Interference II

Lecture 7

Alma Rahat

CS-210: Concurrency

16 February, 2021

Swansea University
Prifysgol Abertawe

What did we do in the last session?

- Modelling concurrency: shared resources.
- Action synchronisation.
- Action hiding.
- Started to look at Java code for an ornamental garden.

In this lecture...

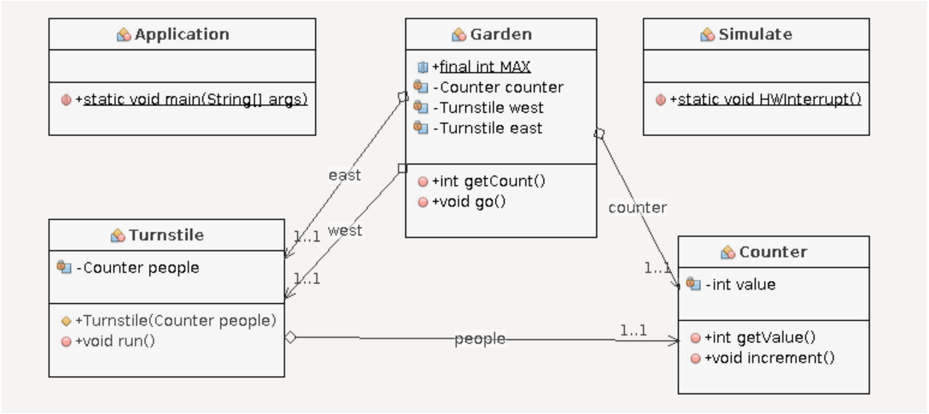
Learning outcomes.

1. To compose concurrent processes from a given scenario and code.
2. To evaluate the code and create models in FSP.
3. To test the model and ensure that the model does what was intended.

Outline.

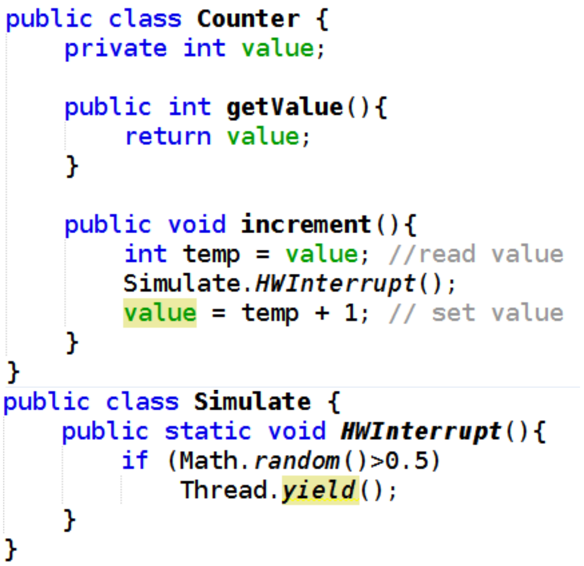
1. Java code for a concurrent scenario.
2. Create a FSP model.
3. Testing the model for correct behaviour.

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

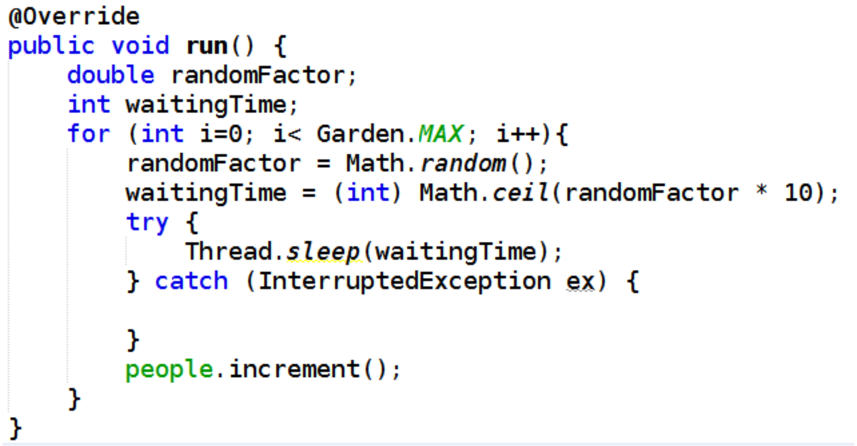


```
public class Counter {
    private int value;

    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.

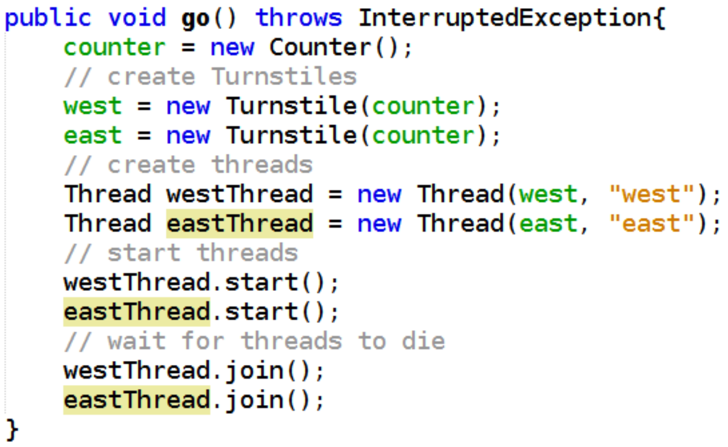
Coding for a Garden



```
@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.

Coding for a Garden

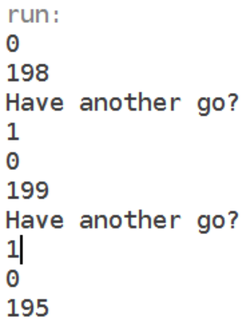
```java
public static void main(String[] args) {
    Garden garden;
    Scanner scan = new Scanner(System.in);
    while(true){
        garden = new Garden(); // create a garden
        // printout initial count
        System.out.println(garden.getCount());
        try {
            garden.go(); // run the garden
        } catch (InterruptedException ex) {
        }
        // count at the end
        System.out.println(garden.getCount());
        // ask to continue
        System.out.println("Have another go?");
        int choice = scan.nextInt();
        if (choice == 0){
            break;
        }
    }
}
```

`main` method displays the count before and after the `go` method in `Garden` class is invoked.

Coding for a Garden

Please note: The Simulate class and the user interactions (`scan.nextInt()` within a `while` loop) are only here for demonstration purposes. We used these to clarify understanding of how the sequence of events unfold. In a real application, we may not use these unless necessary. A similar argument can be made for the use of `sleep` within a Thread.

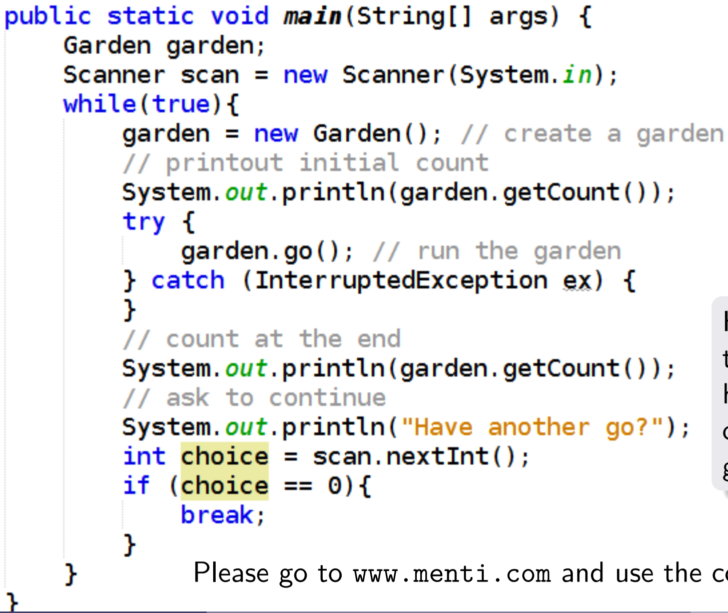
Coding for a Garden



```
run:
0
198
Have another go?
1
0
199
Have another go?
1
0
195
```

Running the program generates arbitrary results for the count at the end, but sometimes it does show the count to be 200. This is a standard example of race condition, and also shows that it may not be easy to identify such errors through formal unit testing.

Coding for a Garden

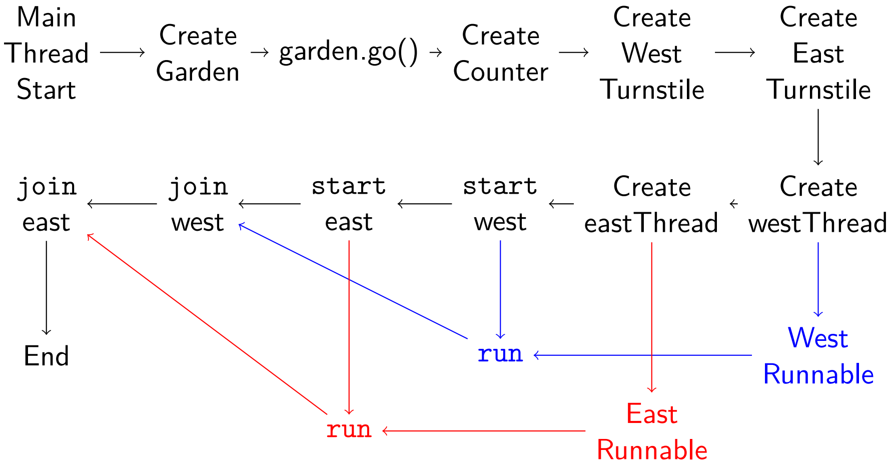


```
public static void main(String[] args) {
    Garden garden;
    Scanner scan = new Scanner(System.in);
    while(true){
        garden = new Garden(); // create a garden
        // printout initial count
        System.out.println(garden.getCount());
        try {
            garden.go(); // run the garden
        } catch (InterruptedException ex) {
        }
        // count at the end
        System.out.println(garden.getCount());
        // ask to continue
        System.out.println("Have another go?");
        int choice = scan.nextInt();
        if (choice == 0){
            break;
        }
    }
}
```

How many threads do we have in the ornamental garden?

Please go to www.menti.com and use the code 11 00 50 4.

Coding for a Garden



Event flow: The main thread creates threads for west and east turnstiles, and invokes respective `run` methods. The `run` method in threads are executed in parallel. We will see if shared counter lead to any issues.

Coding for a Garden

What is happening here? The turnstiles do not know that the other exists, and their job is to look at this scoreboard and add 1. Imagine that *west* looks first and sees the number 4, it goes away to add 1. Now, *east* looks and sees the number 4, and it goes away to add 1. West has now come back and made the scoreboard 5. East comes back afterwards and makes it 5 too (instead of 6, because it does not know that west has updated it already). No synchronisation between their increment actions.

Coding for a Garden

Shared Objects

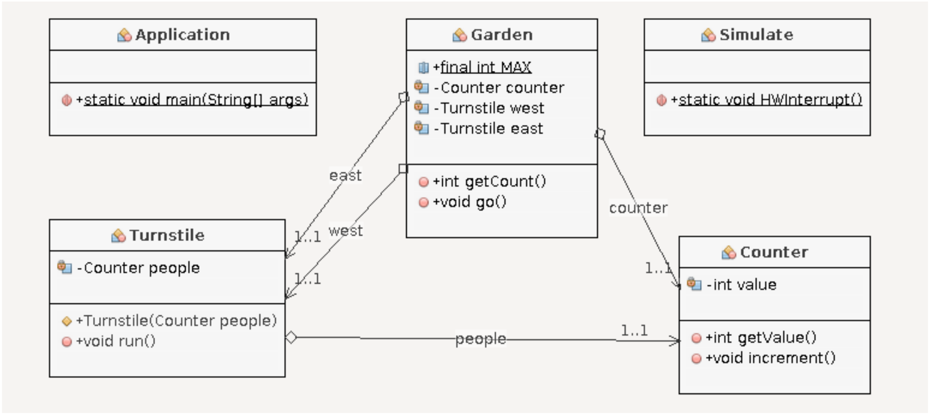
The simplest way for processes to interact is to share objects, and the state of the object can be changed using its methods. Such sharing of resources can lead to interference.

Interference or race condition

The execution of the instructions from a set of threads can be interleaved in an arbitrary fashion. This interleaving may result in incorrect updates to the state of a shared object. This phenomenon is known as *interference* or *race condition*.

Any questions?

Modelling the Garden



We will not model `Application` and `Simulate`. Note that `Garden` – a composite process – is really bringing together `Turnstile` and `Counter` classes to do something useful.

Modelling the Garden

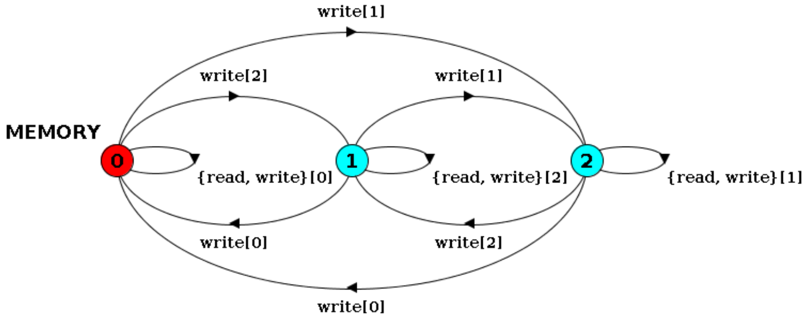
Actions

- `arrive`
- `read`
- `write`
- `go`
- `reset`
- `return`

Processes

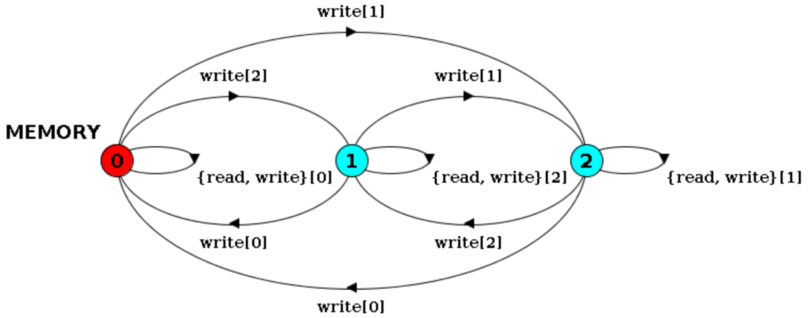
- Memory (Counter)
- Turnstile
- Garden

Modelling the Garden



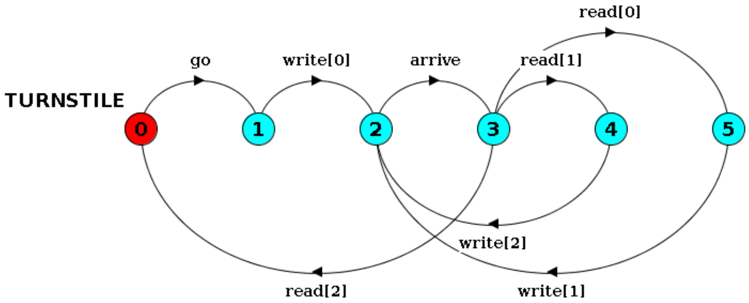
We can consider the `Counter` class we had to be a piece of `Memory`. You can `read` or `write`, and change the state. Here we consider a `Memory` model with three states.

Modelling the Garden



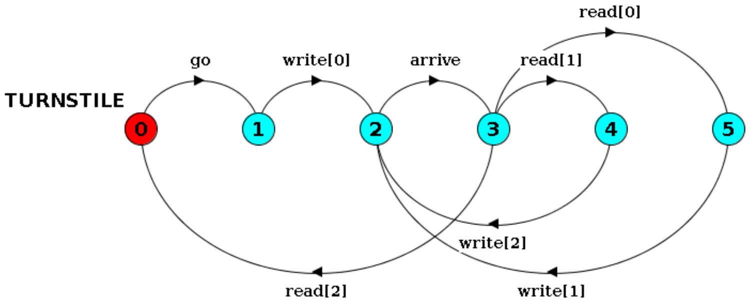
```
const N = 3
T = 0..N
MEMORY = MEMORY[0],
MEMORY[u:T] = (read[u] -> MEMORY[u] | write[v:T] ->
MEMORY[v]).
```

Modelling the Garden



Thinking back the sequence of events in the go method once you called the method were: `reset` everything (like writing 0 in the memory), and then at every `arrive` event in a loop, we read a value and increment that value by adding 1 to it (inside the `increment` method of the `Counter` class).

Modelling the Garden



```
TURNSTILE = (go -> write[0] -> RUN),
RUN = (arrive -> INCREMENT),
INCREMENT = (read[v:0..N-1] -> write[v+1] -> RUN |
read[N] -> TURNSTILE).
```

Modelling the Garden

If we were to now create a composite process `GARDEN`, can we start from the state diagram?

Modelling the Garden

If we were to now create a composite process `GARDEN`, can we start from the state diagram?

How many states would this have?
So, `TURNSTILE` has 6 states, and we have two of those in a `GARDEN`, and then we also share a `MEMORY` with 3 states. Therefore, the total number of states would be at most: $6^2 \times 3 = 2^8$. Too large for us to spend time on.

Modelling the Garden

If we were to now create a composite process GARDEN, can we start from the state diagram?

How many states would this have?
So, TURNSTILE has 6 states, and we have two of those in a GARDEN, and then we also share a MEMORY with 3 states. Therefore, the total number of states would be at most: $6^2 \times 3 = 2^8$. Too large for us to spend time on.

In these circumstances, we will write down the state changes that we expect to happen, and then verify with the animator and alphabets.

Modelling the Garden

The `GARDEN` supposed to have `east` and `west` turnstiles, and a shared counter memory. So, the processes are: `east:TURNSTILE`, `west:TURNSTILE` and `{east, west}::MEMORY`.

```
||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,
west}::MEMORY).
```

Modelling the Garden

The GARDEN supposed to have east and west turnstiles, and a shared counter memory. So, the processes are: east:TURNSTILE, west:TURNSTILE and {east, west}::MEMORY.

```
||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,
west}::MEMORY).
```

Is there a problem with this?

Modelling the Garden

The GARDEN supposed to have east and west turnstiles, and a shared counter memory. So, the processes are: `east:TURNSTILE`, `west:TURNSTILE` and `{east, west}::MEMORY`.

```
||GARDEN = (east:TURNSTILE || west:TURNSTILE || {east,
west}::MEMORY).
```

Is there a problem with this?

Presumably some of the actions are shared? How do we identify that?

Modelling the Garden

We can look at the process alphabets for the components.

Process	Alphabet
east:TURNSTILE	east.{{arrive,go}, {read,write}[0..2]}
west:TURNSTILE	west.{{arrive,go}, {read,write}[0..2]}
{east,west}::MEMORY	{east, west}.{read, write}[0..2]

arrive	should be different for each process.
go	should be common.
read/write	independent actions, except `write[0]` and `read[N]` .

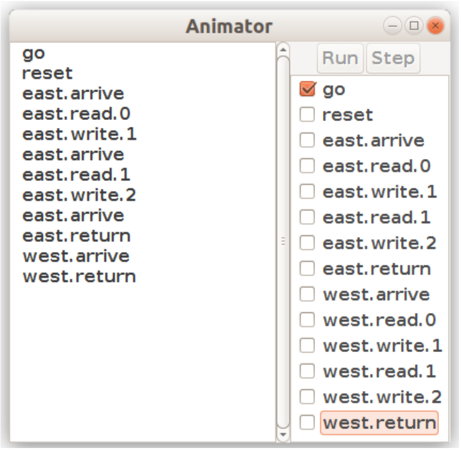
Modelling the Garden

We can look at the process alphabets for the components.

Process	Alphabet
east:TURNSTILE	east.{{arrive,go}, {read,write}[0..2]}
west:TURNSTILE	west.{{arrive,go}, {read,write}[0..2]}
{east,west}::MEMORY	{east, west}.{read, write}[0..2]

arrive	should be different for each process.
go	should be common.
read/write	independent actions, except `write[0]` and `read[N]` .
`write[0]`	this is common as it resets the memory.
`read[N]`	this means that we have written up to N and now ready to return.

Modelling the Garden



```
go
reset
east.arrive
east.read.0
east.write.1
east.arrive
east.read.1
east.write.2
east.arrive
east.return
west.arrive
west.return
```

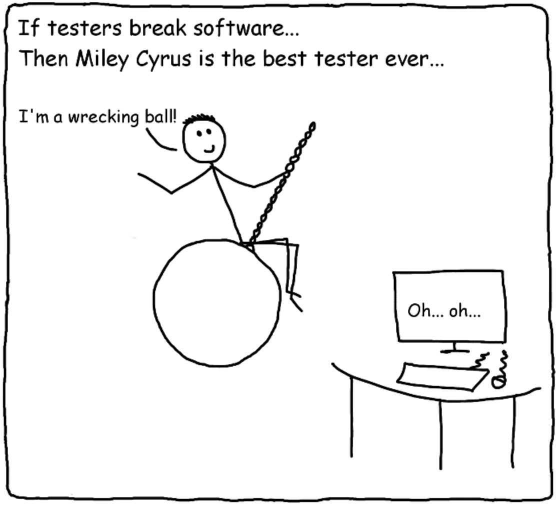
```
✓ go
reset
east.arrive
east.read.0
east.write.1
east.read.1
east.write.2
east.return
west.arrive
west.read.0
west.write.1
west.read.1
west.write.2
west.return
```

```
||GARDEN = (east:TURNSTILE
|| west:TURNSTILE || {east,
west}::MEMORY/{reset/{east,
west}.write[0],
go/{east, west}.go,
east.return/east.read[N],
west.return/west.read[N].
```

Note that our model is slightly simplistic and partial version of the reality. In the real code east and west check for reaching their own counter up to N independently.

Any questions?

Testing for ERROR

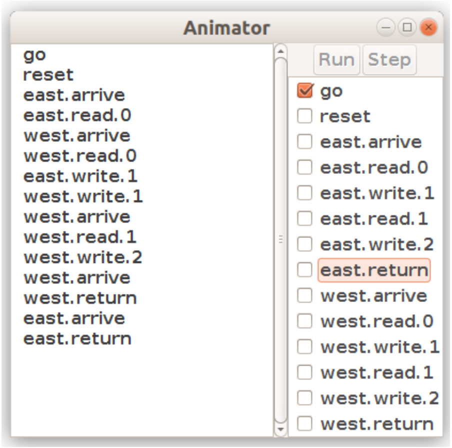

If testers break software...
Then Miley Cyrus is the best tester ever...
I'm a wrecking ball!
Oh... oh...

cartoontester.blogspot.com © 2014

If we analysed the code or FSP we have, it is difficult to say whether race condition is present in it or not. One way we could find out is through animation. But that can be hectic too.
Alternative is to write test code in FSP.

Testing for ERROR

Let’s see how we can identify a problem through animation...



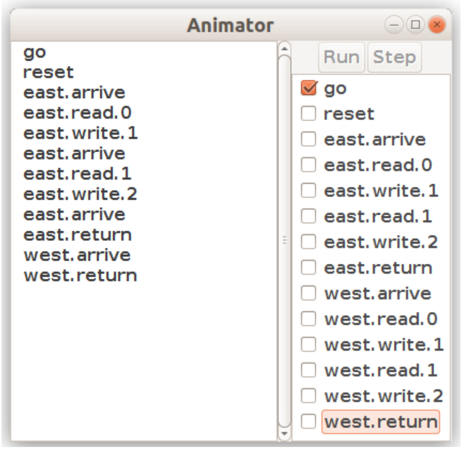
```
go
reset
east.arrive
east.read.0
west.arrive
west.read.0
east.write.1
west.write.1
west.arrive
west.read.1
west.write.2
west.arrive
west.return
east.arrive
east.return
```

- [x] go
- [ ] reset
- [ ] east.arrive
- [ ] east.read.0
- [ ] east.write.1
- [ ] east.read.1
- [ ] east.write.2
- [ ] east.return
- [ ] west.arrive
- [ ] west.read.0
- [ ] west.write.1
- [ ] west.read.1
- [ ] west.write.2
- [ ] west.return

Two parallel `read[0]` followed by two parallel `write[1]`.

Testing for ERROR

How would you test this?



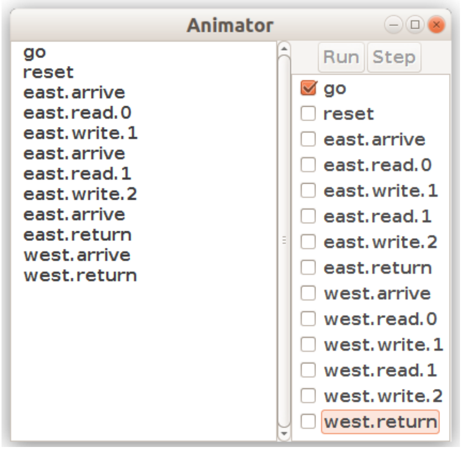
```
go
reset
east.arrive
east.read.0
east.write.1
east.arrive
east.read.1
east.write.2
east.arrive
east.return
west.arrive
west.return
```

[✓] go
[ ] reset
[ ] east.arrive
[ ] east.read.0
[ ] east.write.1
[ ] east.read.1
[ ] east.write.2
[ ] east.return
[ ] west.arrive
[ ] west.read.0
[ ] west.write.1
[ ] west.read.1
[ ] west.write.2
[ ] west.return

Please go to `www.menti.com` and use the code 82 83 79 2

Testing for ERROR

How would you test this?



```
go
reset
east.arrive
east.read.0
east.write.1
east.arrive
east.read.1
east.write.2
east.arrive
east.return
west.arrive
west.return
```

- [x] go
- [ ] reset
- [ ] east.arrive
- [ ] east.read.0
- [ ] east.write.1
- [ ] east.read.1
- [ ] east.write.2
- [ ] east.return
- [ ] west.arrive
- [ ] west.read.0
- [ ] west.write.1
- [ ] west.read.1
- [ ] west.write.2
- [ ] west.return

Please go to `www.menti.com` and use the code 82 83 79 2

Best option: count `write` actions; more than 2 means error has occurred.

Any questions?

Summary

- Arbitrary interleaving – on which we have no control – can cause interference.
- We model the behaviours and identify where the problem is through FSP and testing.