L4 and A2 tips

L4-A2 Tip -2

Have sufficient command of Promela!

otherwise, you'll use the constructs wrong or in random ways without understanding what they do and how they work

Students often just randomly add/change the model or properties until all checks pass only to realize the checks fail on Vocareum or the model fails visual inspection for having violated the requirements

L4-A2 Tip -1

READ THE DESCRIPTION CAREFULLY – ALL OF IT!

Lots of DOs and DON'Ts in there!

There is a rubric with what deductions can apply if you violate the requirements!

L4-A2 Tip 0

Draw a process diagram first!

What are the different process instances?

How do they communicate?

Unidirectional channels?

Bidirectional channels?

Shared variables (e.g., that count balls)?

L4-A2 Tip 1

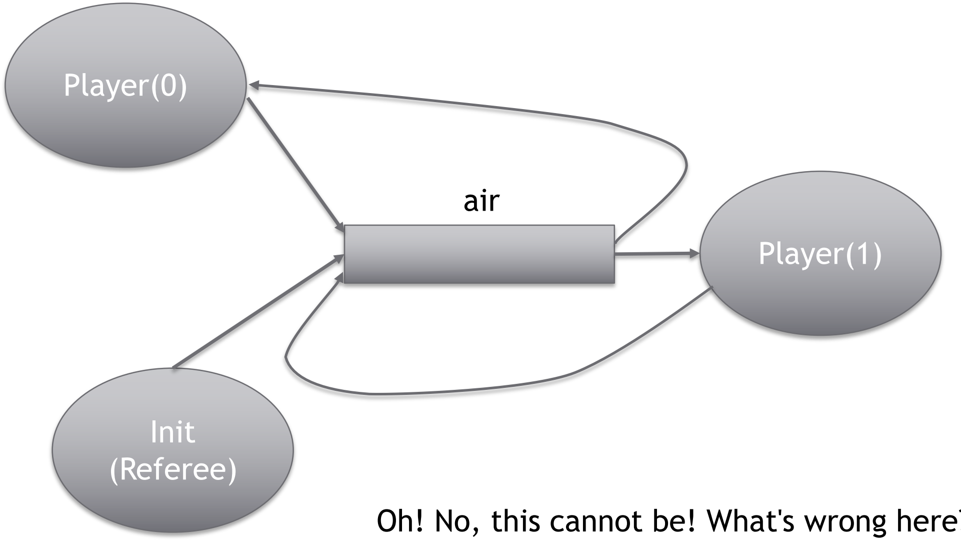
Stick to the physical metaphor of the “games”

DO NOT OVERTHINK!

When you throw the ball, it moves in one direction,
and cannot come back to the same player

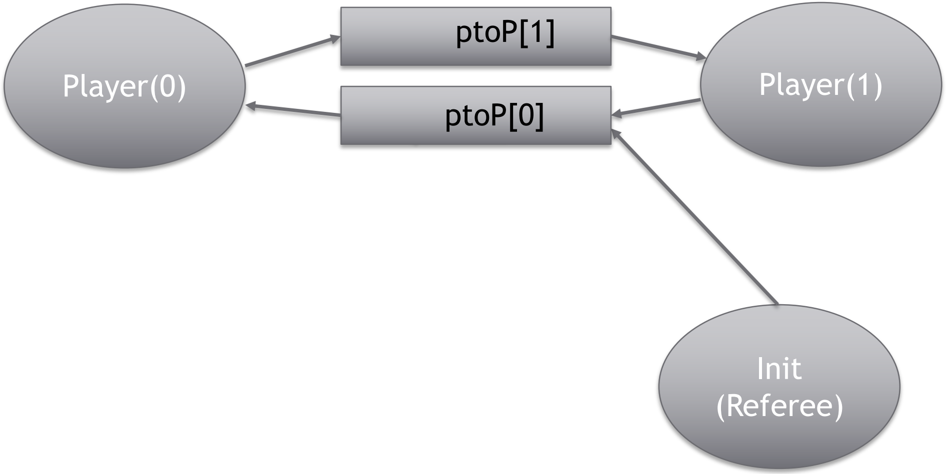
If you draw a process diagram, this will be obvious! Back to Tip 0!

Ugh!



Oh! No, this cannot be! What's wrong here?

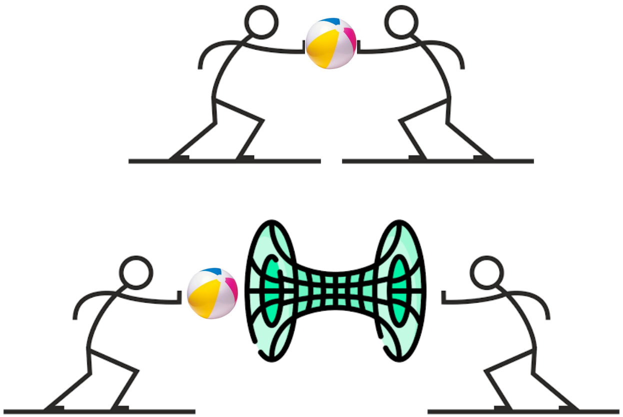
Better?



L4-A2 Tip 2.1

Channels cannot be synchronous (rendez-vous, capacity [0])

Then a player exchanges the ball instantly with the other – a ball is never in midair, violates the physical metaphor



No wormholes!
No transporters!
No stargates!

This is in the description!

Image: https://www.designboom.com/design/aerogel-twice-as-dense-as-air-and-the-lightest-material-in-the-world-06-04-2019/

L4-A2 Tip 2.2

Channels must have at least a capacity of 2

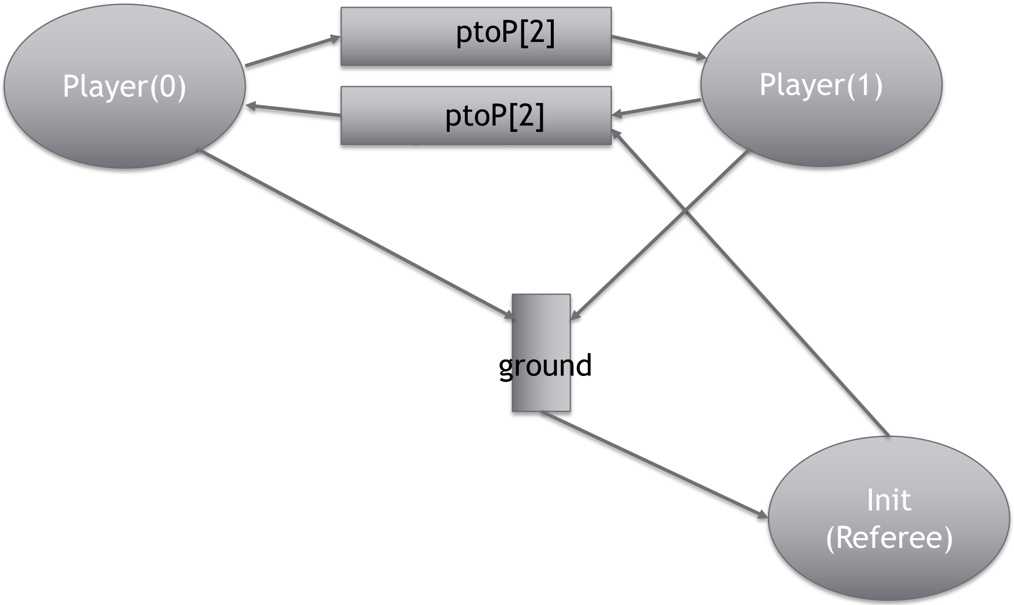
Otherwise, you can control the game by channel behavior alone, rather than processes

(when a channel is full, the air becomes solid in that direction, players will be forced to wait before they can throw another ball, but that's not a behavior that follows the physical metaphor)

This is in the description!

Image: https://www.designboom.com/design/aerogel-twice-as-dense-as-air-and-the-lightest-material-in-the-world-06-04-2019/

Better?



L4-A2 Tip 3

Don’t use observation variables as guards or to control behavior

This is a universal no-no:
observation variables are for properties only!

```
(allBalls > 0) -> toP0!BALL // not Ok
d_step { toP0?WHITE; balls[WHITE]++ } // Ok
```

Violation will result in a deduction!

len(ch) is treated as an observation variable => don’t use it as a guard

L4-A2 Tip 4

Don't abuse d_step or atomic

Again, these constructs can artificially restrict the behavior and violate the original intent of the model!

d_step { toP0?WHITE; balls[WHITE]++ } // Ok

atomic { toP0?WHITE; } // Useless!

d_step { toP0?WHITE; toP1!WHITE; } // No!

L4-A2 Tip 5

First forget about the properties, observation variables, and property interface

- Model the system as described, and **simulate** to see if it works
- Run a safety check to look for deadlocks
- If all looks good, only then add the observation variables, property interface, and properties, *making sure observation variables don't change the behavior*

L4-A2 Tip 6

Do not use <u>timeout</u> in Jugglers and JugglersBW!

It's not permitted in A2! You must find another way (and it's the simpler way)!

Can be used in L4 to allow the referee to detect a dropped ball, but it's not necessary!

L4-A2 Tip 7

Obey Promela Coding Style for readability

Avoid ";" after "}"

If we cannot read your model easily, we will waste too much time!

```
a[i]=a[i]+b[i-1]-1 // not ok in any programming language
a[i] = a[i] + b[i] – 1 // ok in all programming languages
```

L4-A2 Tip 8

You model can pass locally but fail on Vocareum
- Vocareum has extra checks and restricting options
- Unfortunately, feedback is not great when it fails (but this can be avoided by following the requirements and restrictions in the description -- back to Tip -2)

You model can pass locally and on Vocareum and still be incorrect
- You may have violated a requirement in a way that Vocareum cannot detect
- You may have created a model so restrictive or so liberal that it passes all checks trivially
- Can only tell by inspecting your model visually (that's why it needs to be readable -- back to Tip 7)

Promela/Spin ODDS and ENDS needed in A2

A common type of liveness property is non-starvation: the system will never indefinitely deny access to a resource (it will be fair to all users of resource)

```
bool accessed[2], reserved[2];
#define reserveResource(i) …
#define useResource(i) …

proctype Client(int i) { // in init: run Client(0), Client(1)
   accessed[i] = false; reserved[i] = false;
   do // repeatedly access resource
   ::  d_step { reserveResource(i); reserved[i] = true; }
       d_step { useResource(i); accessed[i] = true; }
       d_step { reserved[i] = false; accessed[i] = false; }
   od;
}

…
```

recurrent response pattern

```
ltl guaranteeAccess0 { [](reserved[0] -> <>accessed[0] }
ltl guaranteeAccess1 { [](reserved[1] -> <>accessed[1] }
…
```

Liveness properties (e.g., non-starvation) may rely on fairness assumptions

Weak form: If a statement becomes enabled and stays enabled, it will eventually be executed

– *if a process waits at a state in which in can always make progress, it eventually will*

Strong form: If a statement is infinitely often enabled (even if visiting other states in between), it will eventually be executed

– *if a process constantly revisits a state or stays in a state in which in can periodically make a different type of progress, it eventually will*

Spin can handle weak fairness, but not strong fairness

*We can force Spin to <u>assume</u> **weak fairness** when checking liveness/path properties,*

– Enable "Weak Fairness" in Liveness Check

– Required in some A2 properties

but if we need strong fairness, we must bake strong fairness into the model ourselves (we have to implement it explicitly with some kind of fairness algorithm in the model)

Weak fairness may guarantee certain types of progress, preventing starvation

- **Weak fairness**: If a transition stays enabled indefinitely in the global system, it will eventually be executed

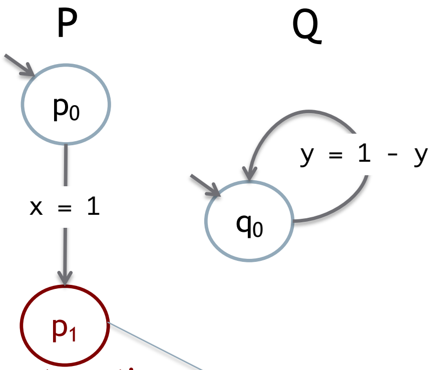
- Turning weak fairness on in *Liveness Verification* says the execution environment guarantees this assumption

(e.g., when you are modeling a problem in which the actual execution environment never forever ignores processes idle-waiting on a forever-enabled move)

A fairness assumption is an assumption about the underlying execution environment

Initially: `x = 0`

P's goal: set x to 1



non-starvation state
(x == 1)

assume: once P reaches here, it hasn't starved

Imagine two different Operating Systems executing these processes concurrently on a single core:

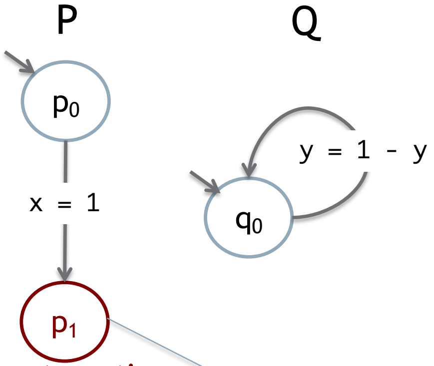
- OS1: unfair, randomly allocates CPU cycles to enabled transitions (no awareness or memory of who used the CPU and how frequently)

- OS1: P <u>can</u> starve: Q continually gets turns, P is stuck at its initial state

A fairness assumption is an assumption about the execution environment

Initially: `x = 0`

P's goal: set x to 1



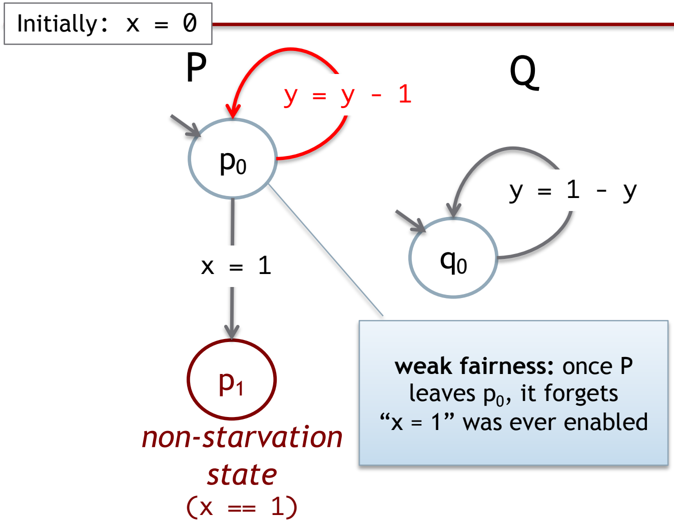
non-starvation state
(x == 1)

assume: once P reaches here, it hasn't starved

- OS1: unfair, randomly allocates CPU cycles to enabled transitions (no memory)
- OS2: fair, tries to equitably allocate CPU cycles to enabled transitions (has awareness of who used the CPU and how frequently)

- OS1: P can starve: Q continually gets turns, P is stuck at its initial state
- OS2: P can not starve: Q may get several turns, but eventually P gets its turn

Fairness assumptions can be strong or weak

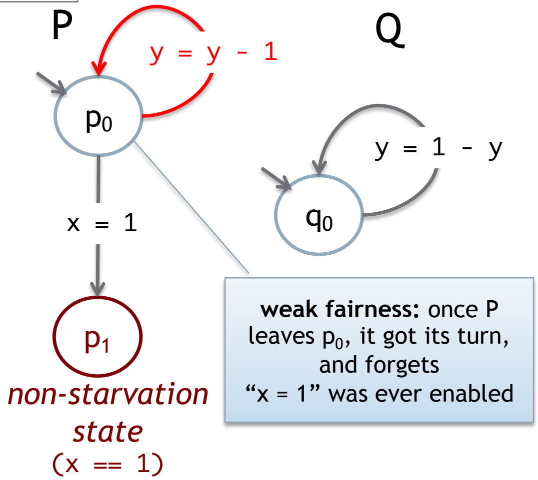


- OS1: unfair, randomly allocates CPU cycles to enabled transitions (no awareness)

- OS1: P can starve: Q continually gets turns or P continually recycles at its initial state, or they alternate

Fairness assumptions can be strong or weak

Initially: `x = 0`



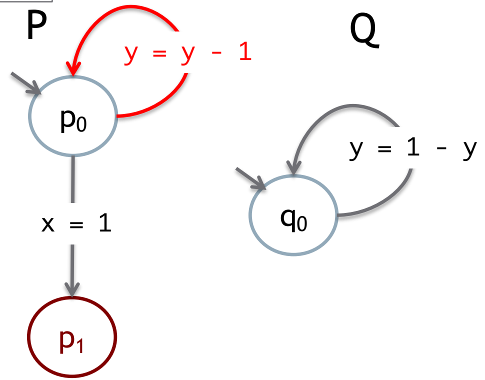
- OS1: unfair, randomly allocates CPU cycles to enabled transitions (no memory)
- OS2.1: <u>weakly</u> fair, makes sure to allocate CPU cycles to a transition only if it notices that the transition has been waiting with no other option (has temporary local awareness)

  Doesn't have memory. Once p has moved, the OS doesn't remember, might self-cycle.

- OS1: P <u>can</u> starve: Q continually gets turns or P continually recycles at its initial state, or they alternate
- OS2.1: P <u>can</u> starve if it has red self-cycle: P continually recycles at its initial state (it starves itself)
  P cannot starve if it doesn't have the red self-cycle, as OS2.1 will detect the waiting state

Fairness assumptions can be strong or weak

Initially: x = 0



non-starvation state
(x == 1)

- OS1: unfair, randomly allocates CPU cycles to enabled transitions (no memory)
- OS2.1: weakly fair, makes sure to allocate CPU cycles to a transition only if it notices that the transition has been waiting with no other option (has temporary local memory)
- OS2.2: strongly fair, tries to equitably allocates CPU cycles to enabled transitions (has full awareness)

This 3rd type of OS has sophisticated memory and aware that p is not taking the transition to achieve its goal.

- OS1: P can starve: Q continually gets turns or P continually recycles at its initial state, or they alternate
- OS2.1: P can starve: P continually recycles at its initial state (it starves itself)
  P cannot starve if it doesn't have the red self-cycle, as OS2 will detect the waiting state
- OS2.2: P can not starve: P may recycle at its initial state for a while, but OS2.2 will eventually recognize that "x = 1" has continuously been ignored, and will favor it

Fail Properties

- Have you noticed properties like this failing in Vocareum?

`REF_notPossiblyAllBallsAreDropped_FAIL`

- Any idea what this means?

It is explained in the description, but let's go over it!

Fail Properties

`REF_notPossiblyAllBallsAreDropped_FAIL`

- The property `notPossiblyAllBallsAreDropped` should fail for the verification to pass (failure is the expected/desired behavior). If it passes, you are in trouble.

Negative verification

LTL means: for all paths, some property is true

If

`ltl someProperty = { <some_random_property> }`

then this means

`<some_random_property>` must be true <u>for all</u> execution traces/paths!

LTL is implicitly universally quantified!

How do I express an existential property?

Suppose I want to check that something weaker...

for <u>some path,</u> <some_random_property> is true!

What do I do?

Use First-Order Logic rule for how negation distributes over universal and existential quantifiers!

P(x): P is true for path x

(ForSome x)(P(x)) == !(ForAll x)(!P(x))

Invert property P(x) and expect the verification to fail!

The failure trace will be the proof that P(x) holds on some path x!

Consider a system with just two paths x1 and x2

P(x): P is true for path x

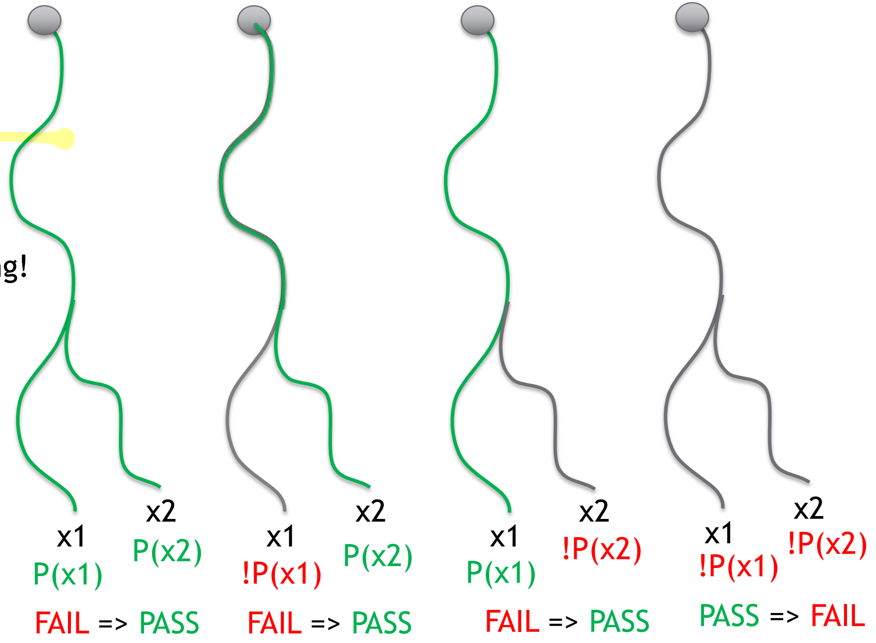
(ForSome x)(P(x)) == !(ForAll x)(!P(x))

![]

What we are actually checking!

What we want to check

For some x ballsOnGround == 2 == !(for all x) (ballson Ground != 2)



Going back to FAIL properties

REF_notPossiblyAllBallsAreDropped_FAIL

- Verifies that for some path, the property PossiblyAllBallsAreDropped holds

- Because if the verification fails, then there is a path in which notPossiblyAllBallsAreDropped is false….

- Which means PossiblyAllBallsAreDropped must be true for that path!

Appendix

Some more subtle points and further examples

Source: spin-public/Mutex

Dekker.pml: A mutex algorithm (Dekker's algorithm [1962])

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1; // I want access
      turn = B_TURN; // B's turn after I'm done
      (y == 0 || (turn == A_TURN)) ->
      mutex++; // in critical section
      mutex--; // out of critical section
      x = 0; // I'm done
}

active proctype B() {
      y = 1; // I want access
      turn = A_TURN; // A's turn after I'm done
      (x == 0 || (turn == B_TURN)) ->
      mutex++; // in critical section
      mutex--; // out of critical section
      y = 0; // I'm done
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Run *Safety Verification*
  - *Assertion violations*
- Does it pass?

This is a fix for the progress problem we had earlier with the initial Mutex example!

[Dekker, 1962] First software-only solution to the mutex problem (for two processes)

Source: spin-public/Mutex

Dekker2.pml

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
// mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1;
      // turn = B_TURN;
      y == 0 // || (turn == A_TURN) ->
      mutex++; // in critical section
      mutex--;
      x = 0;
}

active proctype B() {
      y = 1;
      // turn = A_TURN;
      x == 0 // || (turn == B_TURN) ->
      mutex++; // in critical section
      mutex--;
      y = 0;
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Remove turn variable
- Run *Safety Verification*
  - *Assertion violations*
- Does it pass? Why?

Source: spin-public/Mutex

Dekker2.pml

```
mtype = { A_TURN, B_TURN }
bit x, y;   // signal entering/leaving the section
byte mutex; // # of procs in the critical section
// mtype turn;  // who's turn is it next?

active proctype A() {
      x = 1;
      // turn = B_TURN;
      y == 0 // || (turn == A_TURN) ->
      mutex++; // in critical section
      mutex--;
      x = 0;
}

active proctype B() {
      y = 1;
      // turn = A_TURN;
      x == 0 // || (turn == B_TURN) ->
      mutex++; // in critical section
      mutex--;
      y = 0;
}

active proctype mutex_p() {
      assert(mutex < 2);
}
```

- Remove turn variable
- Run *Safety Verification*
  - *Assertion violations*
- Does it pass? NO!
- This becomes the same mutex algorithm we tried earlier in course (but instead of a busy-wait, we have a simple guard, so we should get a deadlock instead of a livelock)
- Run *Guided Simulation* with failure trace to check your hunch

Liveness properties -- Progress: the system will move to a productive state

```
ltl willProgress { <>(mutex > 0) }
```

```
bit x, y;
byte mutex = 0;

active proctype A() {
    x = 1;
    (y == 0) ->
    mutex++;
// A in critical section
    mutex--;
    x = 0;
}
```

```
active proctype B() {
    y = 1;
    (x == 0) ->
    mutex++;
// B in critical section
    mutex--;
    y = 0;
}
```

Liveness properties -- Progress: the system will move to a productive state

```
ltl willProgress { <>(mutex > 0) }
```

```
bit x, y;
byte mutex = 0;

active proctype A() {
    x = 1;
    y == 0;
    mutex++;
// A in critical section
    mutex--;
    x = 0;
}
```

```
active proctype B() {
    y = 1;
    x == 0;
    mutex++;
// B in critical section
    mutex--;
    y = 0;
}
```

-> has different meanings in LTL and plain Promela

```
ltl willProgress { [](a -> b) }
```

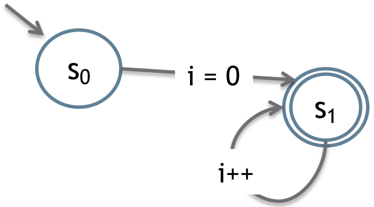
Logical implication

```
bool a;
Int b;

proctype Dummy() {
    …
    a -> // this is a guard
    b++;
    …
}
```

Just statement separator: -> == ;

```
bool a;
Int b;

proctype Dummy() {
    …
    a; // this is a guard
    b++;
    …
}
```

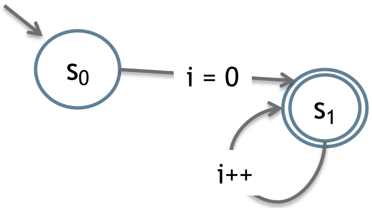
What program code could this FSA represent



i is one-byte unsigned integer

What is the complete state space of this FSA

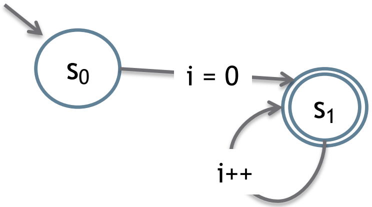
```
active proctype forever() {
    byte i;
    i = 0;
s1:
    do
    :: i++
    od
}
```



i is one byte unsigned integer

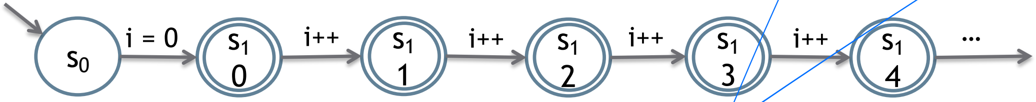
How many global states?
(in Spin)

What is the complete state space of this FSA



local state

i is one byte unsigned integer



global state

257 global states?

Local state is different from global state

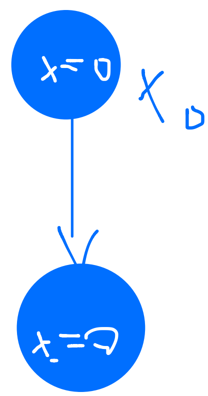
Process state as defined in a model is local: represents execution/process counter, or node-label of underlying FSA

System state is global and comprises:

- identities of all active processes
- local state of all active processes
- values of all local variables of all active processes
- contents of all channels
- values of all global variables

Livness properties - Termination: the system will eventually stop at a valid end state

```
bool done = false;

ltl stops { <>(done) }

active proctype Q() {
    int i = 0;
    do
    :: i < MAX -> i++;
    :: i < MAX -> break;
    :: i == MAX -> break;
    od
    done = true;
}
```


x=0
X 0