] ->
        (when (u==v) right -> TEST[v]
        |when (u!=v) wrong -> ERROR
        )
     )+{display.VarAlpha}.
```

Like **STOP**, **ERROR** is a predefined FSP local process (state), numbered -1 in the equivalent LTS.

ornamental garden model - checking for errors

```
||TESTGARDEN = (GARDEN || TEST).
```

Use *LTSA* to perform an exhaustive search for **ERROR**.

```
Trace to property violation in TEST:
    go
    east.arrive
    east.value.read.0
    west.arrive
    west.value.read.0
    east.value.write.1
    west.value.write.1
    end
    display.value.read.1
    wrong
```

LTSA produces the shortest path to reach **ERROR**.

Interference and Mutual Exclusion

> Destructive update, caused by the arbitrary interleaving of read and write actions, is termed ***interference***.

Interference bugs are extremely difficult to locate. The general solution is to give methods *mutually exclusive* access to shared objects. Mutual exclusion can be modeled as atomic actions.

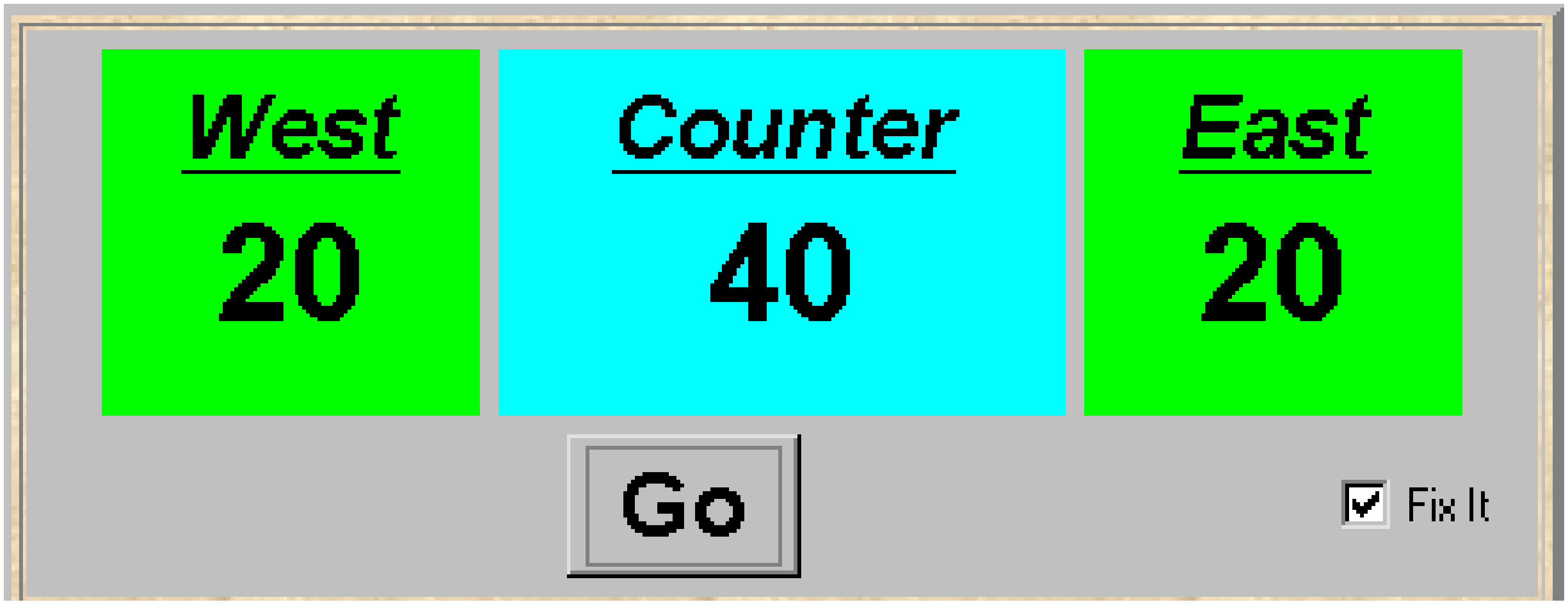
Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword **`synchronized`**.

We correct **`COUNTER`** class by deriving a class from it and making the increment method synchronized:

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n)
      {super(n);}

   synchronized void increment() {
         super.increment();
   }
}
```

mutual exclusion - the ornamental garden



Java associates a ***lock*** with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.

Java synchronized statement

Access to an object may also be made mutually exclusive by using the **`synchronized`** statement:

```
synchronized (object) { statements }
```

A less elegant way to correct the example would be to modify the **`Turnstile.run()`** method:

```
synchronized(counter) {counter.increment();}
```

Why is this "less elegant"?

To ensure mutually exclusive access to an object, **all object methods** should be synchronized.

Modeling mutual exclusion

To add locking to our model, define a **LOCK**, compose it with the shared **VAR** in the garden, and modify the alphabet set :

```
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

set VarAlpha = {value.{read[T],write[T],
                       acquire, release}}
```

Modify **TURNSTILE** to acquire and release the lock:

```
TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.acquire
             -> value.read[x:T]->value.write[x+1]
             -> value.release->RUN
            )+VarAlpha.
```

Revised ornamental garden model - checking for errors

A sample animation execution trace

```
go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right
```

Use TEST and ***LTSA*** to perform an exhaustive check.

Is TEST satisfied?

COUNTER: Abstraction using action hiding

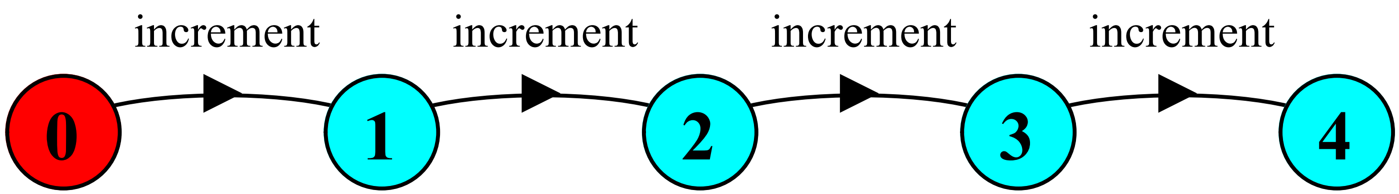
To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For `SynchronizedCounter` we hide **read**, **write**, **acquire**, **release** actions.

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u]->VAR[u]
           | write[v:T]->VAR[v]).

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
             -> (when (x<N) write[x+1]
                 ->release->increment->INCREMENT
                )
             )+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
```

COUNTER: Abstraction using action hiding

Minimized LTS:



We can give a more abstract, simpler description of a **COUNTER** which generates the same LTS:

```
COUNTER = COUNTER[0]
COUNTER[v:T] = (when (v<N) increment -> COUNTER[v+1]).
```

This therefore exhibits "equivalent" behavior i.e. has the same observable behavior.

Typical efficiency measurement (of synchronized calls)

```
 double start = new Date().getTime();
for(long i = ITERATIONS; --i >= 0 ;)
     tester.locking(0,0);
 double end = new Date().getTime();
 double locking_time = end - start;

 start = new Date().getTime();
for(long i = ITERATIONS; --i >= 0 ;)
     tester.not_locking(0,0);
 end = new Date().getTime();
 double not_locking_time = end - start;

 double time_in_synchronization =
        locking_time - not_locking_time;
```

Synchronized calls are 7-8% slower than normal calls. Alternative to synchronization: value assignment

Methods returning a **simple** value in a shared valiable, do not need synchronization due to the atomicity of assignment, e.g.:

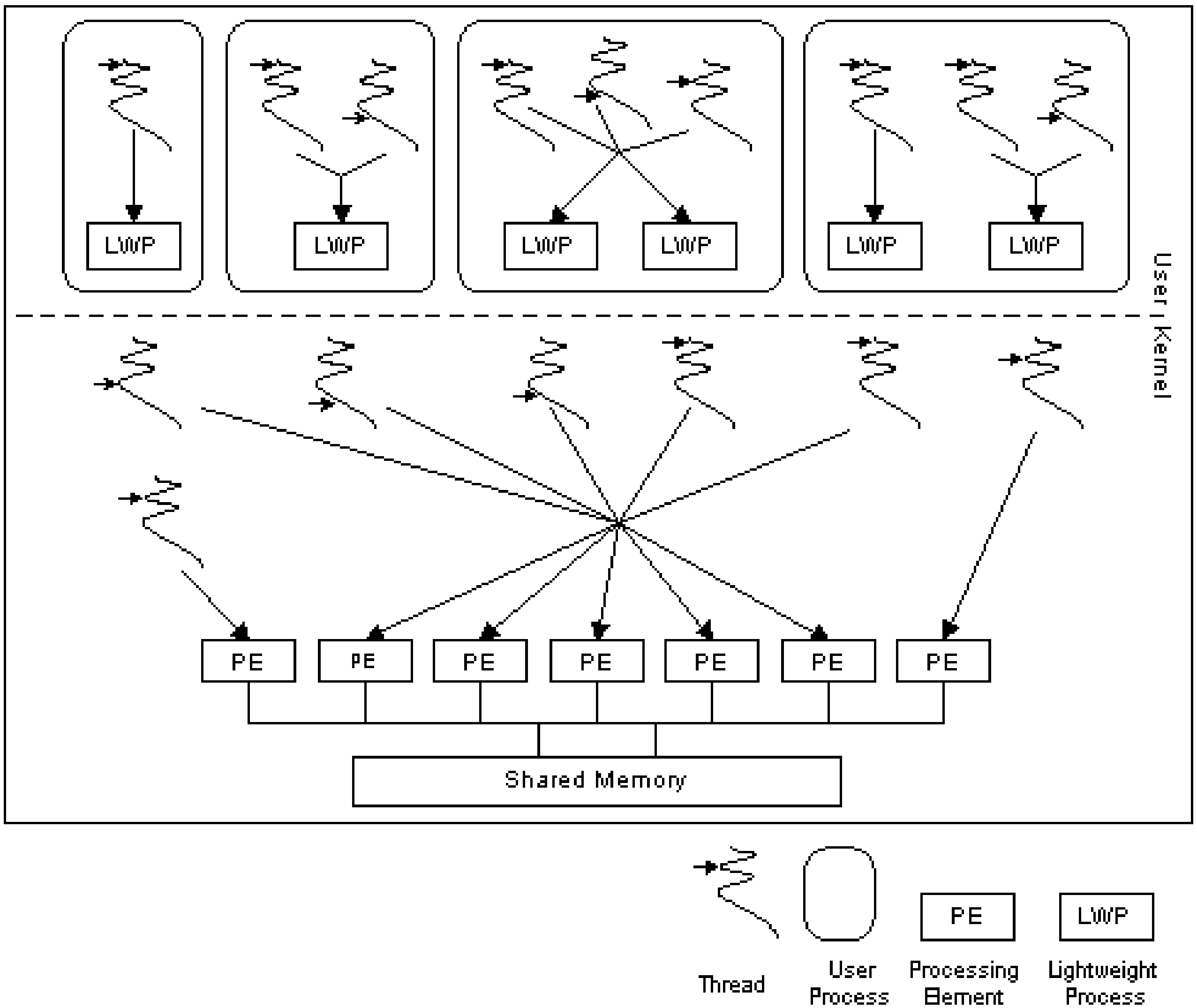
```
class some_class
{
    int some_field;
    void f( some_class arg ) // deliberately no
                                synchronized
    {
        // ...
        some_field = new_value;      // do this
                                        last.
    }
}
```

Alternative to synchronization: priorities?

According to JLS, threads may be assigned (10) different priorities. However, the effect is highly *platform dependant*!

Solaris provides both cooperative (no implicit preemption) and preemptive thread execution, but without the users control. *Windows* provides only preemptive execution.terference bugs are extremely difficult to locate.

Typical execution platform



Summary

- Concepts
  - process interference
  - mutual exclusion
- Models
  - model checking for interference
  - modeling mutual exclusion
- Practice
  - thread interference in shared Java objects
  - mutual exclusion in Java (synchronized objects/methods).
  - priorities do not offer predictable thread execution