Properties III

Lecture 17

Alma Rahat

CS-210: Concurrency

23 March 2021

Swansea University
Prifysgol Abertawe

What did we do in the last session?

- Safety properties: single lane bridge
- Washing machine

In this lecture...

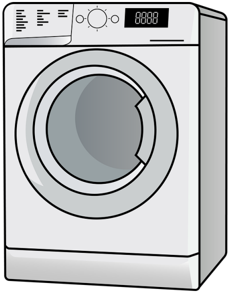
Learning outcomes.

1. To apply safety properties in FSP model development and analyse the system.
2. To apply modelling techniques to identify progress issues.
3. To devise appropriate ways to eliminate progress issues.

Outline.

1. Washing machine.
2. Progress and fairness.
3. Example: tossing a coin.
4. Terminal sets.
5. Fixing Single Lane Bridge.

Example: Washing Machine and Safety



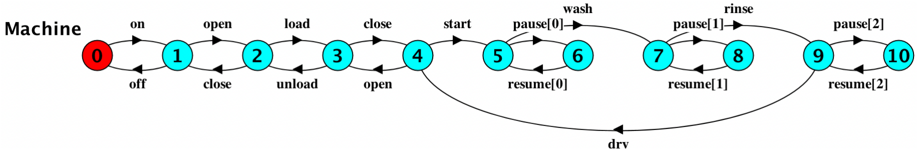
In a washing machine, you can `turn it on` and then `turn it off`. Once turned on, it has a `closed door`. At this stage, you can `open` the machine, and find it to be `empty`. Now, you can `load` it and make it `full`. When full, you can `unload` it and make it go back to original `empty` state. When it is full, you can close the door, and `start` the washing process, where you `wash`, `rinse`, and subsequently `dry`. At any point during the washing process, you should be able to `pause` and `resume` current operation.

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 safety property.

Working through the problem

In a washing machine, you can `turn it on` and then `turn it off`. Once turned on, it has a `closed door`. At this stage, you can `open` the machine, and find it to be `empty`. Now, you can `load` it and make it `full`. When full, you can `unload` it and make it go back to original `empty` state. When it is full, you can close the door, and `start` the washing process, where you `wash`, `rinse`, and subsequently `dry`. At any point during the washing process, you should be able to `pause` and `resume` current operation.

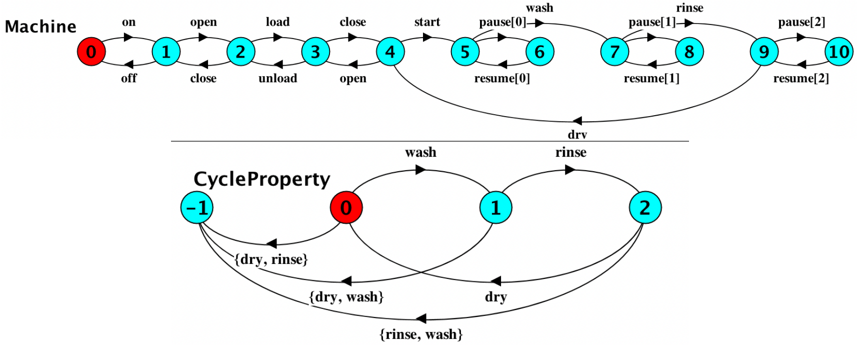


FSP

```
Machine = (on -> ClosedDoor[0]),
ClosedDoor[i:0..1] = (when i==0 open -> Empty |
                      when i==0 off -> Machine |
                      when i==1 open -> Full |
                      when i==1 start -> Cycle[0]),
Full = (unload -> Empty |
        close -> ClosedDoor[1]),
Empty = (load -> Full | close -> ClosedDoor[0]),
Cycle[i:0..2] = (when i==0 wash -> Cycle[i+1] |
                 when i==1 rinse -> Cycle[i+1] |
                 when i==2 dry -> ClosedDoor[1]|
                 pause[i] -> Paused[i]),
Paused[i:0..2] = (resume[i] -> Cycle[i]).

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

LTS



Machine
on
off
open
close
load
unload
close
open
start
wash
pause[0]
resume[0]
rinse
pause[1]
resume[1]
pause[2]
resume[2]
dry
0
1
2
3
4
5
6
7
8
9
10
CycleProperty
wash
rinse
-1
0
1
2
{dry, rinse}
{dry, wash}
dry
{rinse, wash}

Any questions?

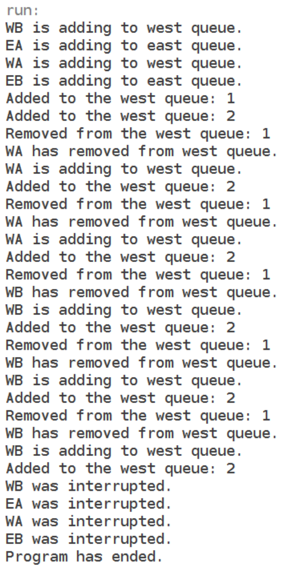
Safety and Liveness Properties

Safety	property holds in all states – nothing bad happens.
Liveness	property eventually holds – something good happens.

Examples.

- Safety: ✓
  - Deadlock free
  - Mutual exclusion
- Liveness (next)
  - A result.
  - Fairness.
  - Progress.

Revisiting Single Lane Bridge Problem



```
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.
```

We modelled and coded the system to guard against the following safety issues:

- Maintains entrance and exit orders.
- Does not allow face-to-face crashes.

The output of the program revealed something surprising: 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. This is lack of *progress* due to the lack of *fairness*.

Progress and Fairness

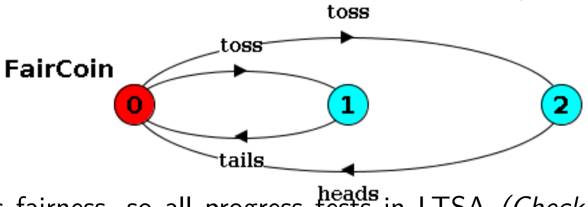
Progress	an action always eventually gets executed.
Fairness	If a choice over a set of transitions is made infinitely often, then every transition in the set will be chosen infinitely often.

```
progress P = {a_1, ..., a_n}
```

Defines a progress property P: At least one of the actions $a_1, \dots, a_n$ will be executed infinitely often.

Progress: Tossing a Coin

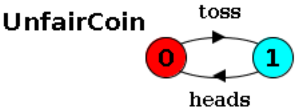
```
FairCoin = (toss -> heads -> FairCoin | toss -> tails ->
FairCoin). // a simple coin that produces heads or tails.
progress Heads = {heads} //tests that heads can occur
infinitely many times
progress Tails = {tails} /tests that heads can occur
infinitely many times
progress HeadsOrTails = {heads, tails} //tests that at
least one of heads or tails occur infinitely many times.
```



FSP assumes fairness, so all progress tests in LTSA *(Check → Progress)* pass.

Progress: Tossing a Coin

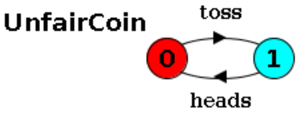
```
UnfairCoin = (toss -> heads -> FairCoin). // a simple unfair
coin that produces heads only.
progress Heads = {heads} //tests that heads can occur
infinitely many times
progress Tails = {tails} /tests that heads can occur
infinitely many times
progress HeadsOrTails = {heads, tails} //tests that at
least one of heads or tails occur infinitely many times.
```

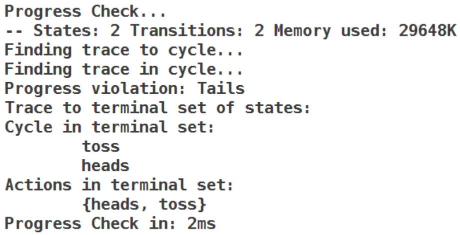


Are all progress properties satisfied?

Progress: Tossing a Coin

```
progress Heads = {heads} ✓
progress Tails = {tails} ✗
progress HeadsOrTails = {heads, tails} ✓
```



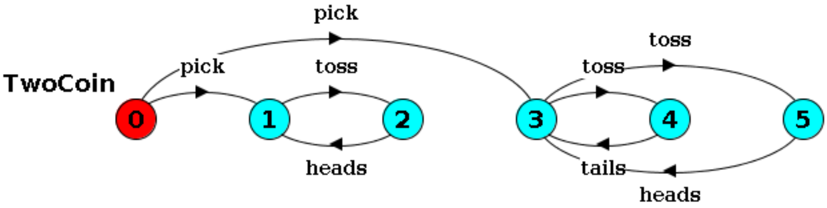


```
Progress Check...
-- States: 2 Transitions: 2 Memory used: 29648K
Finding trace to cycle...
Finding trace in cycle...
Progress violation: Tails
Trace to terminal set of states:
Cycle in terminal set:
        toss
        heads
Actions in terminal set:
        {heads, toss}
Progress Check in: 2ms
```

Clearly, there are no possibility of *tails* occurring. More on terminal sets later.

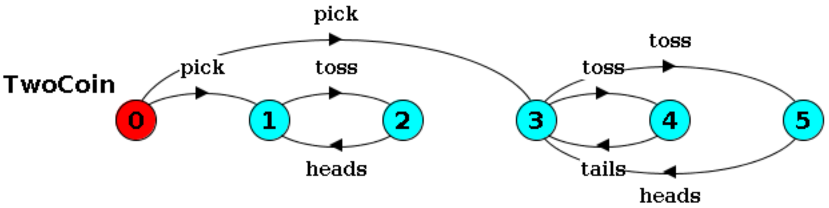
Progress: Tossing a Coin

```
TwoCoin = (pick -> FairCoin | pick -> UnfairCoin),
FairCoin = (toss -> heads -> FairCoin
| toss -> tails -> FairCoin),
UnfairCoin = (toss -> heads -> UnfairCoin).
progress Heads = {heads}
progress Tails = {tails}
progress HeadsOrTails = {heads, tails}
```



If we put a FairCoin and an UnfairCoin together, then what happens?

Progress: Tossing a Coin



```
Progress Check...
-- States: 3 Transitions: 4 Memory used: 58739K
Finding trace to cycle...
Depth 1 -- States: 1 Transitions: 2 Memory used: 59139K
Finding trace in cycle...
Depth 1 -- States: 1 Transitions: 1 Memory used: 59539K
Progress violation: Tails
Trace to terminal set of states:
        pick
Cycle in terminal set:
        toss
        heads
Actions in terminal set:
        {heads, toss}
Progress Check in: 1ms
```

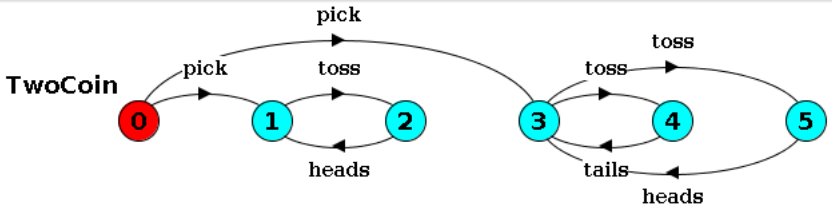
`pick` action leads to a set of states where `tails` action is not possible.

Graph Theory: Strong Connections

- Strong connections: *Two nodes m and n are said to be strongly connected, iff there is a path from m to n and vice-versa.*
- Strongly connected set: *A set of nodes in a graph is said to be a strongly connected set, iff every pair of nodes in the set is strongly connected.*

Terminal Sets

A terminal set is a strongly connected set from which there are no link out of the set.



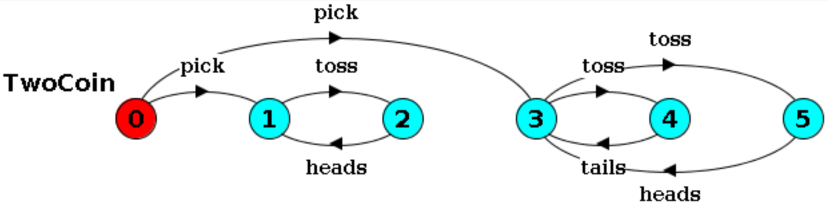
There are *three* strongly connected sets here.

1. $\{Q_0\}$: Single node, links out of it, not terminal. ✓
2. $\{Q_1, Q_2\}$ no links out, terminal. ✗
3. $\{Q_3, Q_4, Q_5\}$ no links out, terminal. ✗

The central concept for checking progress violation: identify terminal sets and check if at least one action from the progress set appears in these sets.

Terminal Sets

A terminal set is a strongly connected set from which there are no link out of the set.



There are *three* strongly connected sets here.

1. $\{Q_0\}$: Single node, links out of it, not terminal. ✓
2. $\{Q_1, Q_2\}$ no links out, terminal. ✗
3. $\{Q_3, Q_4, Q_5\}$ no links out, terminal. ✗

The `TwoCoin` process is clearly unfair to `tails` action.

Any questions?

Summary

Fairness and progress are important properties through which we can ensure liveness of a system. It is important to identify these and then design out such subtle issues.