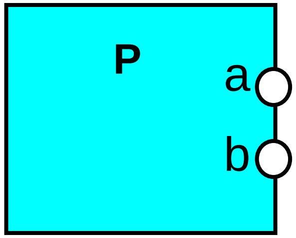
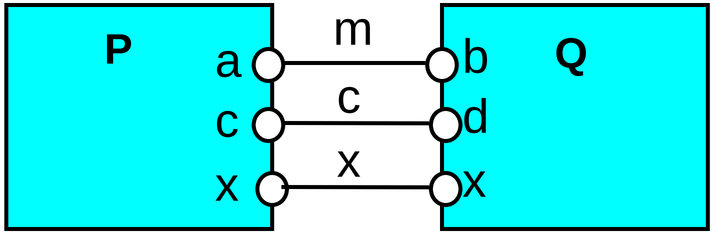
Lecture 5

- Administration

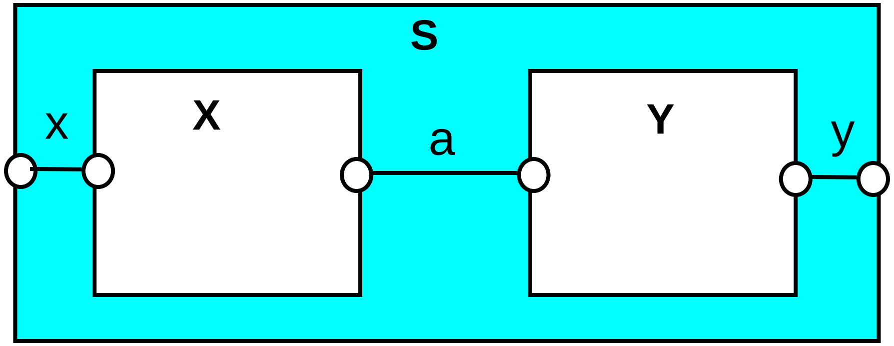
structure diagrams



Process P with alphabet {a,b}.



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

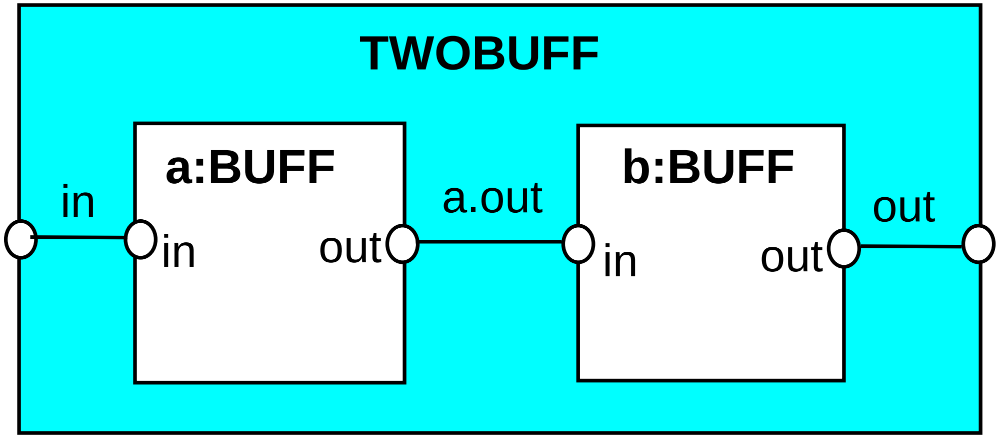


Composite process
||S = (X||Y) @ {x,y}

structure diagrams

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

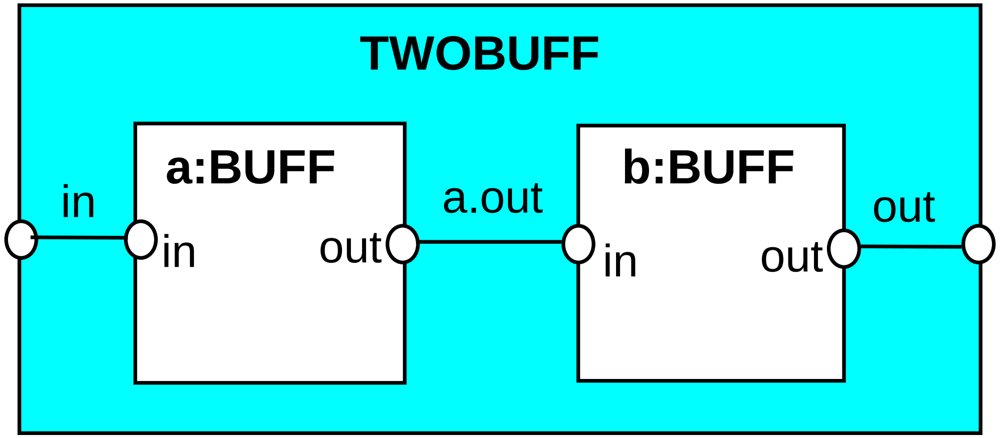
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).

||TWOBUF = ?
```



structure diagrams

```
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*.

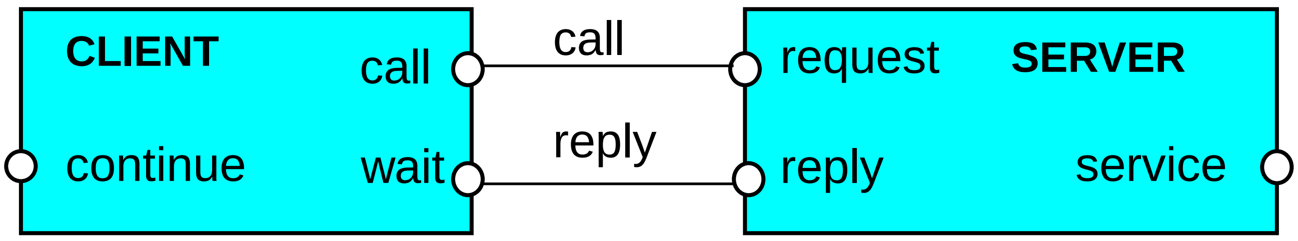



```
||TWOBUF =     (a:BUFF || b:BUFF)
                  /{in/a.in, a.out/b.in, out/b.out}
                     @{in,out}.
```

structure diagrams

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

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

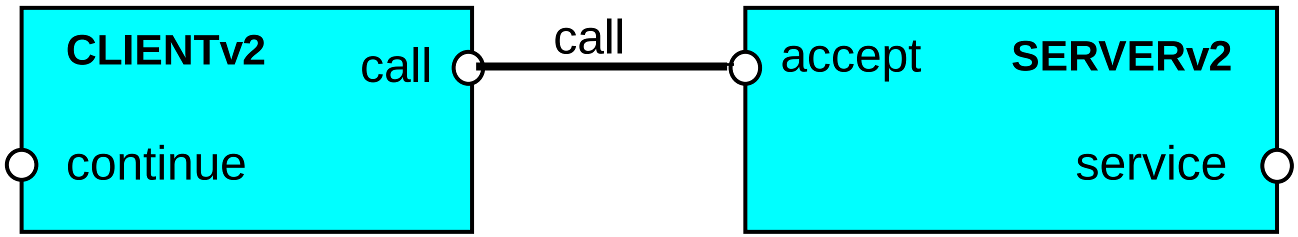
Structure diagram for **CLIENT_SERVER** ?



structure diagrams

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

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

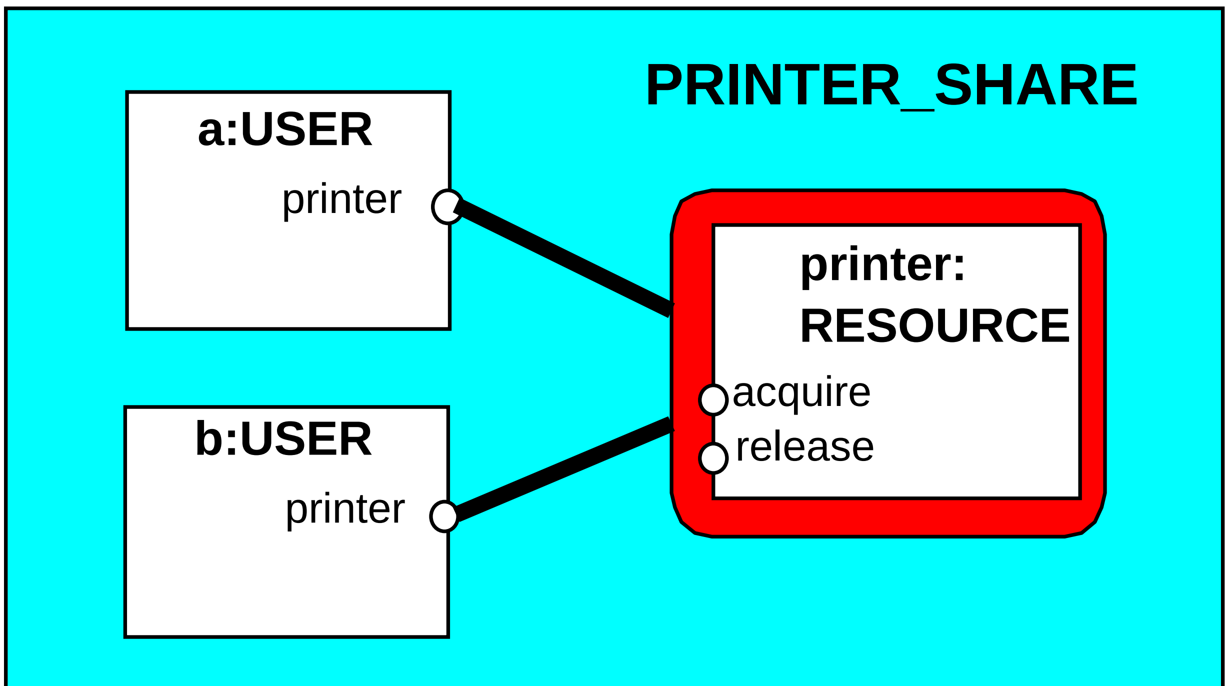
Structure diagram for **CLIENT_SERVERv2** ?



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).
```



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).
```

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

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

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

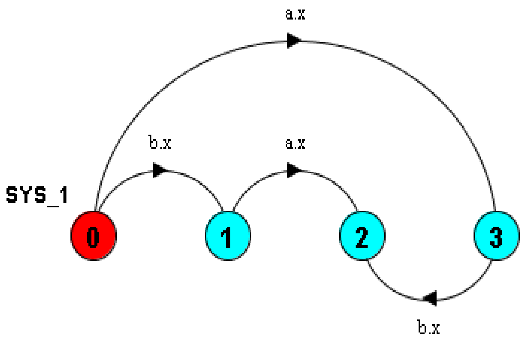
```
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

Example

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

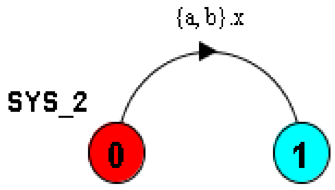
```
||SYS_1 = {a,b}:X.
```

LTS?   Traces?   Number of states?




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

LTS?   Traces?   Number of states?



action relabelling

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

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

Relabeling to ensure that composed processes synchronize on particular actions:

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

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

action relabelling

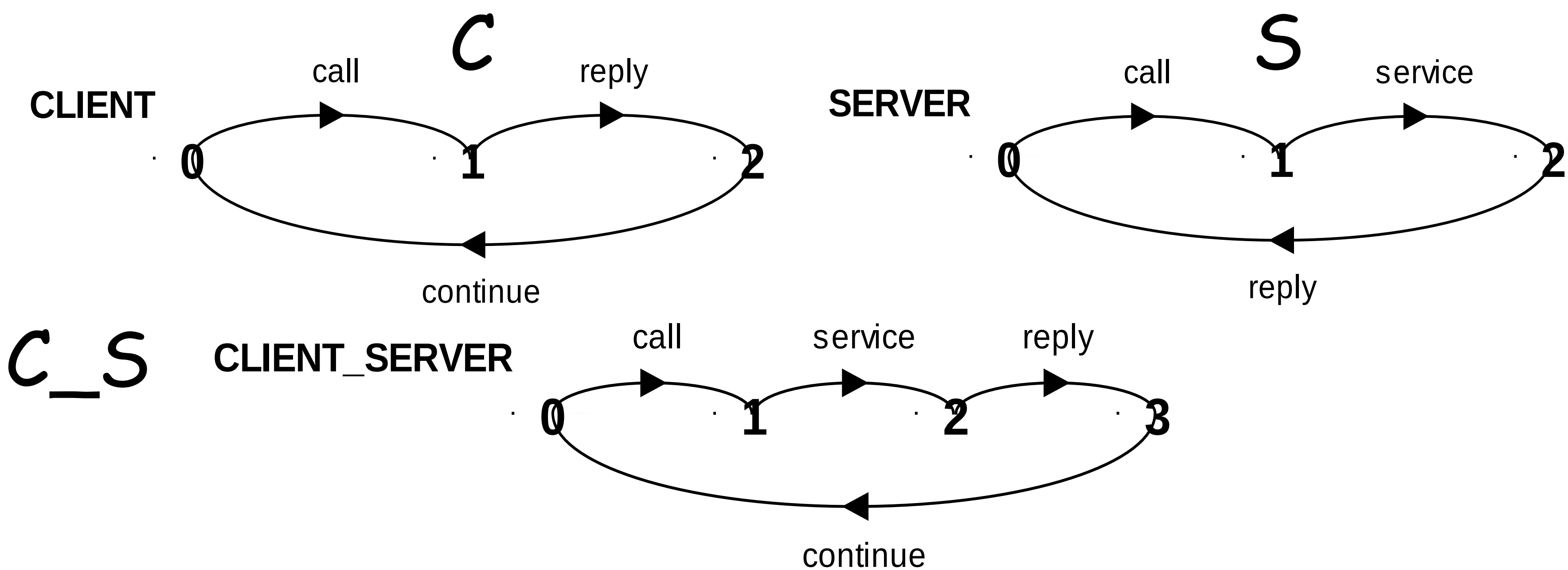
```
CLIENT = (call->wait->continue->CLIENT).
```

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

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

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

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



action relabelling - prefix labels

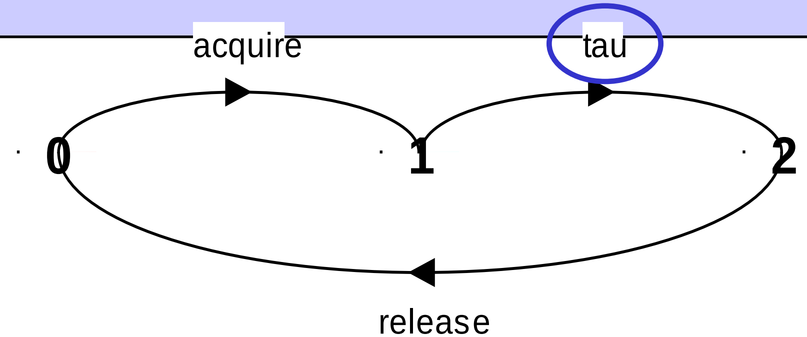
An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
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 **\{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.

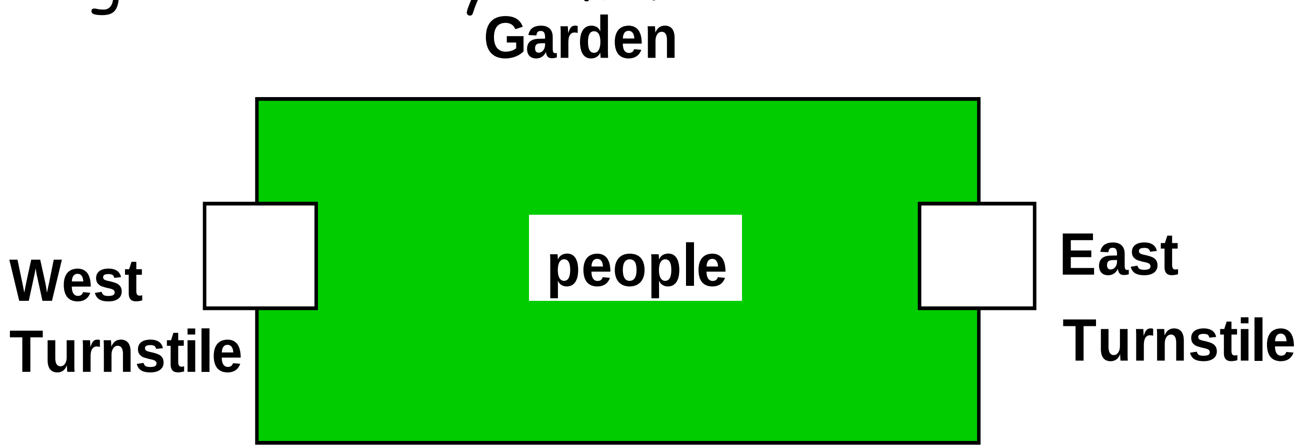
```
USER = (acquire->use->release->USER)

\{use}.
```



4.1 Interference

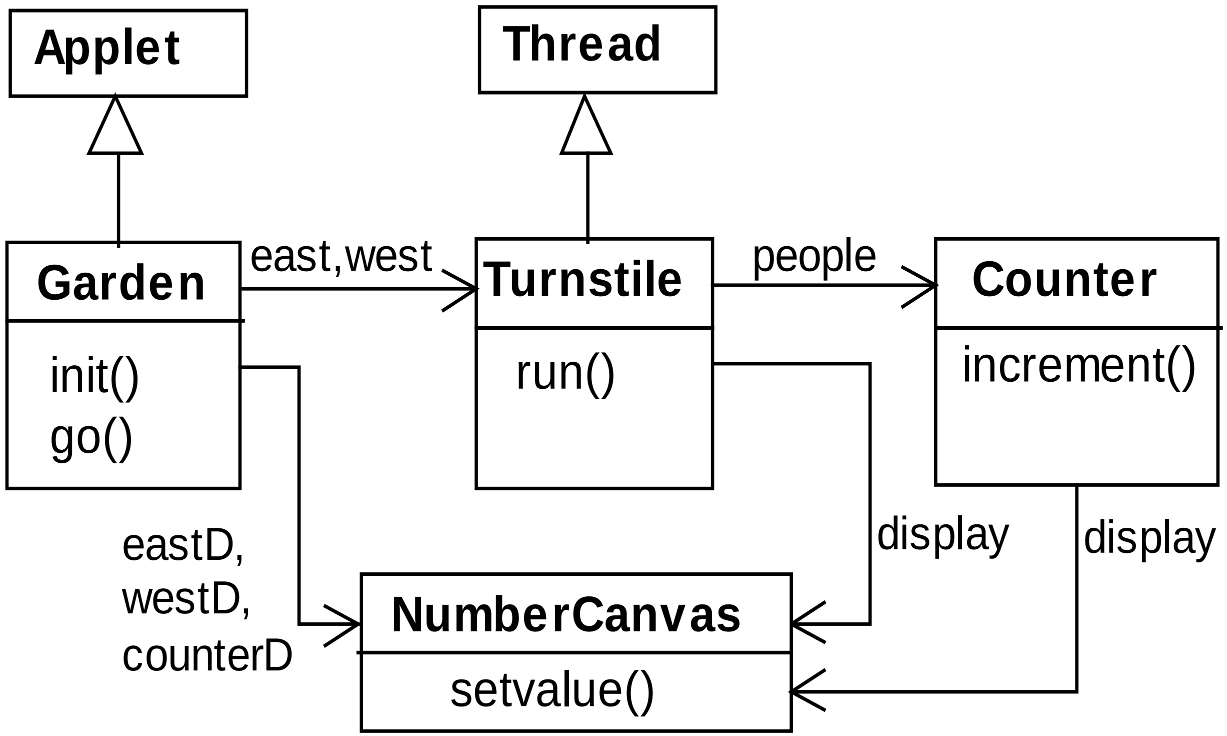
Ornamental garden problem:

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

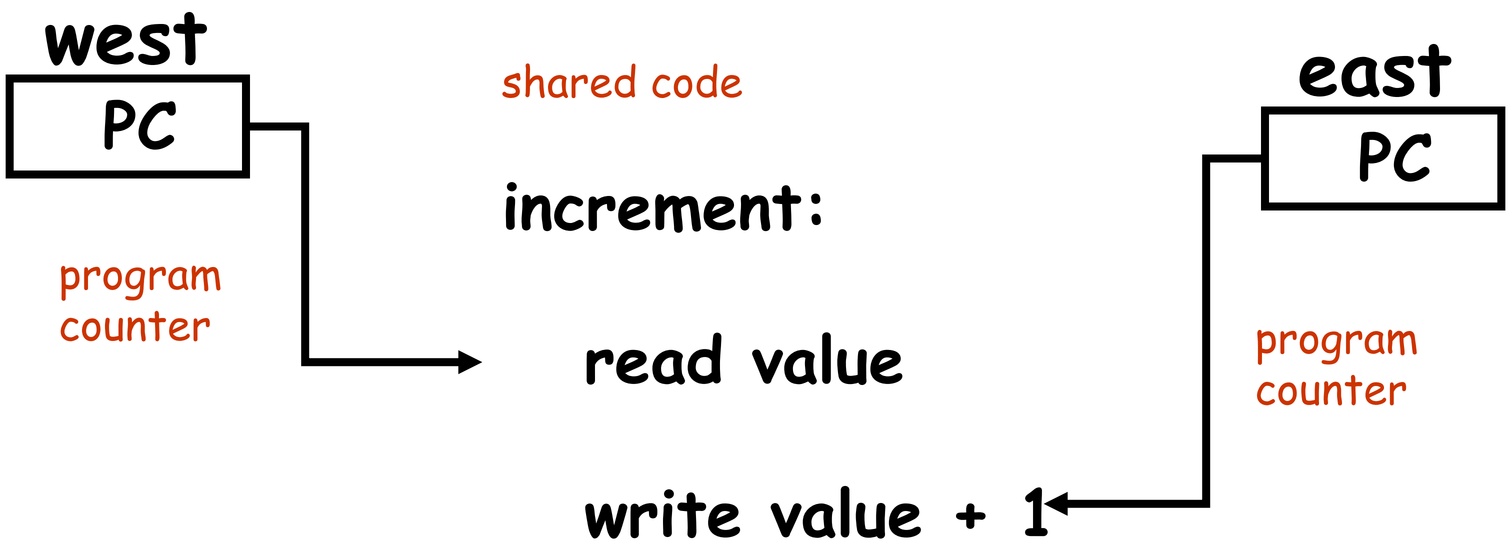
The **Counter** object and **Turnstile** threads are created by the **go()** method of the Garden applet:

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

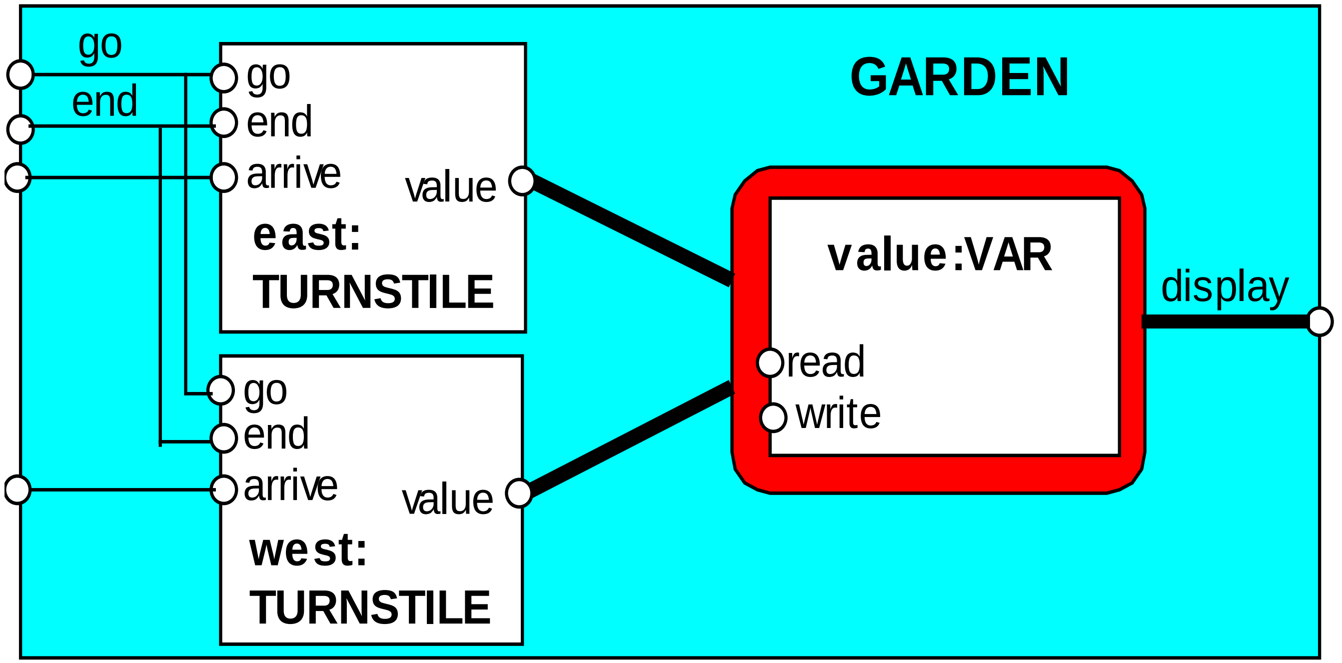
Note that **counterD**, **westD** and **eastD** are objects of **NumberCanvas** used in chapter 2.

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 shared process **VAR** is declared explicitly as a **set** constant, **VarAlpha**.

The **TURNSTILE** alphabet is extended with **VarAlpha** to ensure no unintended **free (autonomous) actions** in **VAR** eg. **value.write[0]**. All actions in the shared **VAR** must be controlled (shared) by a **TURNSTILE**.

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.