Concurrent Execution III & Interference I

Lecture 6

Alma Rahat

CS-210: Concurrency

10 February, 2021

What did we do in the last session?

- We learned how to create models of parallel processes.
- We learned how to model shared actions and its implications.

In this lecture...

Learning outcomes.

1. To model shared resources between processes.
2. To apply relabelling in FSP for action synchronisation between processes.
3. To identify unimportant actions and hide them using FSP.
4. To compose concurrent processes from a given scenario and code.

Outline.

1. Modelling Concurrency.
   - Resources (mutual exclusion).
2. Action synchronisation.
3. Action hiding
4. Java code for a hypothetical Garden.

Recap: Sharing Between Processes

Swansea University
Prifysgol Abertawe

Simple rules to understand the nature of sharing:

Action Common behaviour (what the process does; method) across processes, e.g. two users can play chess with their computers.

Remember You cannot perform a common action until all pre-requisite actions by all sharer have been performed.

Resource Common property (has-a, uses-a or controls-a type relationship; process or object) across processes, e.g. multiple users at a household have a toaster.

Shared Resources: Mutual Exclusion

Swansea University Prifysgol Abertawe

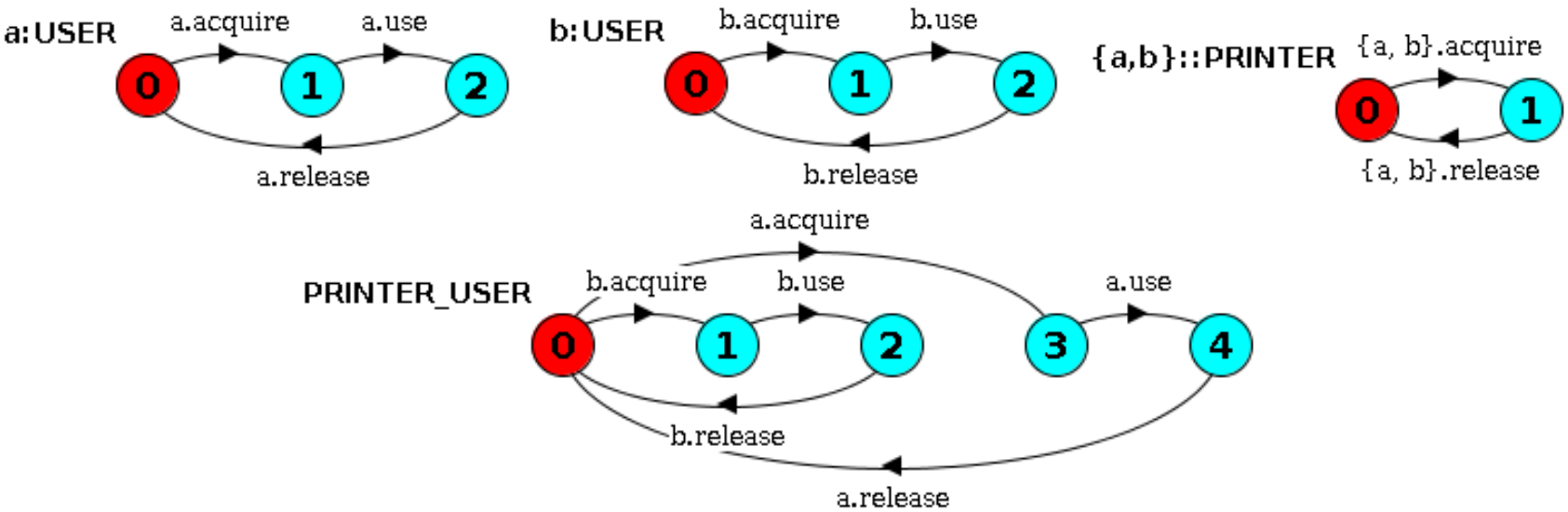
Imagine, two users are sharing a printer: both cannot print at the same time, but have to wait their turn.

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

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

LTSs for Shared Printer

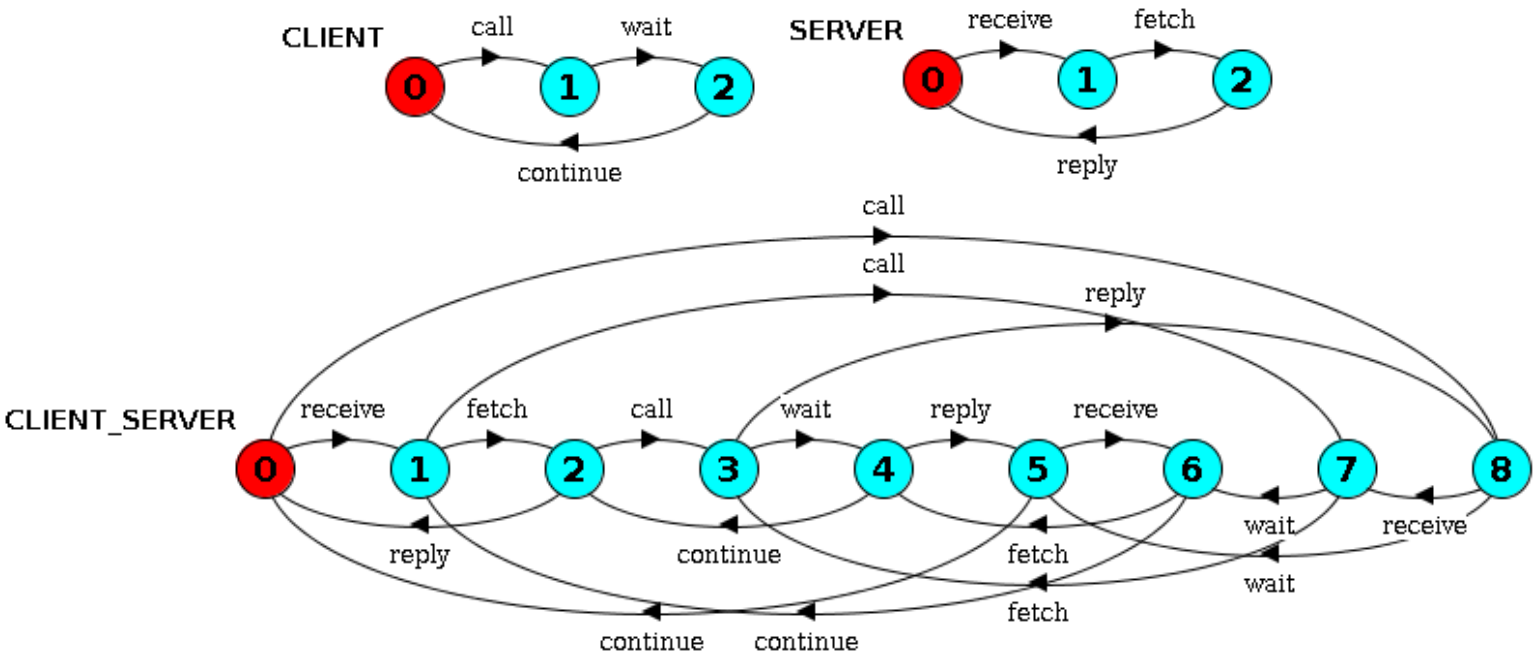
```
USER = (acquire -> use -> release -> USER).
PRINTER = (acquire -> release -> PRINTER).
||PRINTER_USER = (a:USER || b:USER || {a,b}::PRINTER).
```



Any questions?

Action Synchronisation

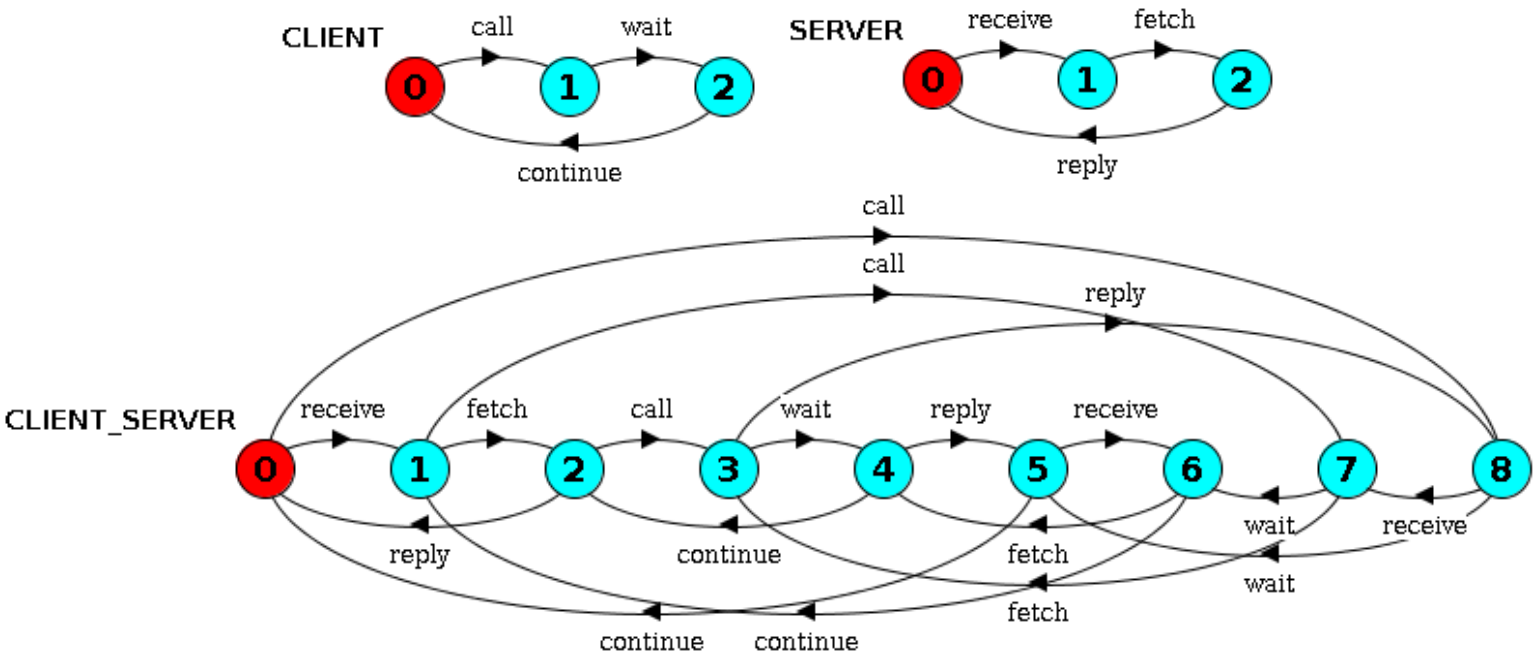
Imagine, we have two processes, `CLIENT` and `SERVER`. When the `CLIENT` wants data from the `SERVER`, it `calls` for data, `waits` for `reply` from the `SERVER`, and upon receiving data `continues` to be a `CLIENT`. The `SERVER` upon `receive`-ing a call, it `fetches` data and sends data in `reply`.



Any issues?

Action Synchronisation

Imagine, we have two processes, `CLIENT` and `SERVER`. When the `CLIENT` wants data from the `SERVER`, it `calls` for data, `waits` for `reply` from the `SERVER`, and upon receiving data `continues` to be a `CLIENT`. The `SERVER` upon `receive`-ing a call, it `fetches` data and sends data in `reply`.



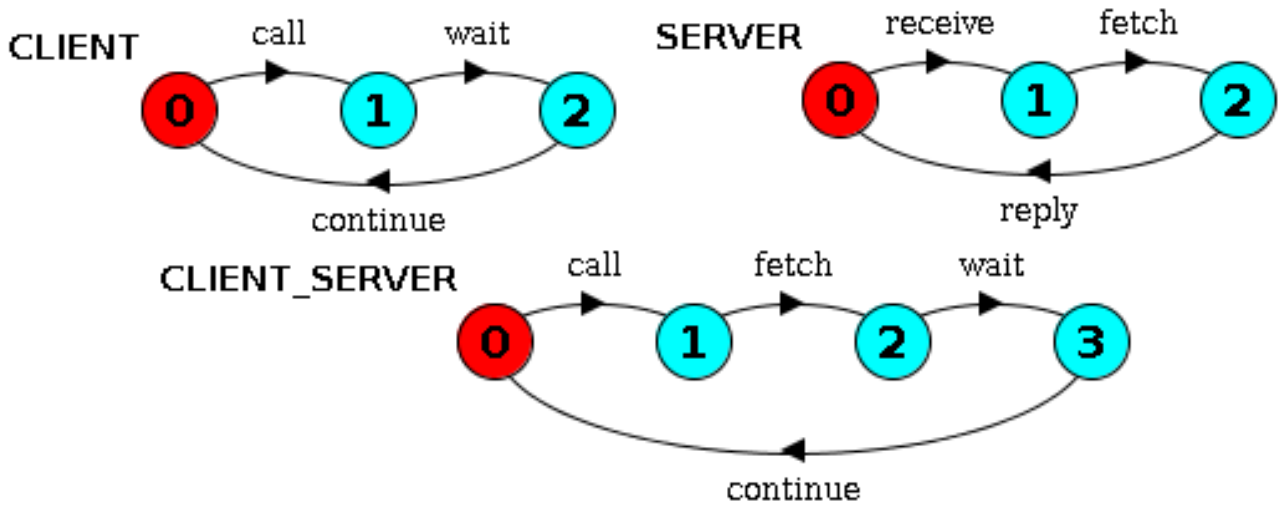
Any issues? `receive` between 0 and 1 happened without any `call`. So, we need to synchronise `call`/`request` and `wait`/`reply` actions. How?

Action Relabelling

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

`/{newLabel1/oldLabel1, ..., newLabelN/oldLabelN }.`

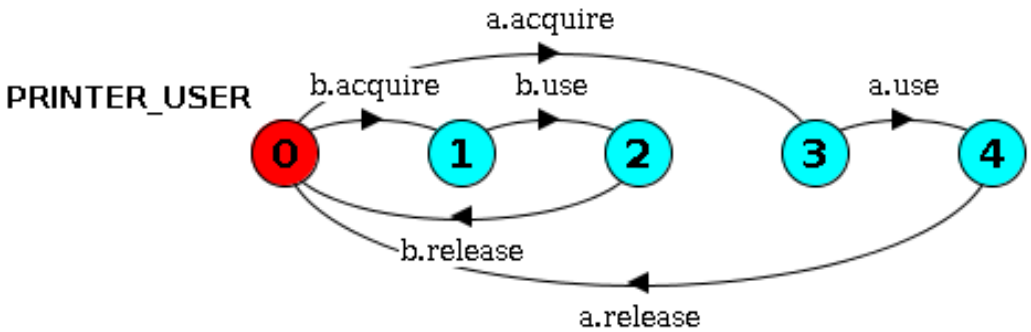
```
||CLIENT_SERVER = (CLIENT || SERVER) /{call/receive,
wait/reply}.
```



Revisiting Shared Printer

Previously, we saw an example of two users sharing a printer. Essentially, all the actions of the PRINTER were in a subset of the alphabets of the USER. What if that is not the case, and there are actions in the ?

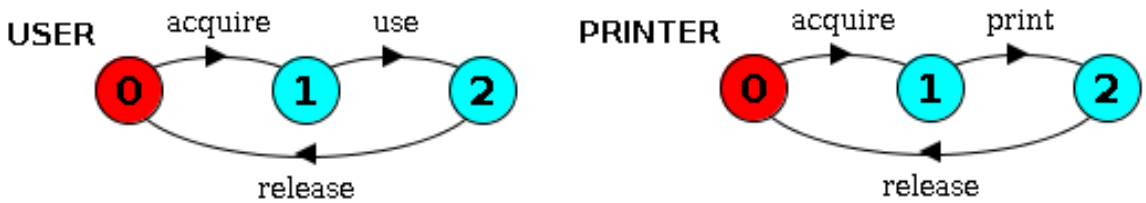
```
USER = (acquire -> use -> release -> USER).
PRINTER = (acquire -> release -> PRINTER).
||PRINTER_USER = (a:USER || b:USER || {a,b}::PRINTER).
```



Revisiting Shared Printer

Consider the following scenario, `PRINTER` has an additional action called `print`, which may be considered as equivalent to `use` in `USER`.

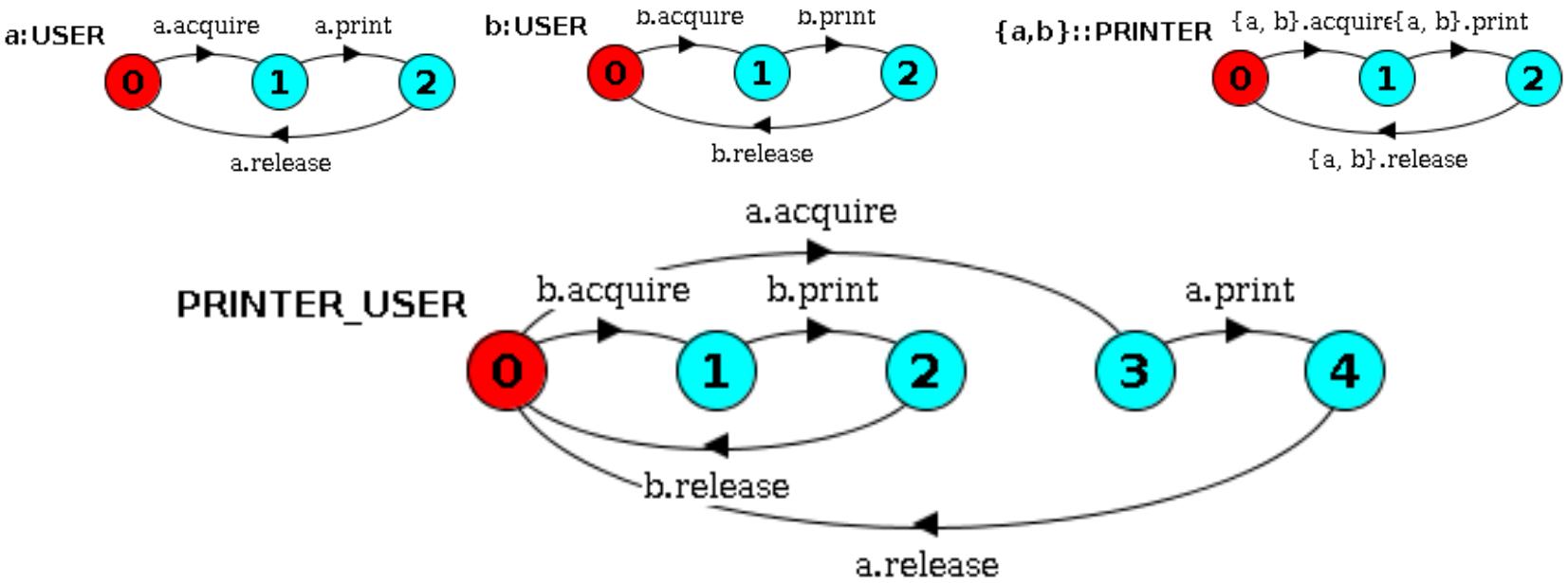
```
USER = (acquire -> use -> release -> USER).
PRINTER = (acquire -> print -> release -> PRINTER).
```



Revisiting Shared Printer

We can use action relabelling to synchronise the actions `use` and `print`, but we would need to do that for each individual sharing process `a` and `b`.

```
USER = (acquire -> use -> release -> USER).
PRINTER = (acquire -> print -> release -> PRINTER).
||PRINTER_USER = (a:USER || b:USER ||
{a,b}::PRINTER)/{b.print/b.use,a.print/a.use}.
```



Action Hiding

Sometimes we have actions that are local and unimportant from analysis perspective, e.g. the `print` action inside the `PRINTER`. How can we hide it?

Method 1

When applied to a process `P`, the hiding operator `\{a1, ..., ax}` removes the actions a1, ..., ax from the alphabet of `P` and makes these concealed actions "silent": represented as tau. Example:

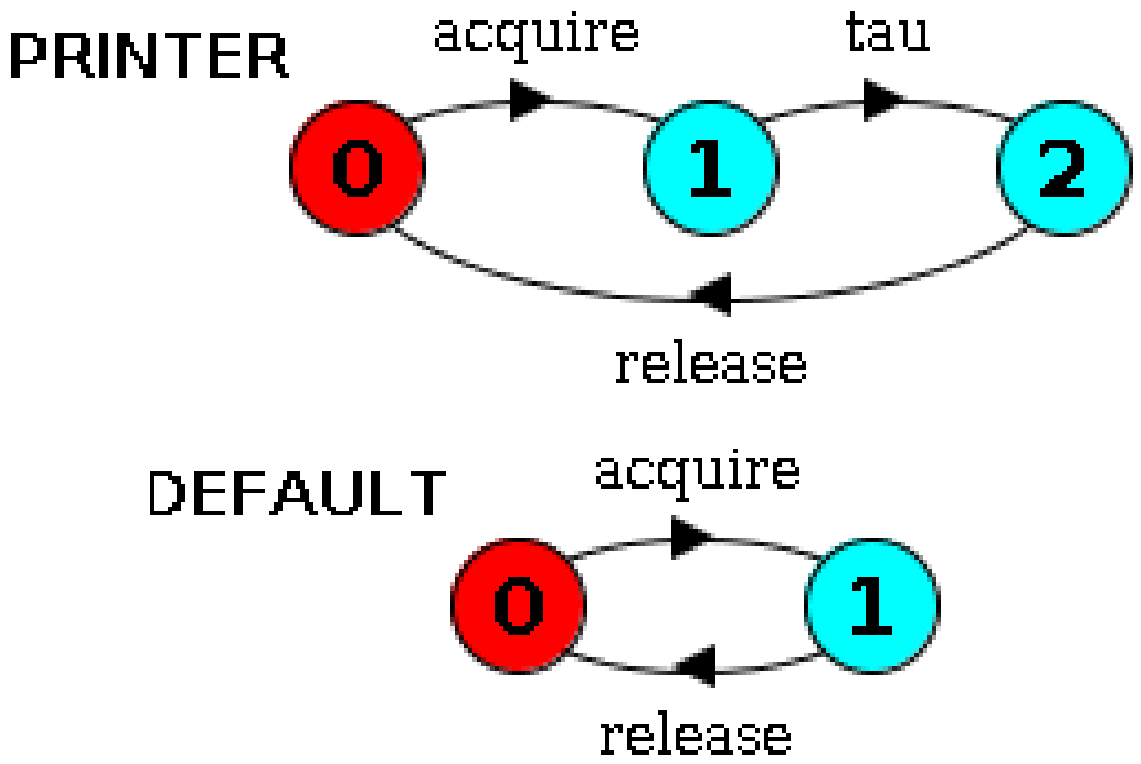
```
PRINTER = (acquire -> print -> release -> PRINTER)\{print}.
```

Method 2

When applied to a process `P`, the interface operator `@{a1, ..., ax}` removes the actions a1, ..., ax from the alphabet of `P`. Example:

```
PRINTER = (acquire -> print -> release -> PRINTER)@{print}.
```

LTSs for action hiding.



Minimising removes tau actions.

Any questions?

Homework

A roller coaster control system permits its car to depart when it is full.

Passengers arriving at the platform are registered by a turnstile. The controller signals the car to depart when there are enough passengers on the platform to fill the car to its maximum capacity of $M = 3$ passengers. The car then goes around the roller coaster track and waits for another M passengers. A maximum of M passengers are allowed on the platform.

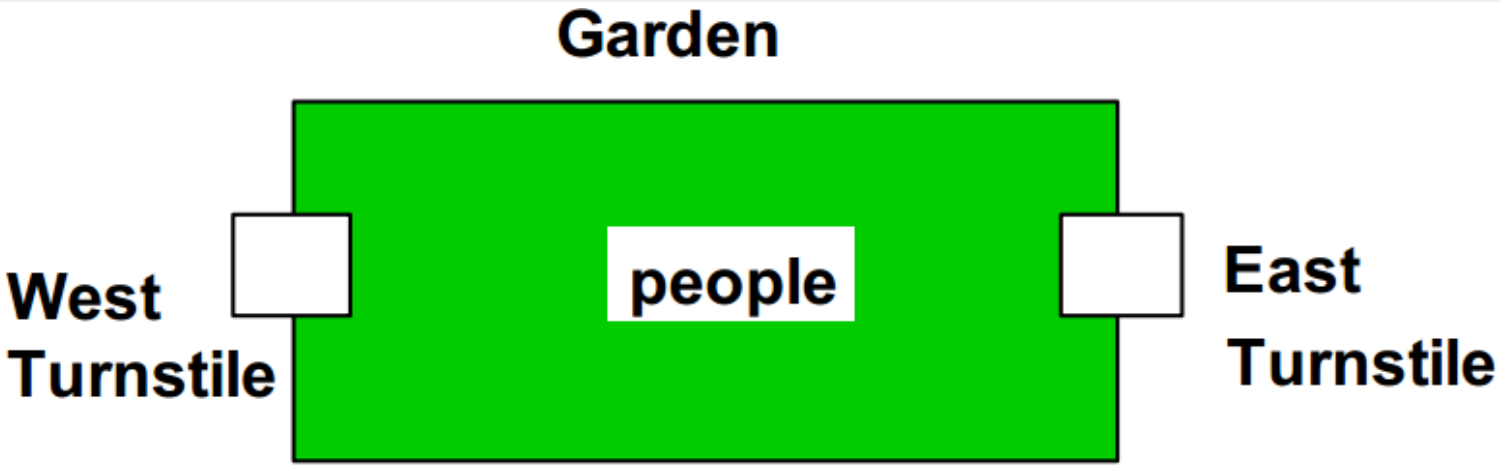
When the car has departed, no passenger can enter or leave the platform.

Write down the FSP for the `ROLLER_COASTER` as a composite of multiple processes.

There are three component processes: `TURNSTILE`, `CONTROLLER` and `CAR`.

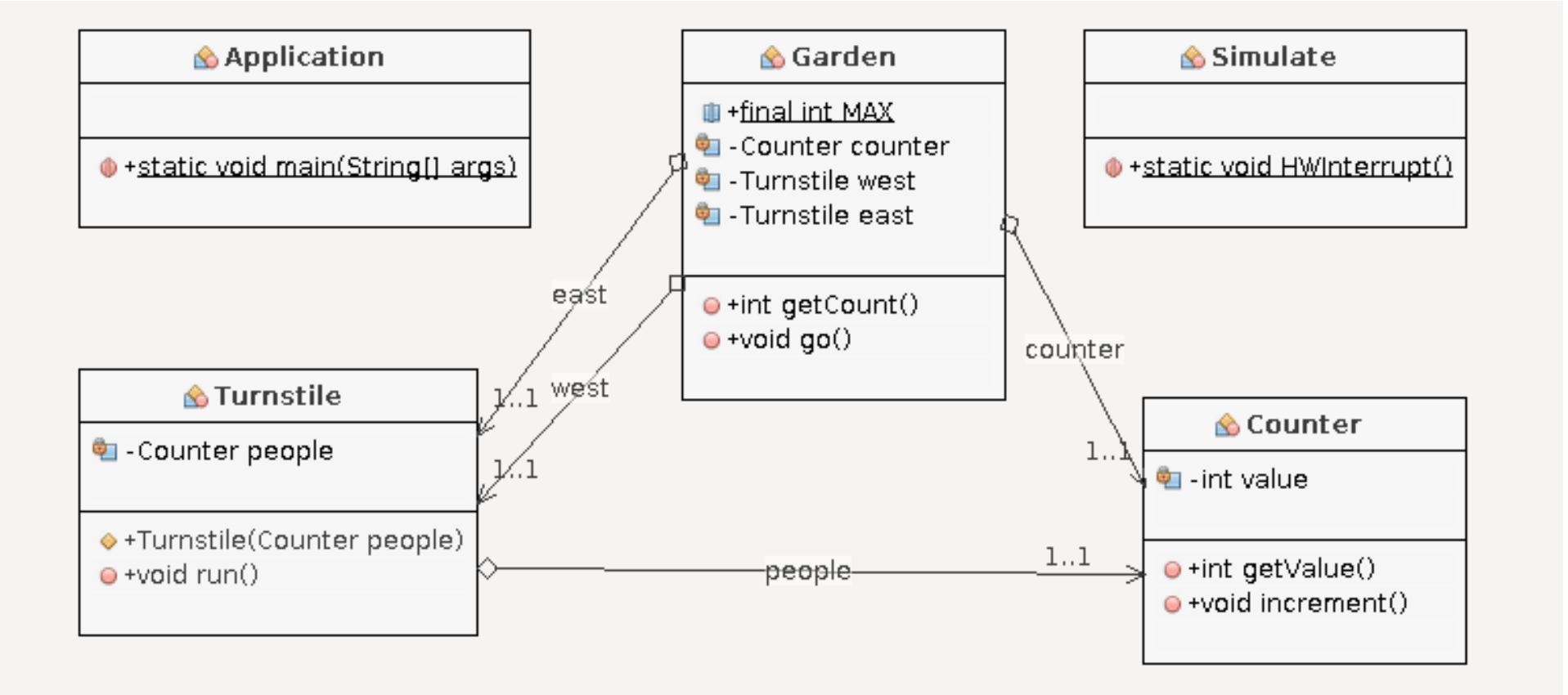
An Ornamental Garden Problem

Consider a garden that has two turnstiles that allow people to enter the garden (no need to let them leave!). The turnstiles run parallelly and share a single counter, and each increment this counter 100 times. Ideal way to generate a race condition.



How would you design an object oriented program for this?

Coding for a Garden



`Application` is the main controller program and the `Simulate` class mimics a hardware interrupt. Full code is in github repository.

Coding for a Garden

```
public class Counter {
    private int value;

    public int getValue(){
        return value;
    }

    public void increment(){
        int temp = value; //read value
        Simulate.HWInterrupt();
        value = temp + 1; // set value
    }
}
public class Simulate {
    public static void HWInterrupt(){
        if (Math.random()>0.5)
            Thread.yield();
    }
}
```

`Thread.yield()` allows the thread to releases control and lets other threads to run.

`increment` in `Counter` class reads `value`, but before incrementing lets the control go so that some other thread can run with 50% probability. It will come back and set the value.

Coding for a Garden

```
@Override
public void run() {
    double randomFactor;
    int waitingTime;
    for (int i=0; i< Garden.MAX; i++){
        randomFactor = Math.random();
        waitingTime = (int) Math.ceil(randomFactor * 10);
        try {
            Thread.sleep(waitingTime);
        } catch (InterruptedException ex) {

        }
        people.increment();
    }
}
```

`Turnstile` class implements the `Runnable` interface and provides an implementation of `run` method. After waiting for arbitrary amount of time, it increments its own `Counter` called `people`.

Coding for a Garden

```
public void go() throws InterruptedException{
    counter = new Counter();
    // create Turnstiles
    west = new Turnstile(counter);
    east = new Turnstile(counter);
    // create threads
    Thread westThread = new Thread(west, "west");
    Thread eastThread = new Thread(east, "east");
    // start threads
    westThread.start();
    eastThread.start();
    // wait for threads to die
    westThread.join();
    eastThread.join();
}
```

`Garden` class has a `go` method that creates a new instance of its `Counter`, threads for its `west` and `east` turnstiles, and starts the threads. Finally, it exits when both threads have finished what they were doing.

Any questions?

Summary

- We can share resources between processes.
- Actions may be synchronised – all prerequisite actions must happen before a synchronised action can happen.
- Unimportant actions can b hidden for simplicity.