Applications of Amdahl's Law, & Properties I

Lecture 15

Alma Rahat

CS-210: Concurrency

16 March 2021

What did we do in the last session?

- Deadlock handling: An OS developer's perspective.
  - Avoidance
  - Detection and Recovery
- Resource Allocation Graphs.
- Introduction to Amdahl's law.

In this lecture...

Learning outcomes.

1. To apply Amdahl's law to scenarios and determine appropriate actions for program optimisation.
2. To define a property of a program.
3. To apply safety property in FSP model development and analyse the system.

Outline.

1. Important insights from Amdahl's law.
2. Properties of a program.
3. single lane bridge.
4. Another example: Washing Machine.

Important Observations Using Amdahl's Law

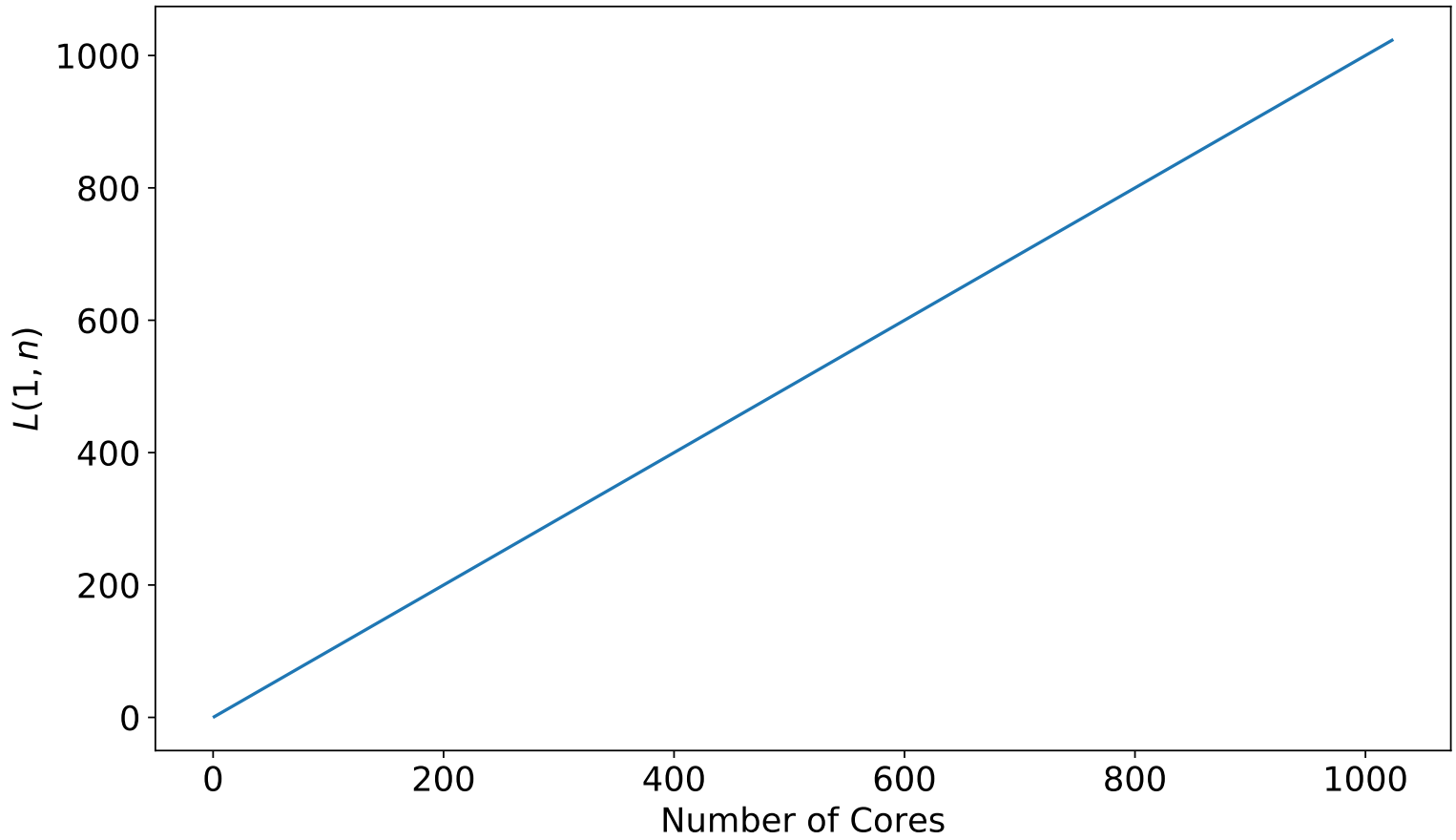
Swansea University
Prifysgol Abertawe

Given that: $L(k, n) \leq \frac{1}{\frac{1}{k}\tilde{t_s}+\frac{1}{n}\tilde{t_p}}$, what happens if $\tilde{t_s} = 0$?

Here, we are assuming there are no serial part at all, and the parallel part is completely parallelisable.

Important Observations Using Amdahl's Law

Given that: $L(k, n) \leq \frac{n}{\tilde{t}_p}$, a linear increase in speed up.



Important Observations Using Amdahl's Law

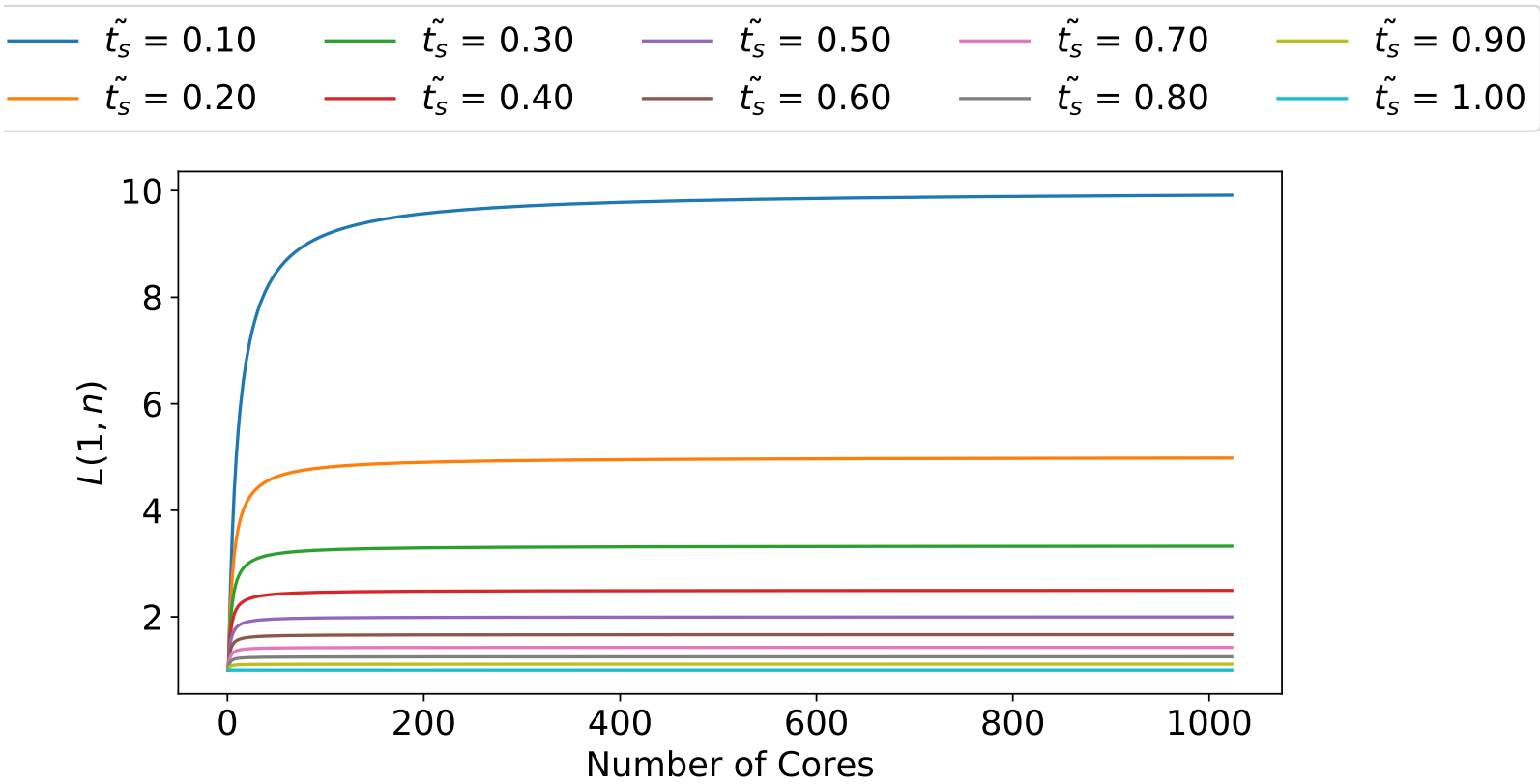
Swansea University Prifysgol Abertawe

Given that: $L(k, n) \leq \frac{1}{\frac{1}{k}\tilde{t}_s + \frac{1}{n}\tilde{t}_p}$, what happens if we vary $\tilde{t}_s \in \{0.1, \ldots, 1\}$?

Here, we want to see how adding cores can affect the performance, given a certain amount of serial part in a program.

Important Observations Using Amdahl's Law

Given that: $L(1,n) \leq \frac{1}{\tilde{t_s}+\frac{1}{n}\tilde{t_p}}$, the benefit diminishes as you add more and more processors, and depending on $\tilde{t_s}$ it may not be beneficial to do this at all.



Important Observations Using Amdahl's Law

Given that: $L(k,n) \leq \frac{1}{\frac{1}{k}\tilde{t_s}+\frac{1}{n}\tilde{t_p}}$, what happens if we have *infinite* number of processors, i.e. $n = \infty$?

Here, we want to see what is the upper bound of speed up for us given that we have unlimited number of cores.

Important Observations Using Amdahl's Law

Given that: $L(k,n) \leq \frac{1}{\frac{1}{k}\tilde{t_s}+\frac{1}{n}\tilde{t_p}}$, what happens if we have *infinite* number of processors, i.e. $n = \infty$?

Here, we want to see what is the upper bound of speed up for us given that we have unlimited number of cores.

$\frac{1}{n}\tilde{t_p} = 0$, so, $L(k,n) \leq \frac{k}{\tilde{t_s}}$. This is the upper bound of improvement.

Any questions?

Exercise

What is the upper bound of speed up due to 12 cores when your program has 40% of parallelisable part?

Please go to `www.menti.com` and use the code 3521 0273.

Exercise

What is the upper bound of speed up due to 12 cores when your program has 40% of parallelisable part?

Please go to `www.menti.com` and use the code 3521 0273.

$\Rightarrow$ Here, $k = 1$, $n = 12$, $\tilde{t_p} = 0.4$ and $\tilde{t_s} = 1 - \tilde{t_p} = 0.6$. So, the upper bound is $L(k, n) \leq \frac{1}{0.6 + \frac{1}{12} 0.4} \approx 1.58$.

Exercise

Swansea University
Prifysgol Abertawe

You come across a program that has 20% sequential part. Now, you are told that any of the following would require the same amount of time and person hours:

1. Double performance for sequential part, and triple the performance of the parallelisable part.
2. Only quadruple the performance of parallelisable part.
3. Only quadruple the serial part. Which one of the above option would you go for?

Please go to `www.menti.com` and use the code 1427 6652.

Exercise

You come across a program that has 20% sequential part. Now, you are told that any of the following would require the same amount of time and person hours:

1. Double performance for sequential part, and triple the performance of the parallelisable part.
2. Only quadruple the performance of parallelisable part.
3. Only quadruple the serial part. Which one of the above option would you go for?

Please go to `www.menti.com` and use the code 1427 6652.

$\Rightarrow$ Each of the above has the following performance gain respectively: 2.72, 2.17 and 1.18. So, you should go for the first option.

Any questions?

Properties of a Program

Properties of a Program

A property is an attribute of a program that is true for every possible execution of that program, i.e. at all possible states.

Properties of interest for concurrent programs fall into two categories:

Safety property asserts that nothing bad (e.g. interference or deadlock) happens during execution. This must be true for all states.

Liveness property asserts that something good (e.g. a result, fairness and no restriction to progress) eventually happens. This is eventually true. We will generally concern ourselves with Liveness.

Safety and Liveness Properties

Swansea University
Prifysgol Abertawe

Safety property holds in all states – nothing bad happens.

Liveness property eventually holds – something good happens.

Examples.

- Safety:
  - Deadlock free
  - Mutual exclusion
- Liveness
  - A result.
  - Fairness.
  - Progress.

Safety Properties

To check the safety property of a process `P`, we define a deterministic process `property Q`, and do the following:

1. Create a composite process `||R = (P ||Q)` .
2. Check whether `R` has paths to an error state or not (much like testing).

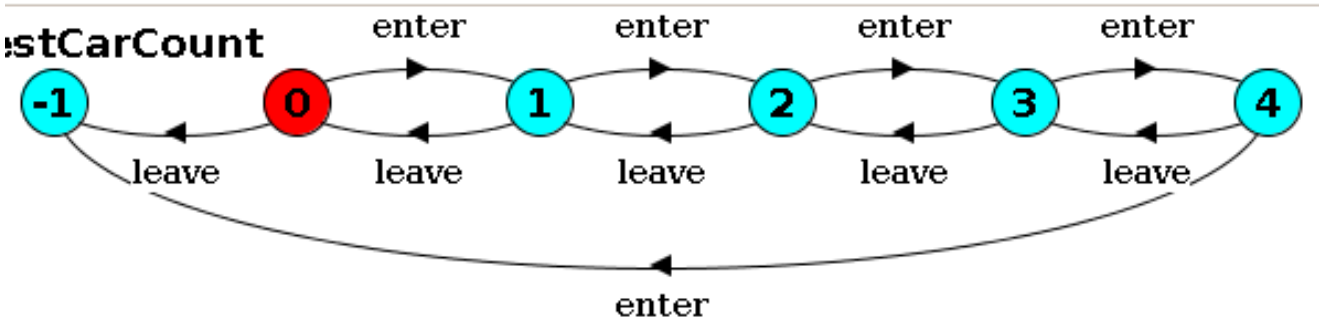
Car Park Example Revisited

```
Entrance = (enter -> Entrance).
Exit = (leave -> Exit).
Controller(Capacity=4) = Spaces[Capacity],
Spaces[spaceLeft:0..Capacity] =
(when spaceLeft>0 enter -> Spaces[spaceLeft-1]
|when spaceLeft<Capacity leave -> Spaces[spaceLeft+1]).
||CarPark = (Entrance || Controller || Exit).
// Property for checking the number of total cars
property TotalCars = TotalCars[0],
TotalCars[i:0..4] = (enter -> TotalCars[i+1] | leave ->
TotalCars[i-1]).
||TestCarCount = (CarPark || TotalCars).
```

In this case, there will be no errors, as we have used conditions to guard against these. What would happen if the guards were not there?

Car Park Example Revisited

```
Entrance = (enter -> Entrance).
Exit = (leave -> Exit).
Controller(Capacity=4) = Spaces[Capacity],
Spaces[spaceLeft:0..Capacity] =
(when spaceLeft>0 enter -> Spaces[spaceLeft-1]
|when spaceLeft<Capacity leave -> Spaces[spaceLeft+1]).
||CarPark = (Entrance || Controller || Exit).
// Property for checking the number of total cars
property TotalCars = TotalCars[0],
TotalCars[i:0..4] = (enter -> TotalCars[i+1] | leave ->
TotalCars[i-1]).
||TestCarCount = (CarPark || TotalCars).
```



Any questions?

Single Lane Bridge Problem

Do you know this symbol?

Source: ISBN 9780115528552 Please go to `www.menti.com` and use the code 6571 3500.

Single Lane Bridge Problem

Usually we use it to show that we have a narrow lane ahead...

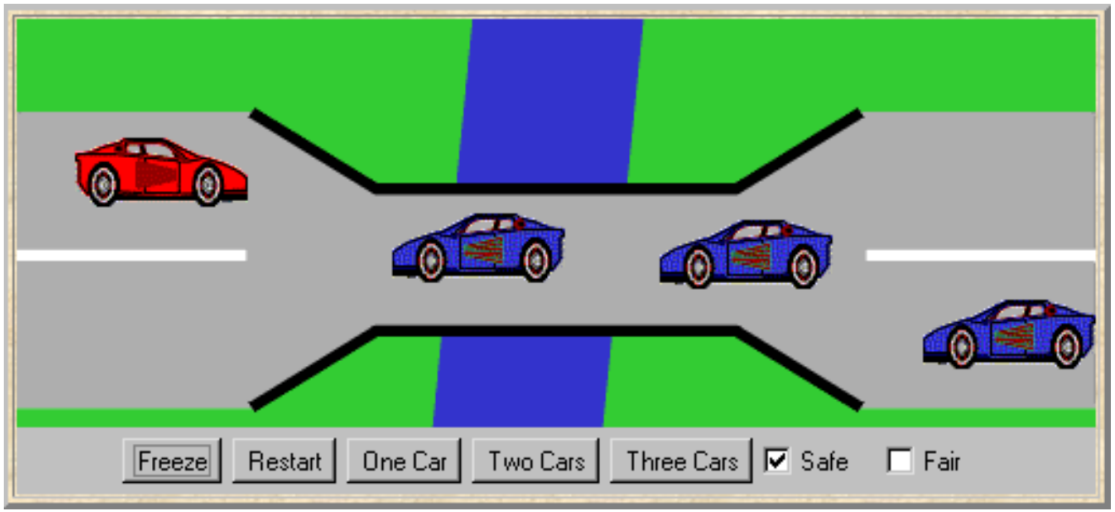
Source: geograph.org.uk

Single Lane Bridge Problem

These lanes can be tricky. We need to ensure safety.

Source: birminghammail.co.uk

Single Lane Bridge Problem



Source: Chapter 7, Magee & Kramer.

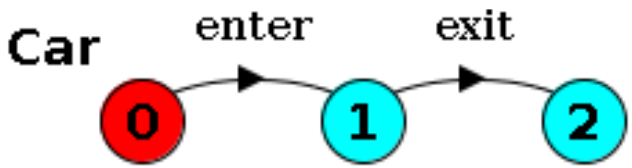
A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in the different directions enter the bridge at the same time.

Single Lane Bridge Problem

- Actions: `enter` and `exit`.
- Processes:
  - Active: `Car`, `Convoy`, `Cars`.
  - Passive: `Bridge`.
- Safety properties: `EntranceOrder`, `ExitOrder` and `SingleCarOnBridge`.

Single Lane Bridge Problem

```
Car = (enter -> exit -> STOP).
```

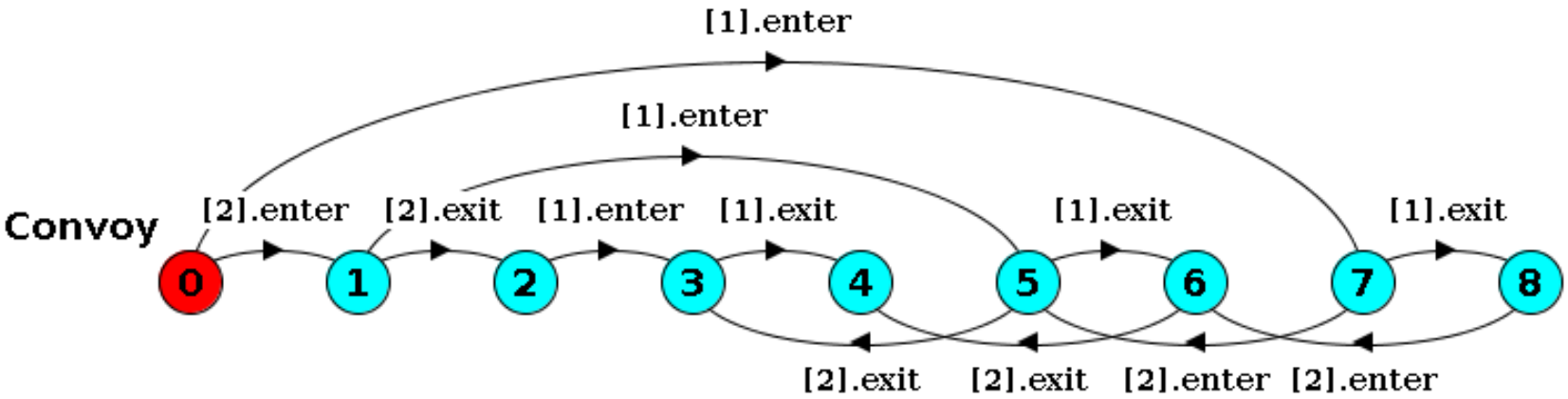


The `Car` model is simple: it just `enters` and then `exits` the bridge. Note that this is only the building block for `Convoy` and `Cars`.

Single Lane Bridge Problem

A `Convoy` is a queue of cars.

```
const N = 2
range T = 0..N
range ID = 1..N  // lower ID is for the first car on the queue.
Car = (enter -> exit -> STOP).
||Convoy = ([ID]:Car).  // composing multiple cars with IDs.
```

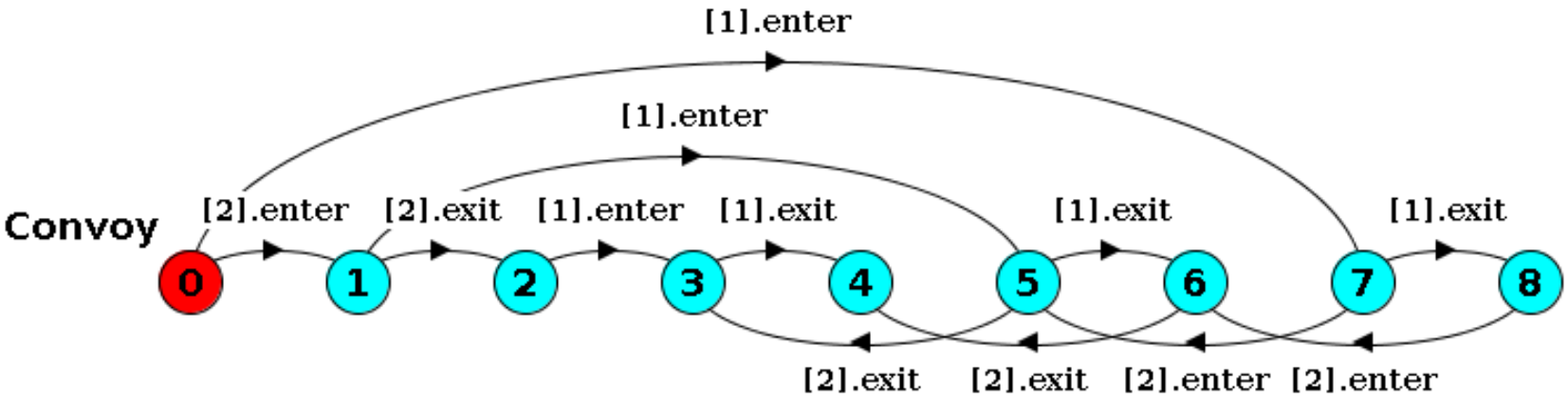


We now have two cars: `[1]` and `[2]`, each with alphabets `enter` and `exit`.
Any issues? Please go to `www.menti.com` and use the code 78 73 05 3

Single Lane Bridge Problem

A `Convoy` is a queue of cars.

```
const N = 2
range T = 0..N
range ID = 1..N  // lower ID is for the first car on the queue.
Car = (enter -> exit -> STOP).
||Convoy = ([ID]:Car).  // composing multiple cars with IDs.
```

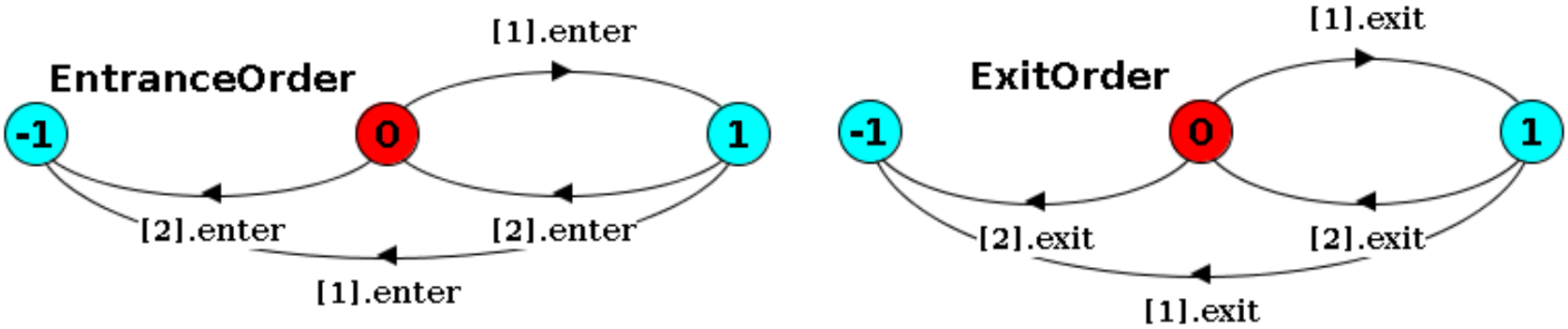


We now have two cars: `[1]` and `[2]`, each with alphabets `enter` and `exit`.Clearly violates the order of entrance and exit.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```



The properties ensures that any invalid sequence of actions is leading to ERROR.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```

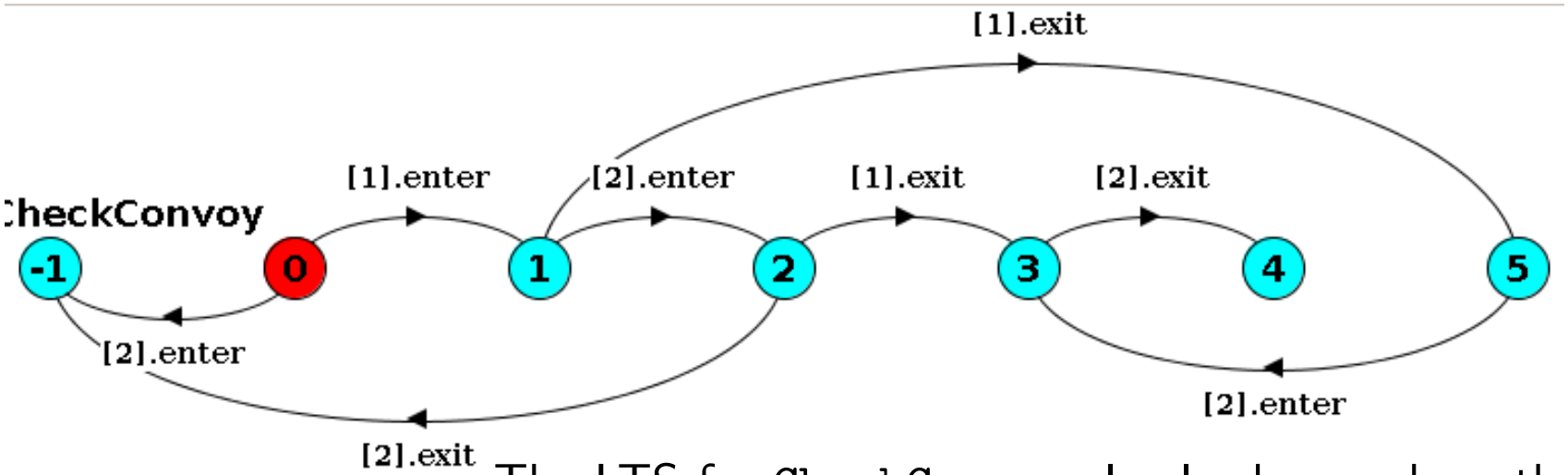
```
Composing...
  property EntranceOrder violation.
  property ExitOrder violation.
  potential DEADLOCK
```

While composing `CheckConvoy`, we can clearly see an example trace to `ERROR`. Note that the deadlock here is simply the end of program, so nothing to worry about.

Single Lane Bridge Problem

We will define a couple of safety properties for *entrance* and *exit*.

```
property EntranceOrder = EntranceOrder[1],
EntranceOrder[id:ID] = ([id].enter ->
EntranceOrder[id%N+1]).
property ExitOrder = ExitOrder[1],
ExitOrder[id:ID] = ([id].exit -> ExitOrder[id%N+1]).
||CheckConvoy = (Convoy || EntranceOrder || ExitOrder).
```

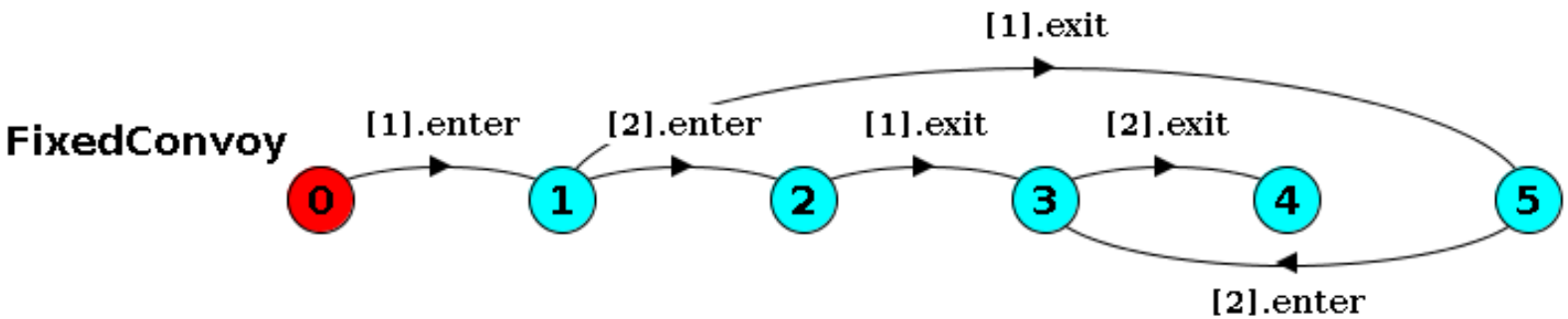


The LTS for `CheckConvoy` clearly shows where the problems are.

Single Lane Bridge Problem

We will create a couple of processes that allow us to model *entrance* and *exit* and associated order of events.

```
Entrance = Entrance[1],
Entrance[i:ID] = ([i].enter -> Entrance[i%N+1]).
Exit = Exit[1],
Exit[i:ID] = ([i].exit -> Exit[i%N+1]).
||FixedConvoy = ([ID]:Car || Entrance || Exit).
||CheckFixedConvoy = (FixedConvoy || EntranceOrder ||
ExitOrder). // for checking the properties.
||Cars = ({west, east}:FixedConvoy).// creating the Cars.
```



Single Lane Bridge Problem

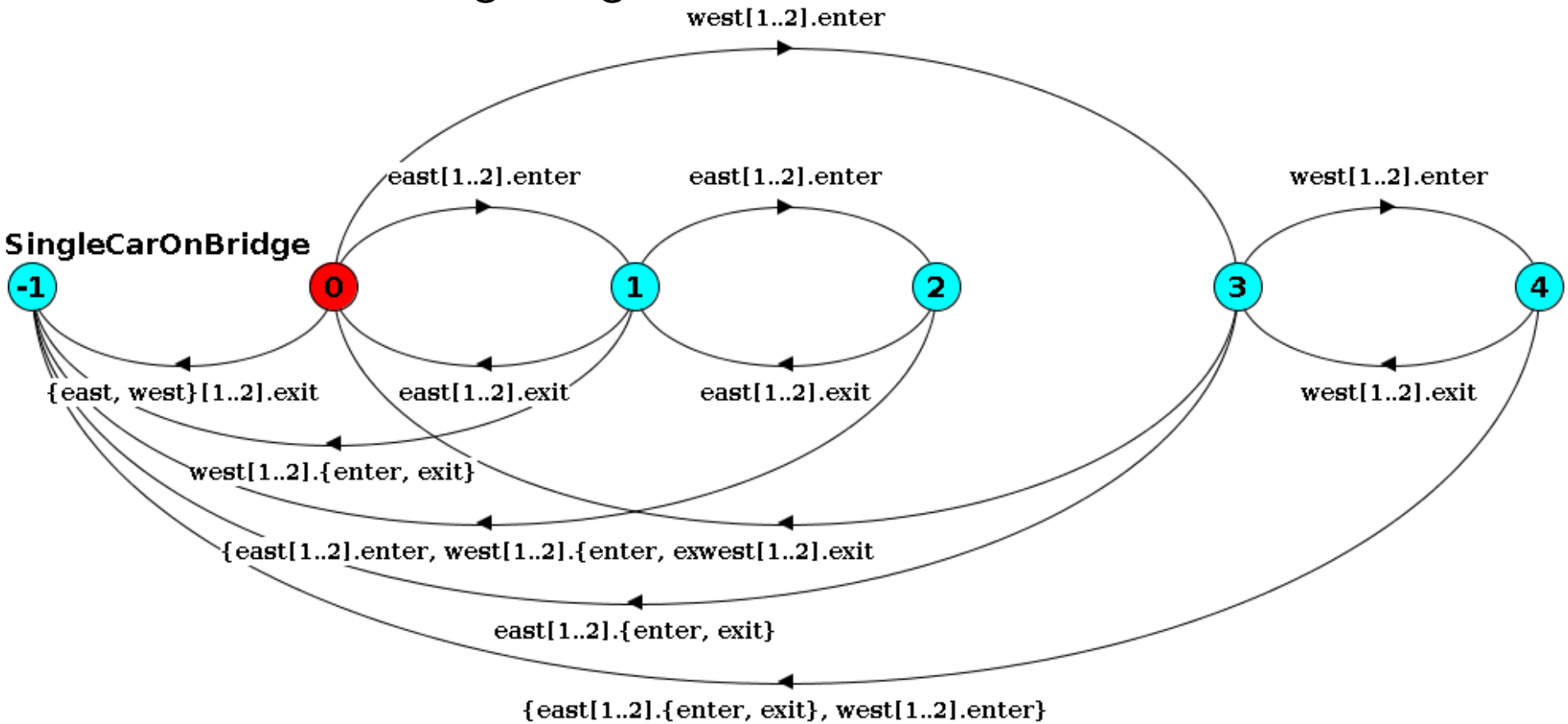
We will now create the property that check that there is only one car on the bridge.

```
property SingleCarOnBridge = (west[ID].enter -> CountWest[1]
                            | east[ID].enter -> CountEast[1]),
CountWest[i:ID] = (west[ID].enter -> CountWest[i+1]
                  | when i>1 west[ID].exit -> CountWest[i-1]
                  | when i==1 west[ID].exit -> SingleCarOnBridge),
CountEast[i:ID] = (east[ID].enter -> CountEast[i+1]
                  | when i>1 east[ID].exit -> CountEast[i-1]
                  | when i==1 east[ID].exit -> SingleCarOnBridge).
||CheckCars = (Cars||SingleCarOnBridge).
```

```
Trace to property violation in SingleCarOnBridge:
        east.1.enter
        west.1.enter
```

Single Lane Bridge Problem

The LTS shows that once we have allowed *east cars* to go ahead, any *west cars* entering will generate an ERROR and *vice versa*.

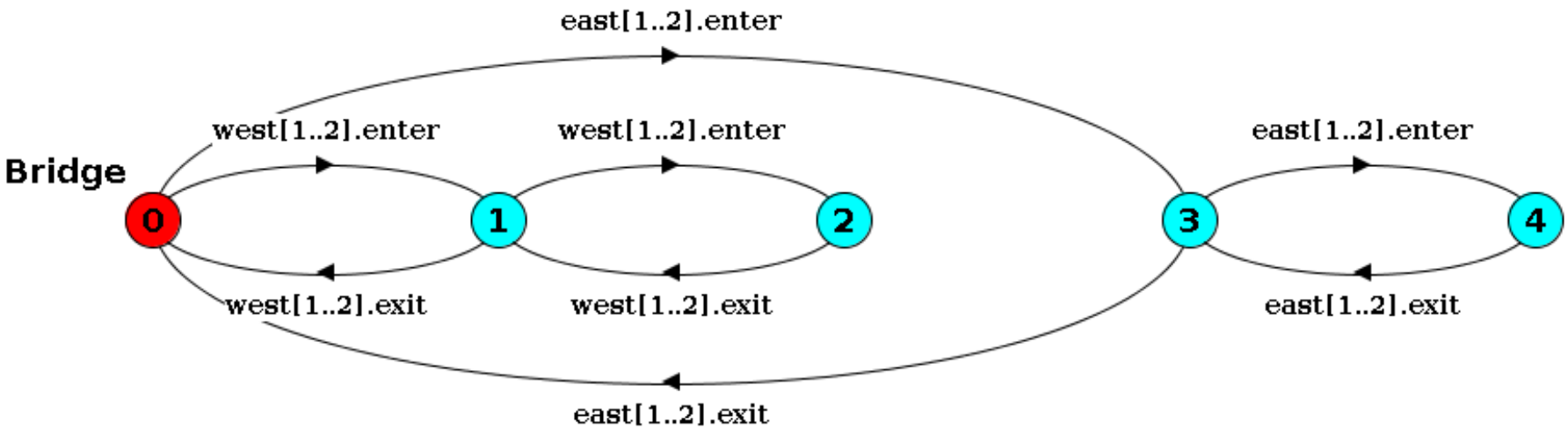


Single Lane Bridge Problem

It is time for us to model the `Bridge`.

```
Bridge = Bridge[0][0],
Bridge[nWest:T][nEast:T] = (
when (nWest == 0 && nEast<N) east[ID].enter -> Bridge[nWest][nEast + 1]
| when (nEast>0) east[ID].exit -> Bridge[nWest][nEast - 1]
| when (nEast==0 && nWest<N) west[ID].enter -> Bridge[nWest+1][nEast]
| when (nWest>0) west[ID].exit -> Bridge[nWest-1][nEast]
).

||SingleLane = (Cars || Bridge).
||CheckSingleLane = (SingleLane || SingleCarOnBridge).
```



Any questions?

Coding the Bridge

```
public class Bridge {
    private int westCount = 0;
    private int eastCount = 0;
    public synchronized void westEnter()
            throws InterruptedException{
        while (eastCount > 0) wait();
        westCount += 1; notifyAll();
    }
    public synchronized void westExit(){
        westCount -= 1;
    }
    public synchronized void eastEnter()
            throws InterruptedException{
        while (westCount > 0) wait();
        eastCount += 1; notifyAll();
    }
    public synchronized void eastExit(){
        westCount += 1;
    }
}
```

A simple Bridge class without any queues for cars from west and east; just using simple counters.

Coding the Bridge

```
public class BridgeQueue {
    private Queue<Integer> westQueue;
    private Queue<Integer> eastQueue;
    private int westCount;
    private int eastCount;
    BridgeQueue(){
        westQueue = new LinkedList<Integer>();
        eastQueue = new LinkedList<Integer>();
        westCount = 0;
        eastCount = 0;
    }
```

An improved version, where we use queues instead to hold cars from west and east. Note that by using linked list first-in-first-out queue implementation, we are making sure that an element that enters the queue first, exits first.

Coding the Bridge

```
public synchronized void westEnter()
        throws InterruptedException{
    while (eastCount > 0)
            wait();
    westCount += 1;
    westQueue.add(westCount);
    System.out.println("Added to the west queue: " + westCount);
    notifyAll();
}
public synchronized void westExit(){
    westCount -= 1;
    westQueue.remove();
    System.out.println("Removed from the west queue: " + westCount);
}
```

It has methods for *westEnter* and *westExit*, adding and removing from the shared `westQueue` respectively.

Coding the Bridge

```
public synchronized void eastEnter()
        throws InterruptedException{
    while (westCount > 0)
            wait();
    eastCount += 1;
    eastQueue.add(eastCount);
    System.out.println("Added to the east queue: " + eastCount);
    notifyAll();
}
public synchronized void eastExit(){
    westCount += 1;
    eastQueue.remove();
    System.out.println("Removed from the east queue: " + eastCount);
}
```

Similar methods for *eastEnter* and *eastExit*, adding and removing from the shared `eastQueue` respectively.

Coding the Bridge

```
public class WestCar implements Runnable {
    private BridgeQueue bridge;
    private String name;
    WestCar(String name, BridgeQueue bridge){
        this.bridge = bridge;
        this.name = name;
    }
```

Cars are coming from west and east. So, we are going to implement each west and east as Threads. Here, we have the `WestCar` implementation: it has the shared `BridgeQueue` instance, and also a `name`.

Coding the Bridge

```
@Override
public void run() {
    Random random = new Random();
    int randomInt = 0;
    while (true){
        try {
            System.out.println(name + " is adding to west queue.");
            bridge.westEnter();
            randomInt = random.nextInt(1000); // upto 1 sec
            Thread.sleep(randomInt);
            bridge.westExit();
            System.out.println(name + " has removed from west queue.");
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted. ");
            break;
        }
    }
}
```

When this `WestCar` is run, it randomly queues a car, and waits for a bit before removing it from the queue.

Coding the Bridge

```
public class EastCar implements Runnable{
    private BridgeQueue bridge;
    private String name;
    EastCar(String name, BridgeQueue bridge){
        this.bridge = bridge;
        this.name = name;
    }
```

`EastCar` is the same in principle, but works on the shared `eastQueue`.

Coding the Bridge

```
@Override
public void run() {
    Random random = new Random();
    int randomInt = 0;
    while (true){
        try {
            System.out.println(name + " is adding to east queue.");
            bridge.eastEnter();
            randomInt = random.nextInt(1000); // upto 1 sec
            Thread.sleep(randomInt);
            bridge.eastExit();
            System.out.println(name + " has removed from east queue.");
        } catch (InterruptedException ex) {
            System.out.println(name + " was interrupted. ");
            break;
        }
    }
}
```

`EastCar` is the same in principle, but works on the shared `eastQueue`.

Coding the Bridge

```
public static void main(String[] args)
        throws InterruptedException {
    BridgeQueue bridge = new BridgeQueue();
    WestCar westCarA = new WestCar("WA", bridge);
    WestCar westCarB = new WestCar("WB", bridge);
    EastCar eastCarA = new EastCar("EA", bridge);
    EastCar eastCarB = new EastCar("EB", bridge);
    Thread twa = new Thread(westCarA);
    Thread twb = new Thread(westCarB);
    Thread tea = new Thread(eastCarA);
    Thread teb = new Thread(eastCarB);
    twa.start();twb.start();tea.start();teb.start();
    Thread.sleep(5000);// sleep for a while
    twa.interrupt();twb.interrupt();tea.interrupt();teb.interru
    twa.join();twb.join();tea.join();teb.join();
    System.out.println("Program has ended.");
}
```

In the main program, we just create the `BridgeQueue` and create four different threads to work on it.

Coding the Bridge

```
run:
WB is adding to west queue.
EA is adding to east queue.
WA is adding to west queue.
EB is adding to east queue.
Added to the west queue: 1
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
WB was interrupted.
EA was interrupted.
WA was interrupted.
EB was interrupted.
Program has ended.
```

This is the output of the program. Is there anything surprising in the output?

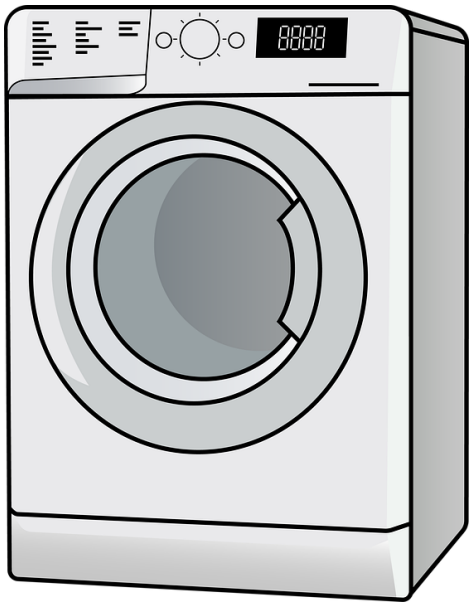
Coding the Bridge

```
run:
WB is adding to west queue.
EA is adding to east queue.
WA is adding to west queue.
EB is adding to east queue.
Added to the west queue: 1
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WA has removed from west queue.
WA is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
Removed from the west queue: 1
WB has removed from west queue.
WB is adding to west queue.
Added to the west queue: 2
WB was interrupted.
EA was interrupted.
WA was interrupted.
EB was interrupted.
Program has ended.
```

This is the output of the program. Is there anything surprising in the output? EA and EB never gets to go through the bridge and add to the queue, as cars continue to come from the west. This is called *starvation*, i.e. some of the threads never get to do anything, and keep waiting on others.

Any questions?

Another Example



You are given a scenario for a *washing machine*: in this machine, you can turn on and open the door to load it, then close the door, and start the process. The process includes: washing clothes, rinseing them, and finally drying them before returning to the initial state. At this point, you should be able to open the door and Once you have started the washing process, at any state, you should be able to pause the machine, which takes you to the Paused state, and you would use unpause to resume operations.

There is a safety *cycle* property: wash must come before rinse, and both must happen before dry.

Produce FSP code for the model and the property.

Working through the problem

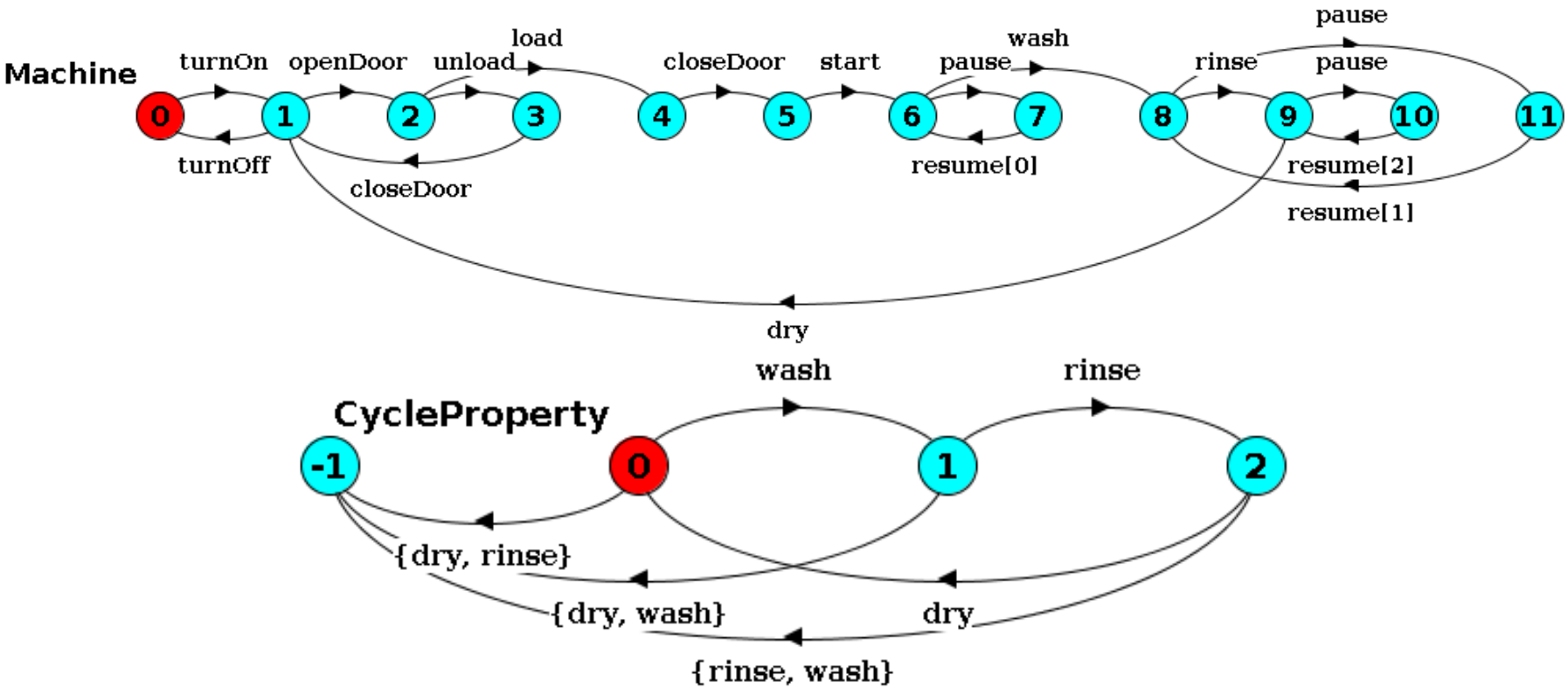
LTSs

```
property CycleProperty = (wash -> rinse -> dry -> CycleProperty).

Machine = (turnOn -> Initial),
Initial = (openDoor -> (load -> closeDoor -> start -> WashProcess[0]
             | unload -> closeDoor -> Initial)
             | turnOff -> Machine),
WashProcess[i:0..2] = (
  when i==0 wash -> WashProcess[i+1]
| when i==1 rinse -> WashProcess[i+1]
| when i==2 dry -> Initial
| pause -> Pause[i]),
Pause[i:0..2] = (resume[i] -> WashProcess[i]).

||CycleTest = (Machine || CycleProperty).
```

LTSs


Machine
turnOn
openDoor
unload
load
closeDoor
start
pause
wash
rinse
pause
pause
turnOff
closeDoor
resume[0]
resume[2]
resume[1]
dry
0
1
2
3
4
5
6
7
8
9
10
11
CycleProperty
wash
rinse
-1
0
1
2
{dry, rinse}
{dry, wash}
dry
{rinse, wash}

Summary

- Safety property asserts cases that must be true for all states throughout the LTS.
- Explicitly describing these properties help us ensure that we are not able to reach a **bad** state, and LTSA will automatically tell us if we did reach a **bad** state during compilation.