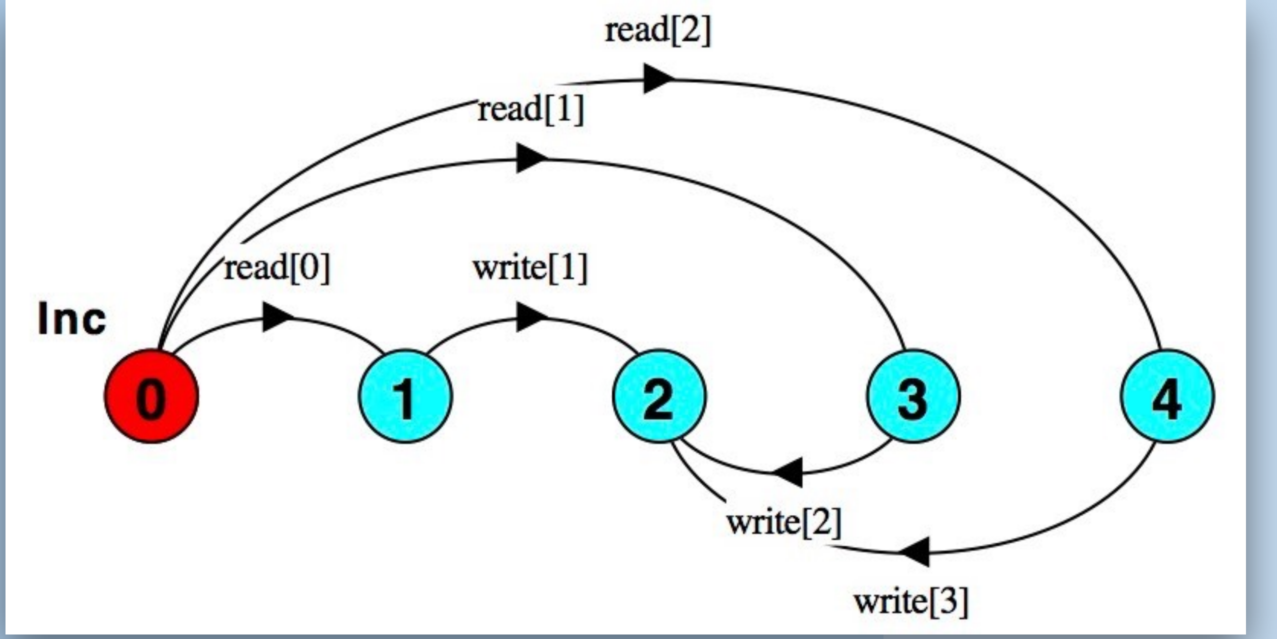
3. Safety and Synchronization

Oscar Nierstrasz



Selected material © Magee and Kramer

Roadmap

- Modelling interaction in FSP
- Safety — synchronizing critical sections
- Locking for atomicity
- The busy-wait mutual exclusion protocol
- Checking Safety properties
- Conditional synchronization

Roadmap

> **Modelling interaction in FSP**
> Safety — synchronizing critical sections
> Locking for atomicity
> The busy-wait mutual exclusion protocol
> Checking Safety properties
> Conditional synchronization

Modelling interaction — shared actions

Actions that are common between two processes are shared and can be used to model *process interaction*:

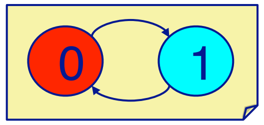
> *Unshared actions* may be *arbitrarily interleaved*

> *Shared actions* occur *simultaneously* for all participants

```
MAKER   = ( make -> ready -> MAKER ).
USER    = ( ready -> use -> USER ).

||MAKER_USER = ( MAKER || USER ).
```

What are the states of the LTS?
The traces?



1-maker_user.lts

Up to now we only considered concurrent composition of two *independent* processes. Communication and synchronization are modeled in FSP as *shared actions*. Note that no “direction” is assumed in these shared actions — it is purely a matter for interpretation in terms of what you are modeling.

In this example, we model a (coffee?) maker and a user. The `MAKER` makes something, and then declares that it is `ready`. The `USER` waits for the `MAKER` to be ready and then uses what it produce. The only shared action is `ready`.

If we compose these two processes as before, we obtain potentially 2x2 = 4 states. However, since both processes must take the `ready` action together, the number of possible independent actions is reduced.

How many states and transitions are there in the composition?

(Work it out on the blackboard and then test it with LTSA.)

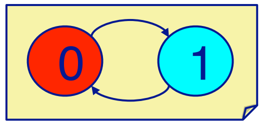
Modelling interaction — handshake

A handshake is an action that signals acknowledgement

```
MAKERv2= ( make -> ready -> used -> MAKERv2 ).
USERv2   = ( ready -> use -> used -> USERv2 ).

||MAKER_USERv2 = ( MAKERv2 || USERv2 ).
```

What are the states and traces of the LTS?



2-maker_userv2.lts

In this example, not only does the MAKER declare when it is ready, but the USER declares when it has consumed what has been produced. There are therefore two shared actions that must be synchronized.

What are the states and traces of the composed process system?

As before, work it out on the blackboard, and verify with LTSA.

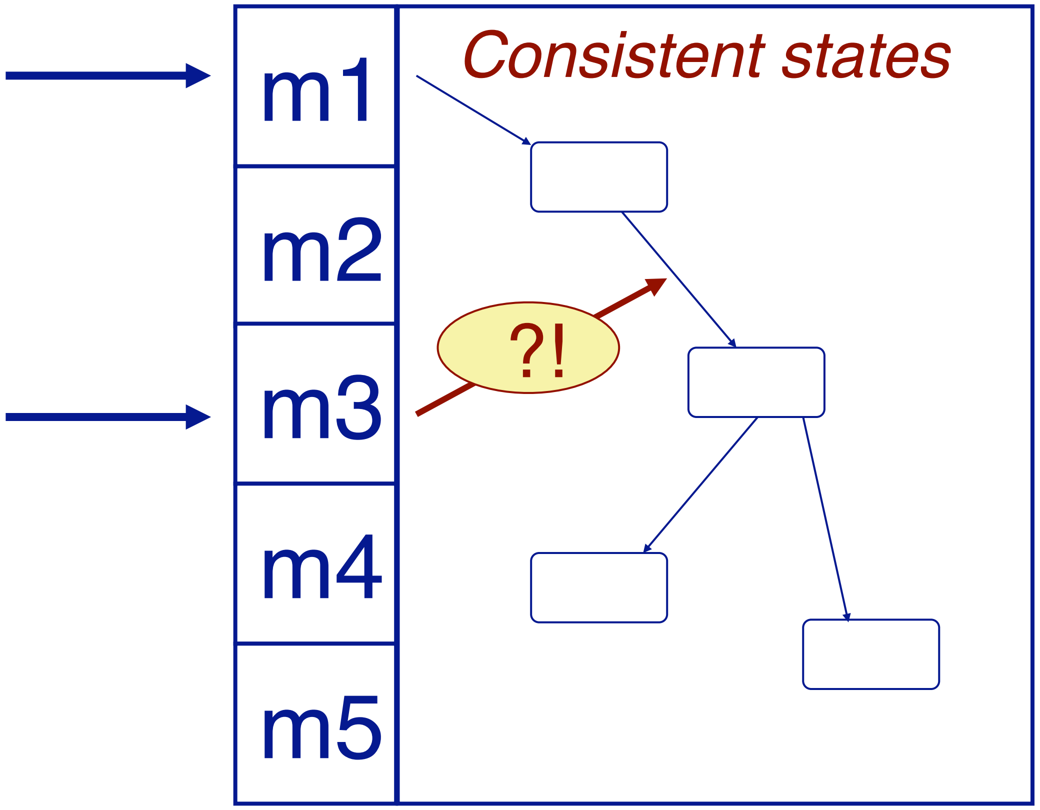
Roadmap

- Modelling interaction in FSP
- **Safety — synchronizing critical sections**
- Locking for atomicity
- The busy-wait mutual exclusion protocol
- Checking Safety properties
- Conditional synchronization

Safety problems

Objects must only be accessed when they are in a *consistent state*, formalized by a class invariant.

Each method *assumes* the class invariant holds when it starts, and it *re-establishes* it when done.



If methods interleave arbitrarily, an inconsistent state may be accessed, and the object may be left in a “dirty” state.

Wherever shared resources are updated may be a critical section.

Recap: in sequential programming, we can formalize the valid states of an object using a class invariant. The class invariant must hold true before and after any public method. We can reason about the correctness of methods by relying on the class invariant always being true at the start of any public method.

During the execution of a method, the state may be updated, and so the class invariant may temporarily be invalid.

If one thread is executing method m1 and a second thread starts executing method m3, the object may not be in a consistent state. As a consequence we no longer can guarantee that method m3 will execute correctly. The entire object may then be left in a corrupt state, and anything can happen.

Race conditions

A race condition exists if safety may be violated by bad timing

```
public class AccountBAD extends Account {
 // unsynchronized!
 public void withdraw(int amount) {
  while (amount > assets) {
   Thread.yield();    // busy wait
  }
  Thread.yield();     // race condition!
  assets -= amount;
  checkInvariant();    // might fail!
 }
}
```

Account

```
if (!(assets >= 0)) { errors++; }
```

A *race condition* exists on a program if safety may be violated just because of unfortunate timing.

In this example, the bad `Account` class *busy waits* until there are enough assets available to satisfy the withdrawal request. However it then performs another `yield`, during which another thread may perform conflicting withdrawal. In the end, the invariant that the `assets` never fall below zero may be violated.

Look at the `AccountTest` class to see how to provoke the race condition. The actual race is whether the invariant still holds or not. The test case counts the number of violations. By reducing the number of deposits, the race condition becomes less likely.

See also: https://en.wikipedia.org/wiki/Race_condition

Interference

Consider these two processes:

```
           { x = 0 }
AInc: x := x+1
BInc: x := x+1
           { x = ? }
```

How can these processes interfere?

Consider all the ways in which these two processes could possibly interfere.

Be careful to explicitly state your assumption about what actions are guaranteed to be *atomic* in this (unnamed) language.

Atomic actions

Individual reads and writes may be atomic actions:

```
const N   = 3
range T   = 0..N

Var           = Var[0],
Var[u:T]      = ( read[u]      -> Var[u]
                | write[v:T]   -> Var[v] ).

set VarAlpha = { read[T], write[T] }
Inc           = (  read[v:0..N-1]
                   ->write[v+1]
                   ->STOP    )          +VarAlpha.
```

Let us model the previous example as finite state processes. We will model the variable `x` as a process `Var`, and the incrementing processes `AInc` and `BInc` as two instances of the process `Inc`.

The `Var` process can hold the values 0 to 3 (as a macro parameter). It participates in the actions `read[]` and `write[]`. As before, no direction is assumed, but we can clearly see that only a `read` of the currently stored value `u` is possible, while `write` causes the stored value to change to the written value `v`.

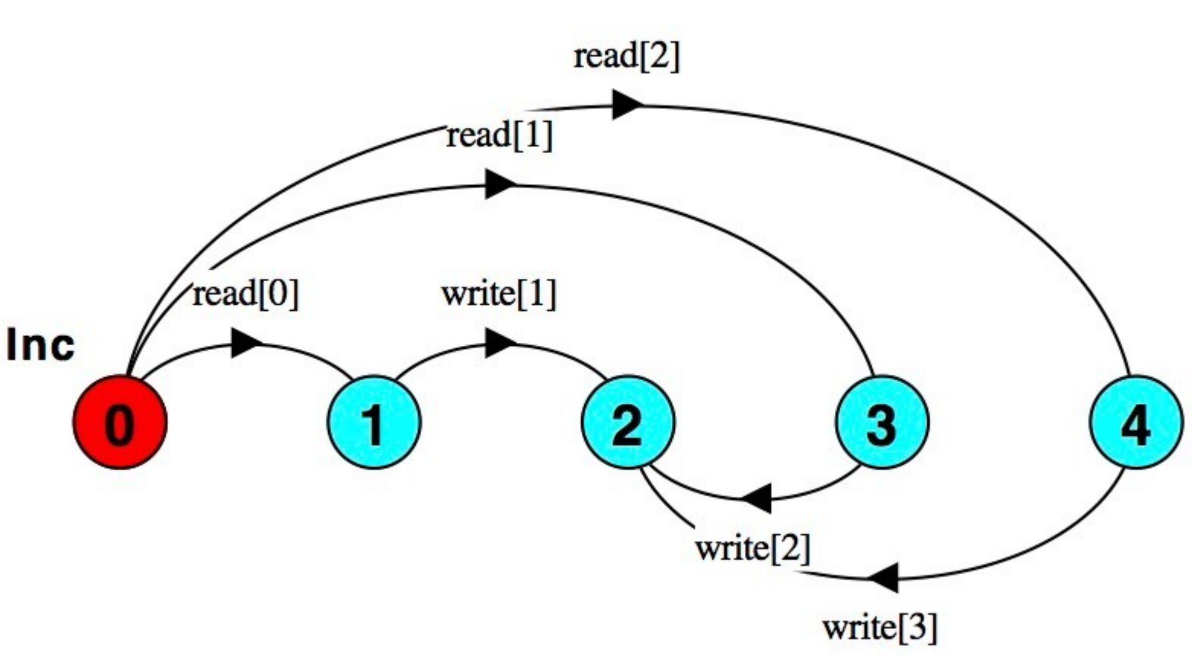
An `Inc` process reads the variable, increments it, and then writes the incremented value.

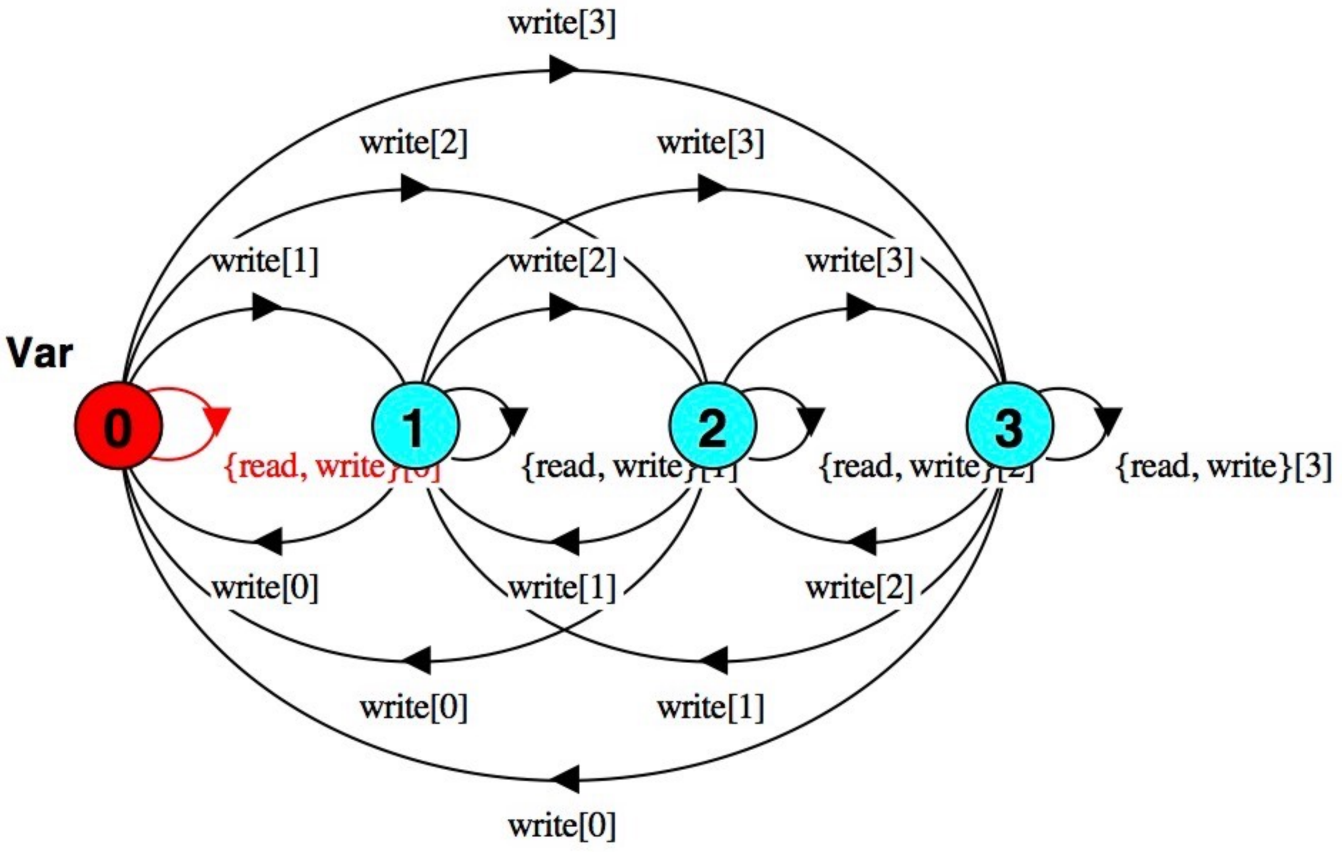
Note that `Inc` cannot perform `write[0]`, nor can it `read[3]`. We must explicitly add these to `Inc` (`+ VarAlpha`) else LTSA will allow `Var` to independently take this action!

(Remove the annotation +VarAlpha and see what it generates ...)

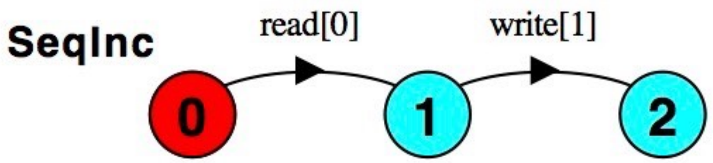
Sequential behaviour

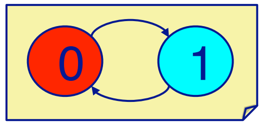
A single sequential thread requires no synchronization:





```
||SeqInc = (Var||Inc).
```

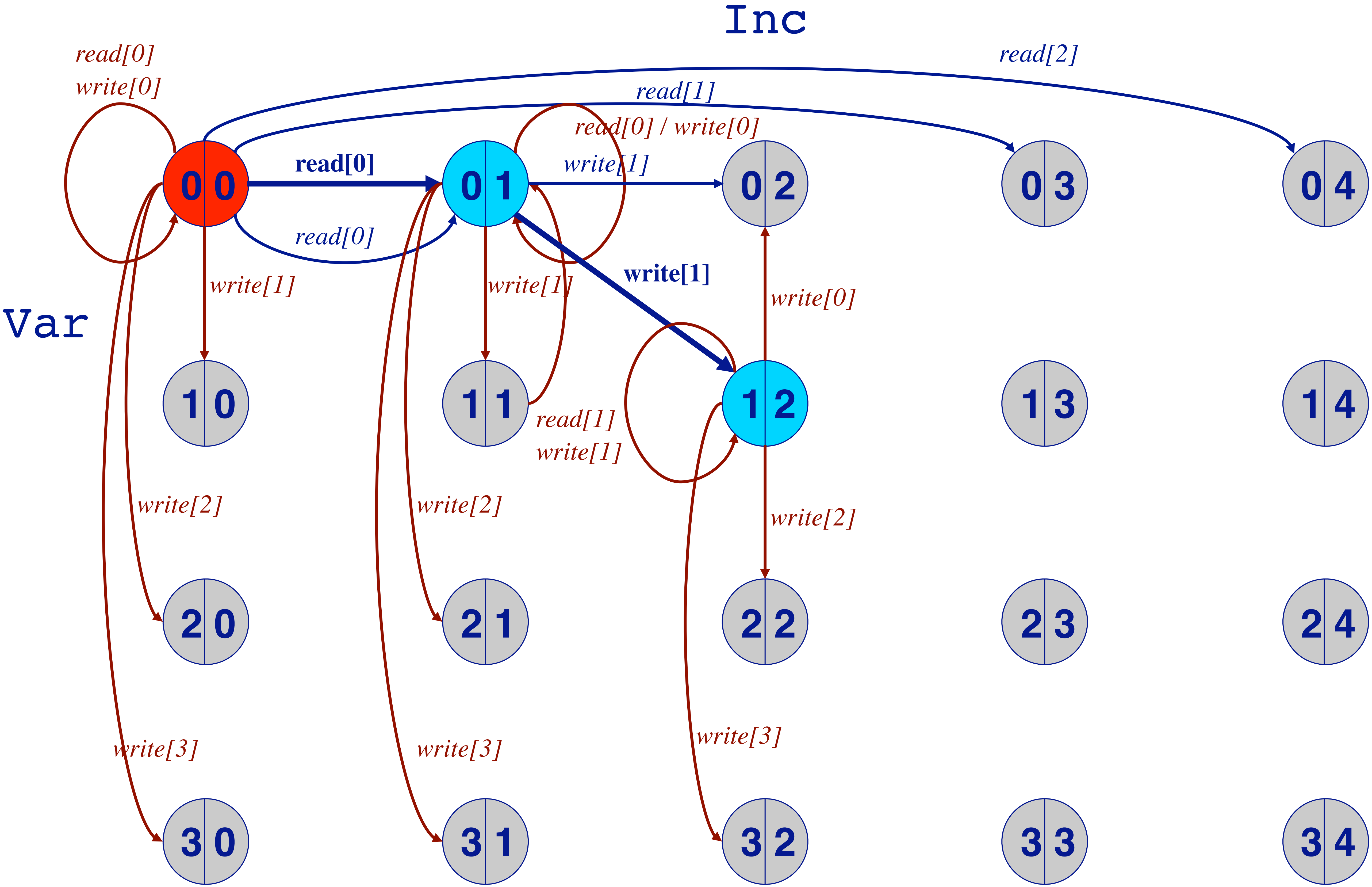




3-inc-conflict.lts

Here we model a single increment of the variable. Although the two processes are quite complex, their parallel composition is trivial.

Try this without `+VarAlpha` — `SeqInc` will then allow a `write[0]` at any time!


Inc
Var
read[0]
write[0]
read[2]
read[1]
read[0] / write[0]
read[0]
write[1]
read[0]
write[1]
write[1]
write[1]
write[0]
read[1]
write[1]
write[2]
write[2]
write[2]
write[3]
write[3]
write[3]
0 0
0 1
0 2
0 3
0 4
1 0
1 1
1 2
1 3
1 4
2 0
2 1
2 2
2 3
2 4
3 0
3 1
3 2
3 3
3 4

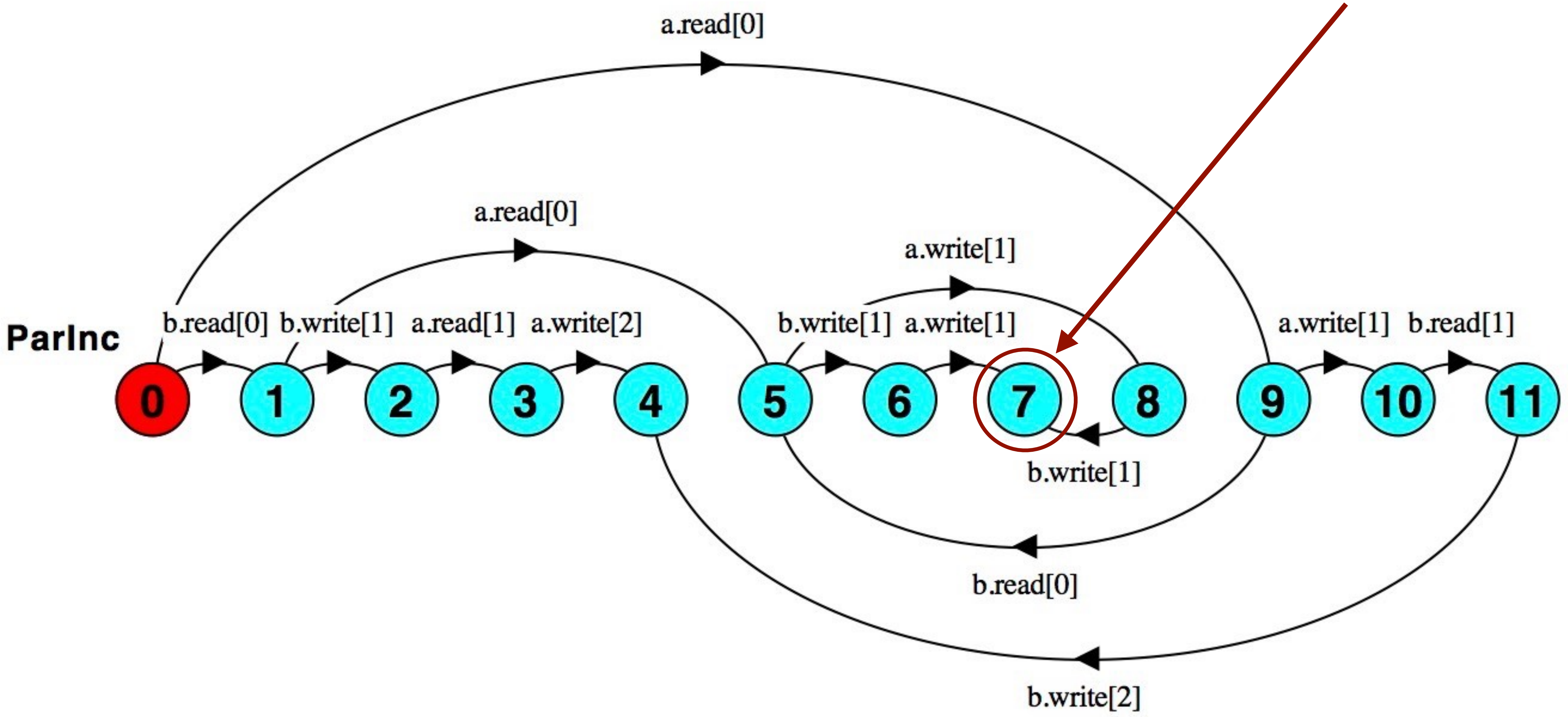
The combined state space contains 4×5 = 20 states, but only 3 of these are reachable due to synchronization between the processes. Both processes must agree on each combined transition, which eliminates most of the states.

Roadmap

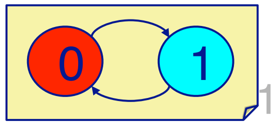
- Modelling interaction in FSP
- Safety — synchronizing critical sections
- **Locking for atomicity**
- The busy-wait mutual exclusion protocol
- Checking Safety properties
- Conditional synchronization

Concurrent behaviour

Without synchronization, concurrent threads may <u>interfere</u>:



```
||ParInc = ({a,b}::Var || a:Inc || b:Inc).
```



3-inc-conflict.lts

Here we compose two instances of `Inc` with a common shared variable. Note how we prefix the two instances of `Inc` with "`a`" and "`b`" to ensure that they are independent. We prefix `Var` with *both* names, so that the variable can be shared between them. This also allows us to distinguish a read by process `a` from one by process `b`.

In the LTS diagram we can see two terminal states: (4) represents the case where either `a` completes its incrementation before `b` or vice versa, resulting in the variable being assigned a value of 2; (7) represents the state reached when they overlap, resulting in a value of 1.

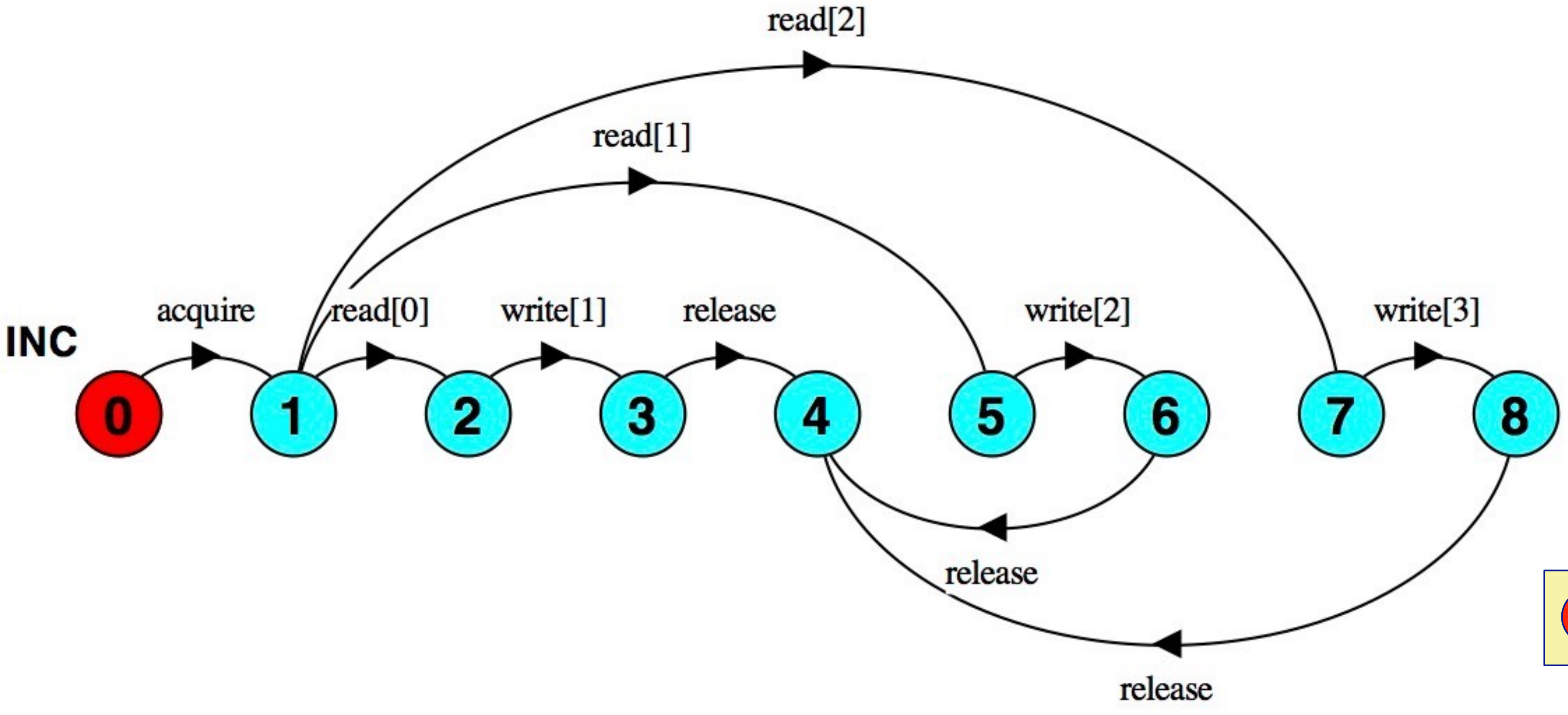
This is a clear example of a *race condition*, since obtaining a correct result or not depends on timing.

(Be sure not to ask LTSA to minimize the result, or the two different final states will be combined!)

Locking

Locks are used to make a critical section atomic:

```
LOCK = (  acquire -> release -> LOCK ).
INC  = (  acquire
          -> read[v:0..N-1]
          -> write[v+1]
          -> release
          -> STOP )                +VarAlpha.
```



4-lock.lts

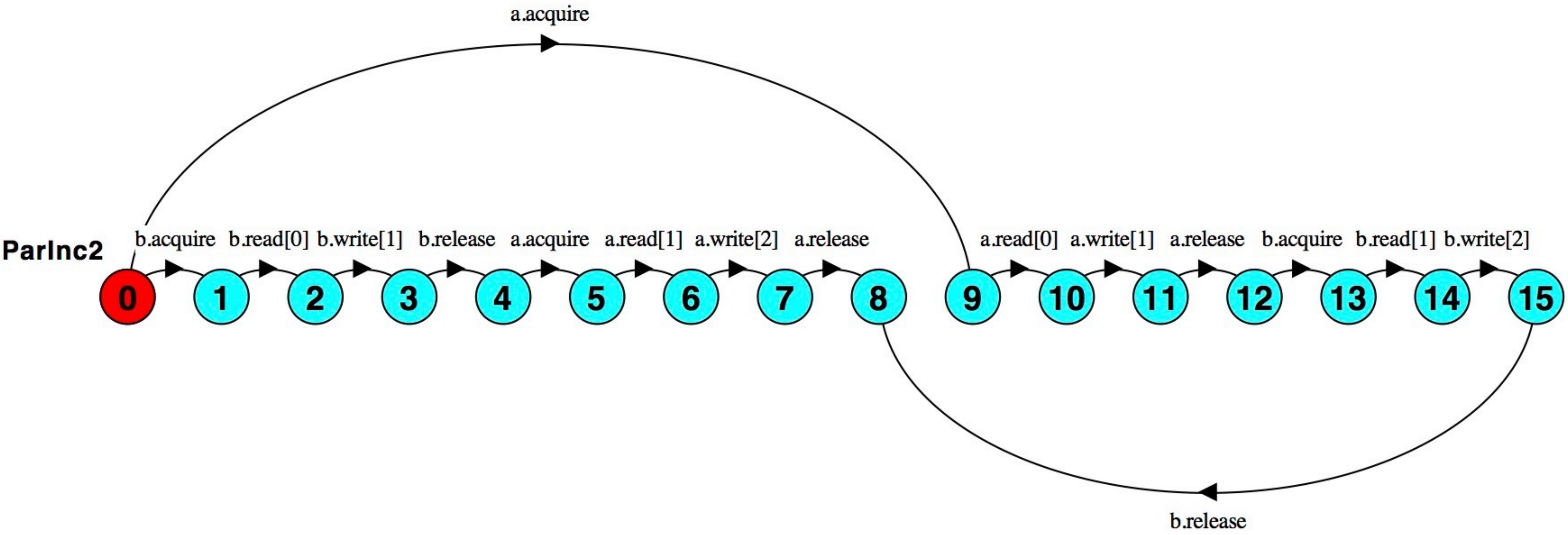
In this example we model a `LOCK` as a process that can be `acquired` and then `released`.

Since reading and writing the variable constitutes a *critical section*, we protect these actions with the help of the lock. This effectively makes the critical section *atomic*, thus eliminating the race condition.

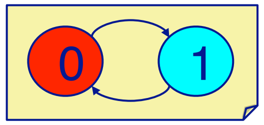
Note that this only works if all participants obey the locking protocol. (This is guaranteed with monitors since they encapsulate both operations on shared data and the locking protocols.)

Synchronization

Processes can synchronize critical sections by sharing a lock:



```
||ParInc2 = ({a,b}::VAR || {a,b}::LOCK || a:INC || b:INC).
```



4-lock.lts

Here we guarantee that a and b cannot both be in their critical section at the same time, and thus eliminate the race condition.

What are the possible traces in this solution?

Synchronization in Java

Java Threads also synchronize using locks:

```
synchronized T m() {
  // method body
}
```

is just *convenient syntax* for:

```
T m() {
  synchronized (this) {
    // method body
  }
}
```

Every object has a lock, and Threads may use them to synchronize with each other.

Roadmap

> Modelling interaction in FSP
> Safety — synchronizing critical sections
> Locking for atomicity
> **The busy-wait mutual exclusion protocol**
> Checking Safety properties
> Conditional synchronization

Busy-Wait Mutual Exclusion Protocol

P1 sets `enter1 := true` when it wants to enter its CS, but sets `turn := "P2"` to yield priority to P2

```
process P1
   loop
      enter1 := true
      turn := "P2"
      while enter2 and
            turn = "P2"
         do skip
      Critical Section
      enter1 := false
      Non-critical Section
   end
end
```

```
process P2
   loop
      enter2 := true
      turn := "P1"
      while enter1 and
            turn = "P1"
         do skip
      Critical Section
      enter2 := false
      Non-critical Section
   end
end
```

Is this protocol correct? Is it fair? Deadlock-free?

Mutual exclusion: When P1 enters its CS, we know that enter1 = true and that either (1) enter2 = false, or (2) turn = “P1”.

1. If enter2 = false, then P2 has entered its non-CS, will set enter2 = true and turn = “P1”. Since enter1 = true, it will then busy-wait till P1 leaves its CS.
2. Else, if turn = “P1”, then we know that P2 set turn = “P1” *after* P1 set enter1 = true and turn = “P2”. So P2 must be busy-waiting!

Fairness: If P1 is busy-waiting, then enter1 = true, enter2 = true and turn = “P2”. Eventually P2 will go around the loop and set turn = “P1”, letting P1 proceed and forcing P2 to busy-wait. Similarly, if P2 wants to get into its CS, P1 will eventually let it do so, as long as its CS and non-CS eventually terminate.

Atomic read and write

We can model integer and boolean variables as processes with atomic read and write actions:

We will use these to model the variables enter1, enter2 and turn.

```
range T = 1..2

Var = Var[1],
Var[u:T] =
   ( read[u]          -> Var[u]
   | write[v:T]       -> Var[v]).

set Bool = {true,false}

BOOL(Init='false)   = BOOL[Init],
BOOL[b:Bool] =
   ( is[b]            -> BOOL[b]
   | setTo[x:Bool]    -> BOOL[x]).
```

Modelling the busy-wait protocol

Each process performs two actions in its critical section:

```
P1 = ( enter1.setTo['true]
   -> turn.write[2]
   -> Gd1),
Gd1 =
   ( enter2.is['false] -> CS1
   | enter2.is['true] ->
      ( turn.read[1] -> CS1
      | turn.read[2] -> Gd1)),
CS1 = ( a -> b
   -> enter1.setTo['false]
   -> P1).
```

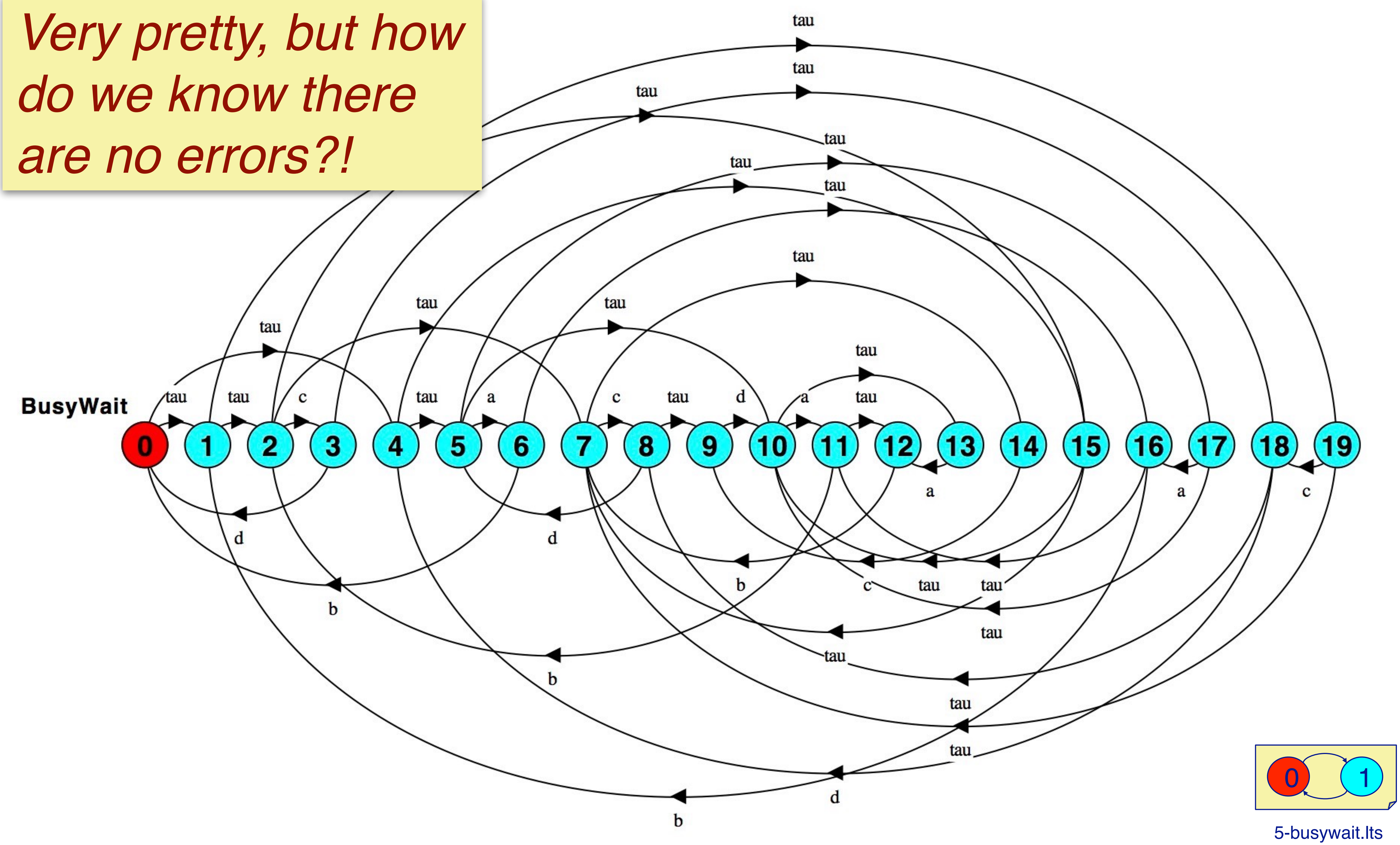
```
P2 = ( enter2.setTo['true]
   -> turn.write[1]
   -> Gd2),
Gd2 =
   ( enter1.is['false] -> CS2
   | enter1.is['true] ->
      ( turn.read[2] -> CS2
      | turn.read[1] -> Gd2)),
CS2 = ( c -> d
   -> enter2.setTo['false]
   -> P2).
```

```
||BusyWait = (enter1:BOOL||enter2:BOOL||turn:Var||P1||P2)@{a,b,c,d}.
```

Since we are only interested in the actions performed in the critical sections (a, b, c, d), we use the interface operator to make only those actions visible. All other, internal actions will be translated to “tau”.

Busy-wait composition

Very pretty, but how do we know there are no errors?!



5-busywait.lts

We have exhaustively modeled all possible transitions of our busy-waiting protocol. Note how LTSA translates all internal actions to tau.

The problem with this analysis is that we have not (yet) expressed which states or traces correspond to safety violations.

Model checking

- LTSA is an example of a *model checker*
  - Express an abstract, finite-state model of a system
  - Exhaustively check all possible state transitions
  - Ensure that certain properties are not violated
    - *E.g., no safety violations, no deadlock or other liveness violations.*

There are many different kinds of model checkers. LTSA is one that is dedicated to modeling and analyzing concurrent systems as finite state processes.

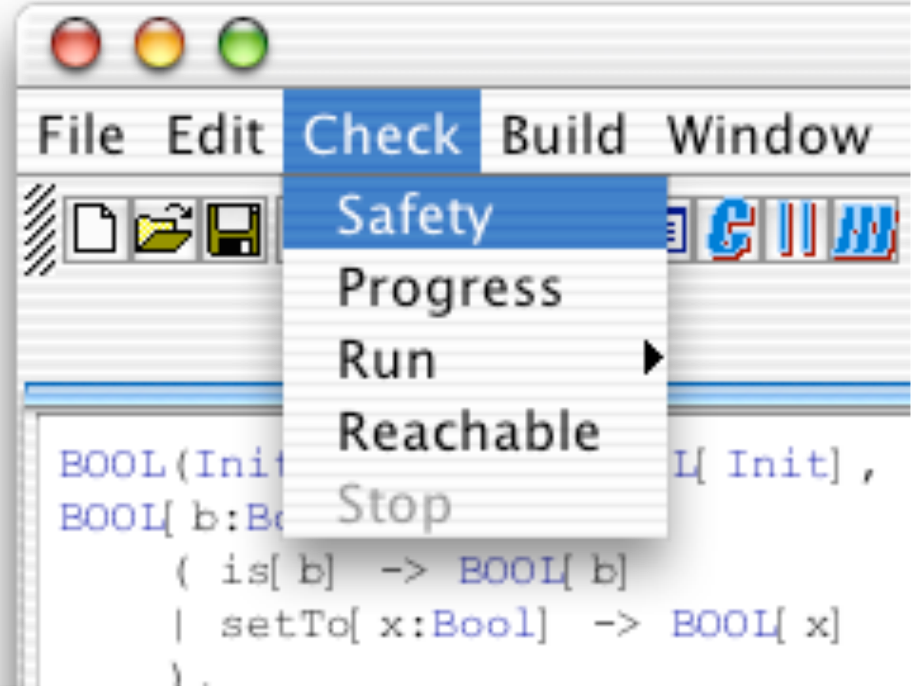
See also: https://en.wikipedia.org/wiki/Model_checking

Checking for errors

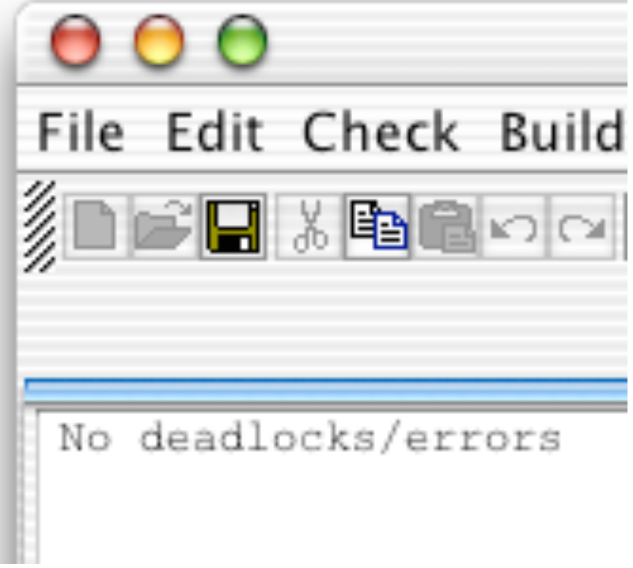
We can check for errors by composing our system with an agent that moves to the `ERROR` state if atomicity is violated:

```
Ok =  ( a -> ( c -> ERROR | b -> Ok )
      | c -> ( a -> ERROR | d -> Ok)).

||BusyWaitOK = (enter1:BOOL||enter2:BOOL||turn:Var||P1||P2||Ok).
```

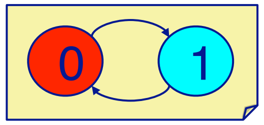


```
BOOL(Init                     I[ Init] ,
BOOL[ b:B
      ( is[ b]  -> BOOL[ b]
      | setTo[ x:Bool]  -> BOOL[ x]
      ).
```



```
No deadlocks/errors
```

What happens if we break the protocol?



5-busywait.lts
6-brokenbusywait.lts

In order for LTSA to check for safety errors, we have to tell it explicitly which are the erroneous states that must be avoided.

In the busy-wait example, we want to avoid that two processes may be in their critical sections at the same time. This would be the case if we observe the sequence of actions `ac` (i.e., `P2` starts its critical section while `P1` has not yet completed its own). The process `Ok` expresses exactly the traces that should lead to an `ERROR` state, as well as the ones that are safe.

(In a few slides we will see a better way to express this by turning `Ok` into a "*property*".)

Change one of the guards to see how the protocol breaks!

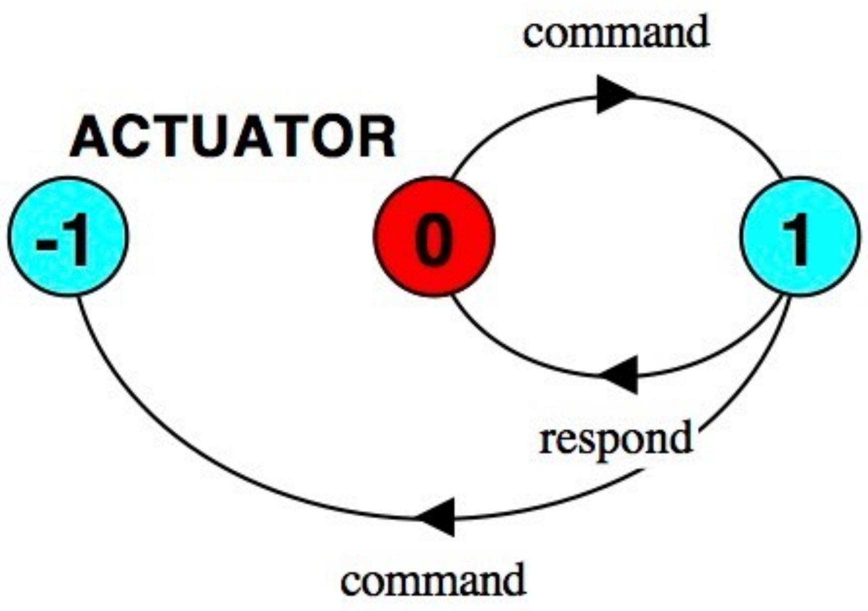
Roadmap

- > Modelling interaction in FSP
- > Safety — synchronizing critical sections
- > Locking for atomicity
- > The busy-wait mutual exclusion protocol
- > **Checking Safety properties**
- > Conditional synchronization

Safety revisited

A safety property asserts that nothing bad happens

Use the `ERROR` process (-1) to indicate erroneous behaviour



```
ACTUATOR  =  ( command->ACTION),
ACTION    =  ( respond->ACTUATOR
             | command->ERROR).
```

```
Trace to property violation in ACTUATOR:
      command
      command
```

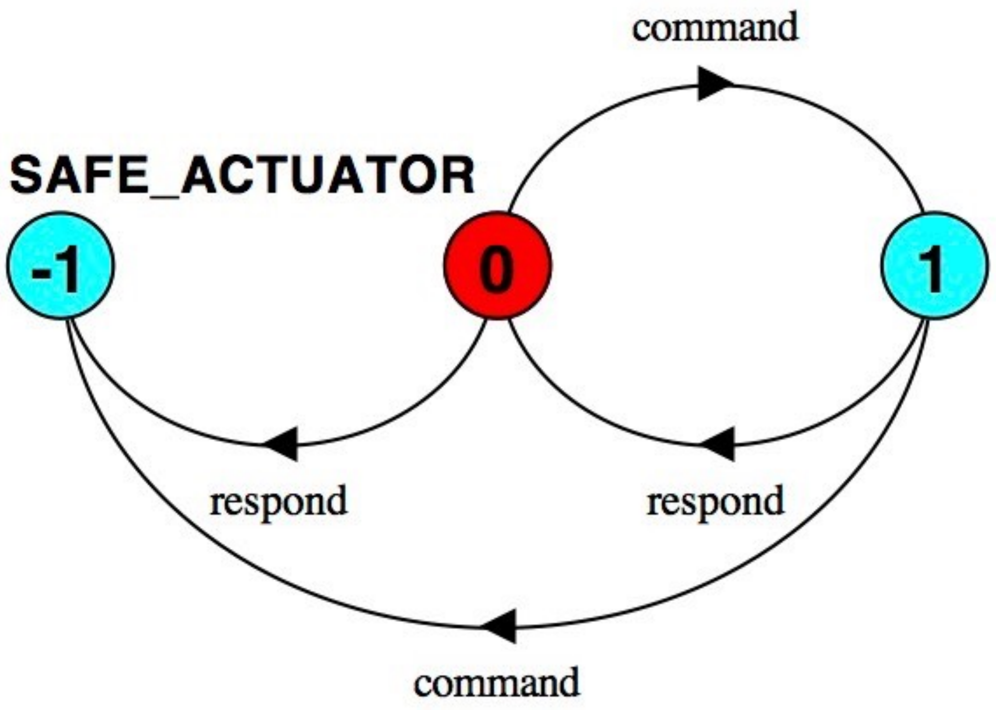
7-actuator.lts

In this example we explicitly define which traces should lead to an ERROR state. This is clumsy, and may accidentally leave out some erroneous traces. (For example, “`respond`” should also not be possible in the initial state.)

Safety — property specification

`ERROR` conditions state what is not required.

In complex systems, it is usually better to specify directly what *is* required.



```
property SAFE_ACTUATOR
   = ( command
     -> respond
     -> SAFE_ACTUATOR ).
```

```
Trace to property violation in SAFE_ACTUATOR:
      respond
```

8-checkActuator.lts

A much better way to express safety conditions is as a property.

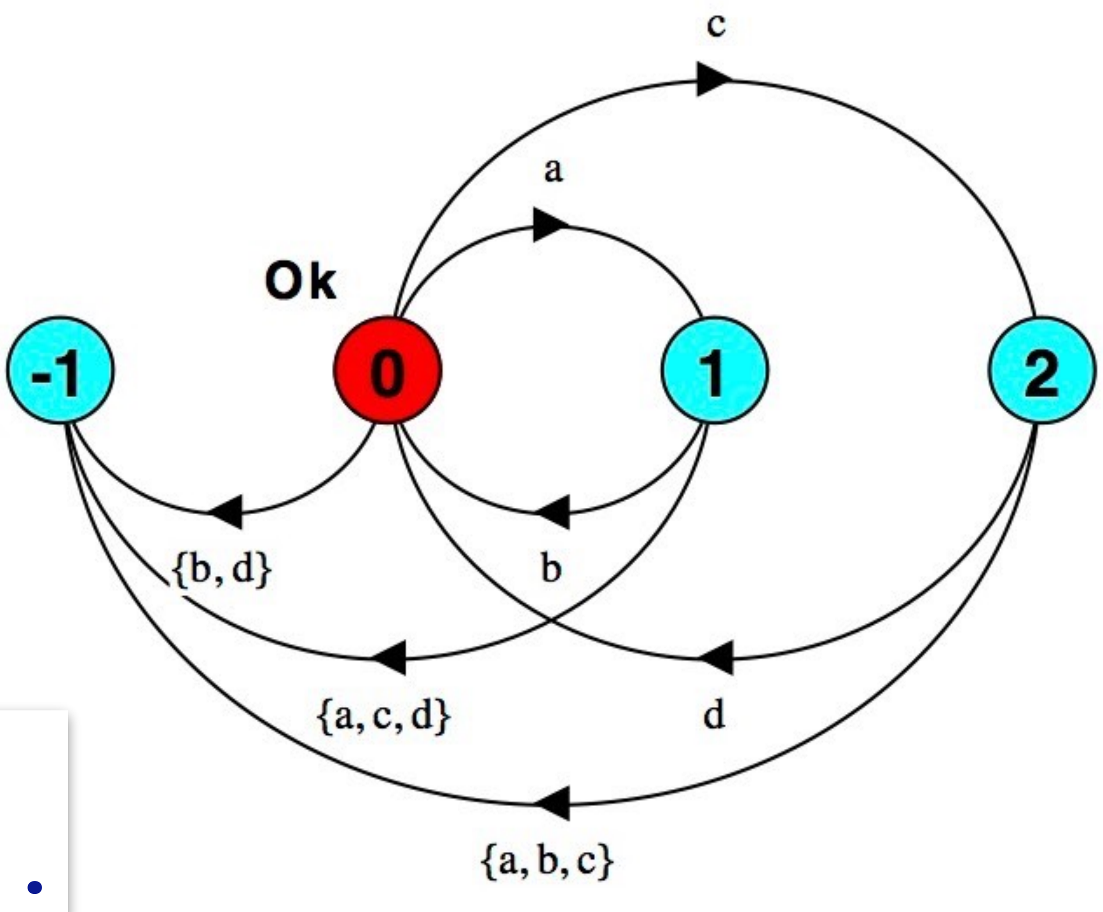
A safety property is an FSP expression that states which actions are allowed; all other actions are assumed to lead to an `ERROR`.

Notice here that `SAFE_ACTUATOR` only specifies that the actions command and respond must alternative. Since it is defined as a property, LTSA automatically expands this to a process that leads to an error state if any unspecified actions occur.

We can now compose this property with our original system and LTSA will either verify that `ERROR` is unreachable, or it will provide a trace that proves the property is violated.

Checking Busy-Wait (revisited)

Ok is a *deterministic process* that specifies which traces are safe — everything else leads to ERROR.

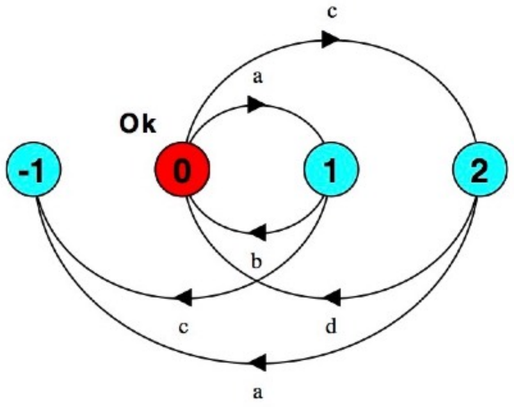


Contrast:

```
property Ok = (  a -> b -> Ok
              |  c -> d -> Ok ).
```

with:

```
Ok = ( a -> ( c -> ERROR | b -> Ok )
     | c -> ( a -> ERROR | d -> Ok )).
```



9-busywait2.lts

Now we are ready to specify the safety property for our busy-wait protocol. The property `Ok` states that either the first critical section or the second one may complete, but they may not overlap. Everything else leads to an error.

Notice how the property expresses additional error traces that we missed with our first attempt (e.g., `aa`, or `cb`).

As we shall see in the next slide, it is essential that a property be *deterministic*.

Safety properties

A safety property P defines a *deterministic process* that asserts that any trace including actions in the alphabet of P is accepted by P.

Transparency of safety properties:

> Since all actions in the alphabet of a property are eligible choices, composing a property with a set of processes *does not affect their correct behaviour.*

> If a behaviour can occur which violates the safety property, then ERROR is reachable.

A safety property is a process that is composed with a system to check that an error state cannot be reached. The safety property *must not alter the behaviour of the system being checked*, that is, it must be *transparent*.

If a process is non-deterministic, then when it is composed with an existing system, it can arbitrarily decide to take one path rather than another. This means that such a process alters the behaviour of the base system. As a consequence such a process is not transparent and cannot be a safety property.

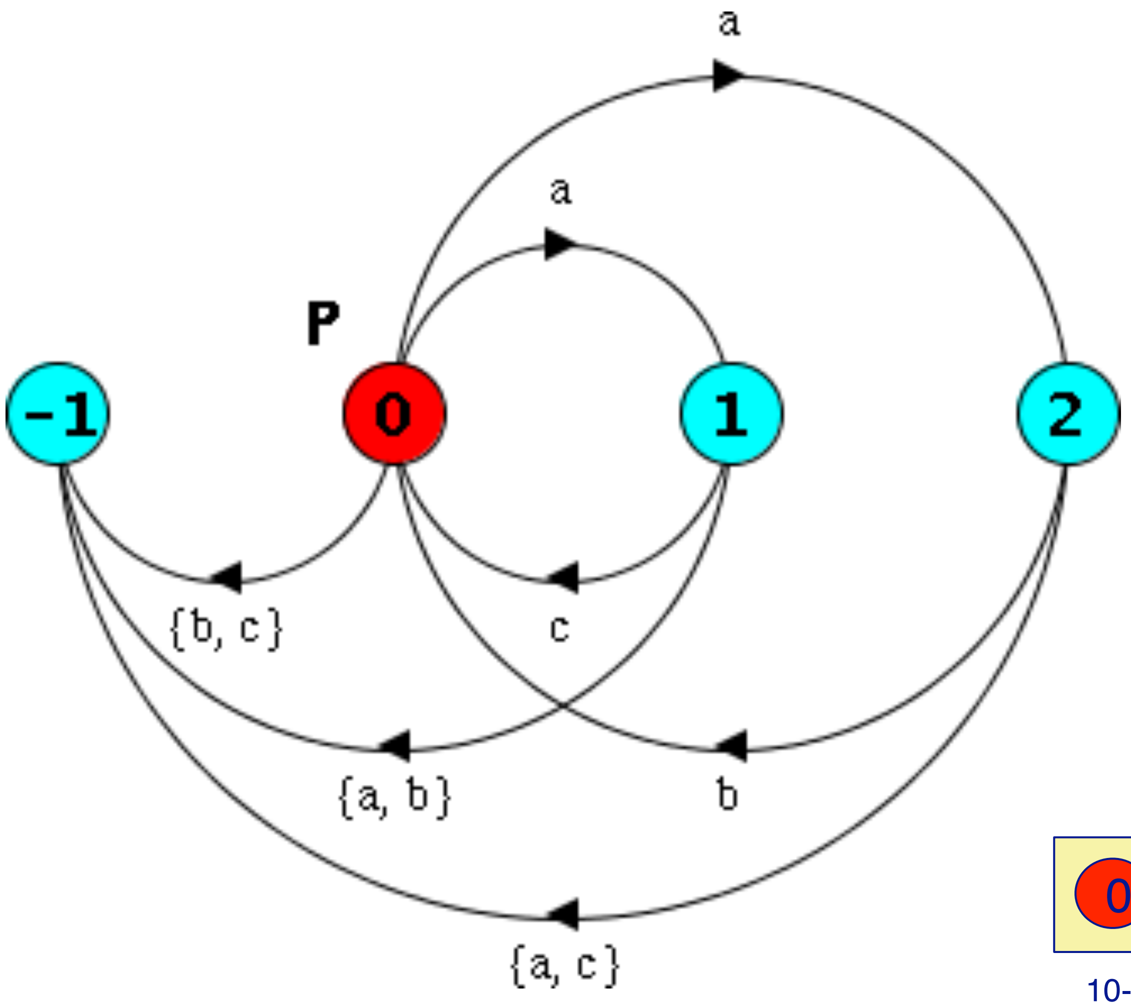
Transparency

Why must properties be deterministic to be transparent?

Consider:

```
property P = (a->b->P|a->c->P).
```

Is `a->b` allowed or not?



10-NDprop.lts

This example won’t compile because LTSA detects that the property is ND. Instead we could manually specify an equivalent process:

```
PropertyP = P,
P  = (a -> P1 | a -> P2 |{b,c} -> ERROR),
P1 = (b -> P | {a,c} -> ERROR),
P2 = (c -> P | {a,b} -> ERROR).
```

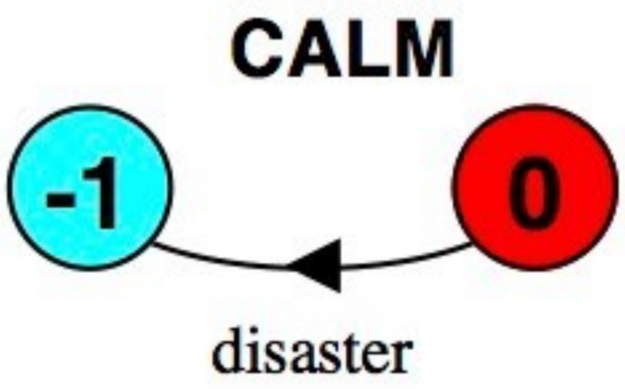
This is still ND. What we really want is the following:

```
property P = (a ->{b,c}->P).
```

This is deterministic, and states that after action `a`, either `b` or `c` may follow.

Safety properties

How can we specify that some action, disaster, never occurs?



```
property CALM = STOP + {disaster}.
```

A safety property must be specified so as to include all the acceptable, valid behaviours in its alphabet.

LTSA expands a property with actions to the `ERROR` state only using actions in the alphabet of the process. In this case, `STOP` has an *empty alphabet*, so we must explicitly add the actions that must not take place.

Roadmap

- > Modelling interaction in FSP
- > Safety — synchronizing critical sections
- > Locking for atomicity
- > The busy-wait mutual exclusion protocol
- > Checking Safety properties
- > **Conditional synchronization**

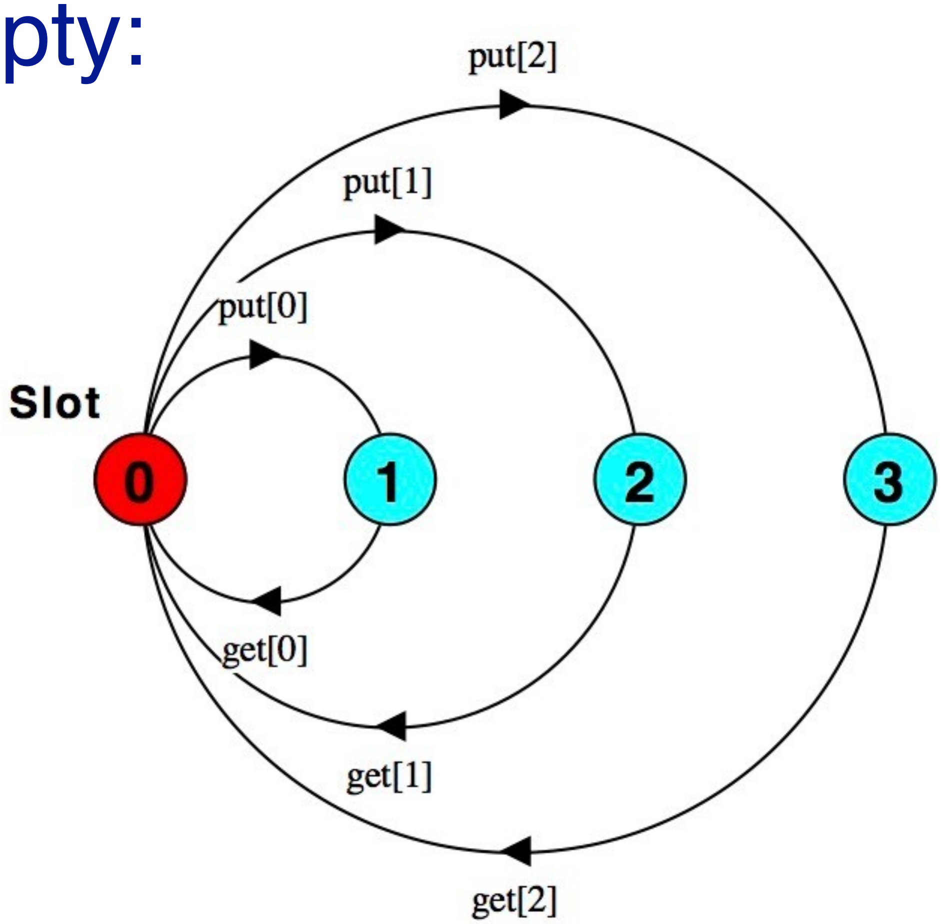
Conditional synchronization

A lock *delays* an acquire request if it is already locked:

```
LOCK = ( acquire -> release -> LOCK ).
```

Similarly, a one-slot buffer delays a put request if it is full and delays a get request if it is empty:

```
const N = 2
Slot = ( put[v:0..N]
        ->get[v]
        ->Slot ).
```

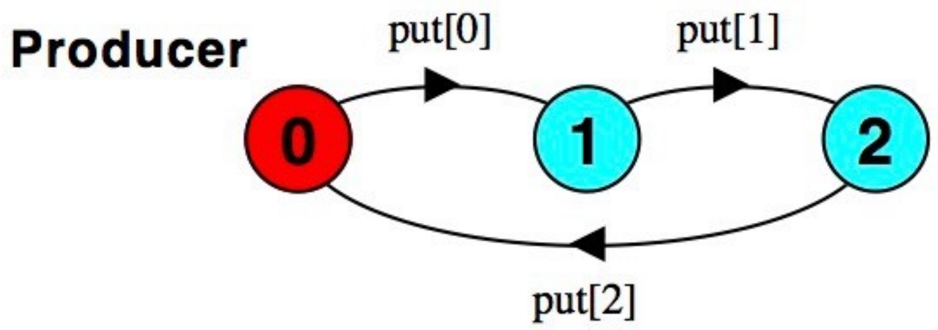


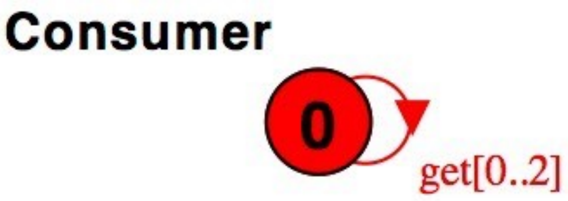
Producer/Consumer composition

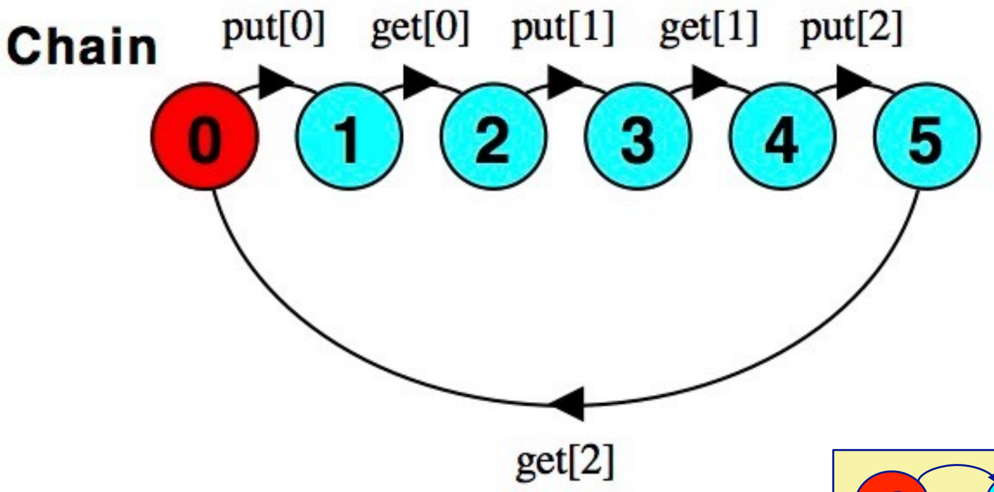
```
Producer =  ( put[0]
           -> put[1]
           -> put[2]
           -> Producer ).

Consumer =  ( get[x:0..N]
            -> Consumer ).

||Chain = (   Producer
           || Slot
           || Consumer )
```







11-slot.lts

Notice how the Slot is used to synchronize the behaviour of the Producer and the Consumer.

Wait and notify

A Java object whose methods are all synchronized behaves like a monitor

Within a synchronized method or block:

> `wait()` suspends the current thread, *releasing the lock*

> `notify()` wakes up *one thread waiting on that object*

> `notifyAll()` *wakes up all threads* waiting on that object

NB: Outside of a synchronized block, `wait()` and `notify()` will raise an `IllegalMonitorStateException`

Always use notifyAll() unless you are sure it doesn't matter which thread you wake up!

Slot (put)

```
class Slot<Value> implements Buffer<Value> {
   private Value slotVal;        // initially null

   public synchronized void put(Value val) {
      while (slotVal != null) {
         try { wait(); }         // become NotRunnable
         catch (InterruptedException e) { }
      }
      slotVal = val;
      notifyAll();               // make waiting threads Runnable
      return;
   }
...
```

Slot

```
interface Buffer<Value> {
   public void put(Value val);
   public Value get();
}
```

Note the idiomatic structure of a Java monitor visible in this method and the next:

- All public methods are `synchronized`.
- Any synchronization condition is checked *before* taking any other action.
- The condition is checked within a while loop, and `wait()` is performed within a try-catch statement.
- Finally, after changing state but just before completing the method, `notifyAll()` is signalled to wake up *all* waiting threads.

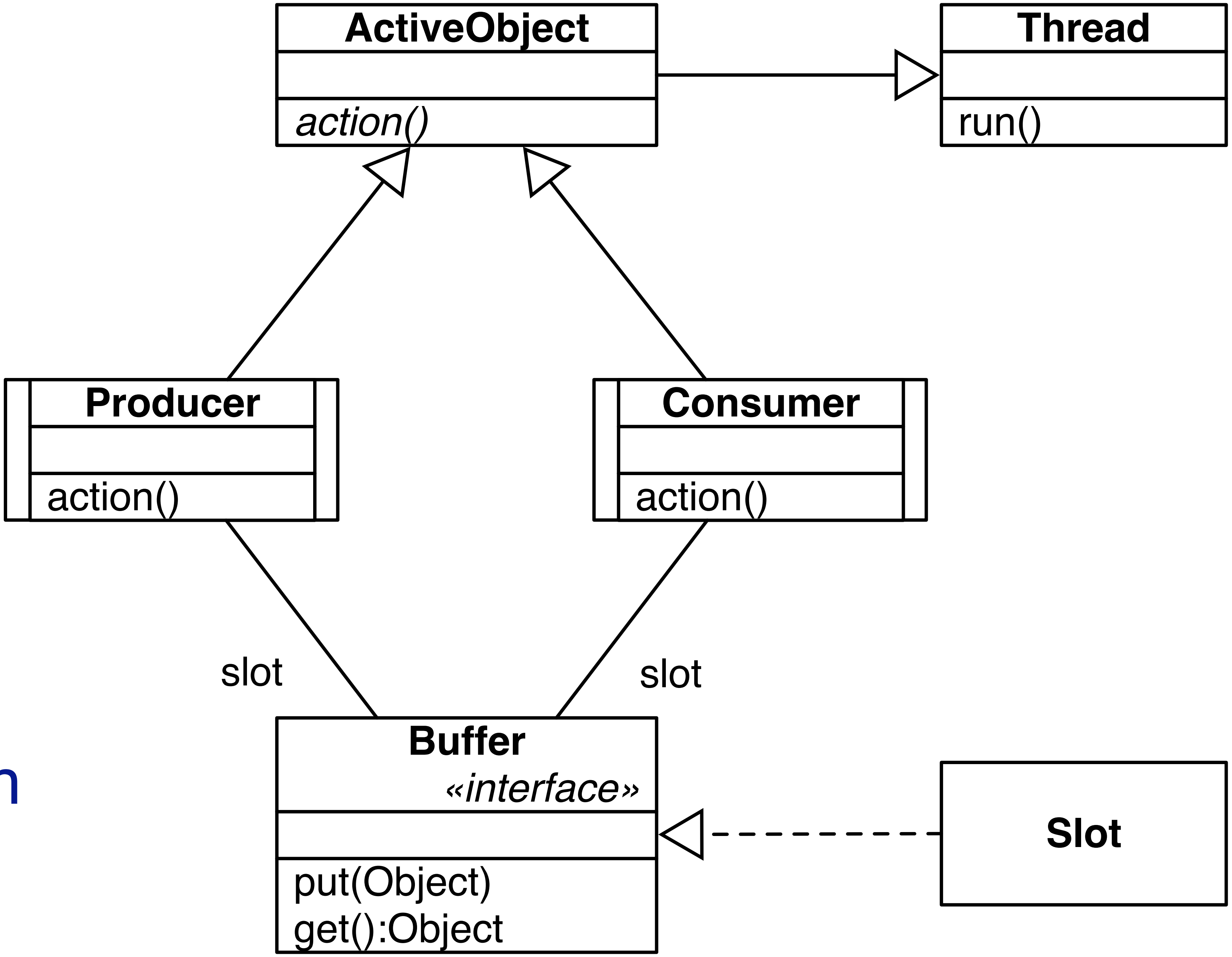
Slot (get)

```
...
   public synchronized Value get() {
      Value rval;
      while (slotVal == null) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      rval = slotVal;
      slotVal = null;
      notifyAll();
      return rval;
   }
}
```

Slots and Active Objects

Active objects have their own thread.

Producers and Consumers are active objects that communicate and synchronize through a shared buffer.



NB: in UML active objects are designated using class boxes with sidebars.

Active objects

```
abstract class ActiveObject extends Thread {
  protected int count;
  ActiveObject(String name, int count) {
     super(name);
     this.count = count;
  }
  public void run() {
     int i;
     for (i=1;i<=count;i++) {
        this.action(i);
     }
  }
  protected abstract void action(int n);
}
```

Slot

An active object has a thread of its own.

Producer in Java

A generic Producer puts `count` messages to the slot:

```
abstract class Producer<Value> extends ActiveObject {
  protected Buffer<Value> slot;
  Producer(String name, int count, Buffer<Value> slot) {
     super(name, count);
     this.slot = slot;
  }
  protected void action(int n) {
     slot.put(produce(n));
  }
  protected abstract Value produce(int n);
}
```

Slot

Consumer in Java

... and the Consumer gets them:

```
abstract class Consumer<Value> extends ActiveObject {
   protected Buffer<Value> slot;
   Consumer(String name, int count, Buffer<Value> slot) {
      super(name, count);
      this.slot = slot;
   }
   protected void action(int n) {
      consume(n, slot.get());
   }
   protected abstract void consume(int n, Value val);
}
```

Fruit producers and consumers

```
public class ProducerConsumerDemo {
  ...
  private class FruitProducer extends Producer<String> {
     protected String wares;
     FruitProducer(...) { ... }
     protected String produce(int n) {
        String message;
        message = wares + "(" + n + ")";
        System.out.println(getName() + " put " + message);
        return message;
     }
  }

  private class FruitConsumer extends Consumer<String> {
     ...
  }
}
```

Composing Producers and Consumers

Multiple producers and consumers may share the buffer:

```
public class ProducerConsumerDemo {
  static int COUNT = 5;
  ...
  public void demo() {
    Buffer<String> slot = new Slot<String>();

    new FruitProducer("Peter", COUNT, slot, "apple").start();
    new FruitProducer("Paula", COUNT, slot, "orange").start();
    new FruitProducer("Patricia", COUNT, slot, "banana").start();

    new FruitConsumer("Carla", COUNT, slot).start();
    new FruitConsumer("Cris", 2*COUNT, slot).start();
  }
  ...
}
```

```
Peter put apple (1)
Carla got apple (1)
Paula put orange(1)
Cris got orange(1)
Patricia put banana(1)
Carla got banana(1)
Peter put apple (2)
Cris got apple (2)
Patricia put banana(2)
Carla got banana(2)
Peter put apple (3)
Cris got apple (3)
Paula put orange(2)
Carla got orange(2)
Patricia put banana(3)
Cris got banana(3)
Peter put apple (4)
Cris got apple (4)
Peter put apple (5)
Carla got apple (5)
Paula put orange(3)
Cris got orange(3)
Patricia put banana(4)
Cris got banana(4)
Patricia put banana(5)
Cris got banana(5)
Paula put orange(4)
Cris got orange(4)
Paula put orange(5)
Cris got orange(5)
```

What you should know!

- *How do you model interaction with FSP?*
- *What is a critical section? What is critical about it?*
- *Why don’t sequential programs need synchronization?*
- *How do locks address safety problems?*
- *What primitives do you need to implement the busy-wait mutex protocol?*
- *How can you use FSP to check for safety violations?*
- *What happens if you call wait or notify outside a synchronized method or block?*
- *When is it safe to use notifyAll()?*
- *What are safety properties? How are they modelled in FSP?*

Can you answer these questions?

- *What is an example of an invariant that might be violated by interfering, concurrent threads?*
- *What constitute atomic actions in Java?*
- *Can you ensure safety in concurrent programs without using locks?*
- *When should you use synchronize(this) rather than synchronize(someObject)?*
- *Is the busy-wait mutex protocol fair? Deadlock-free?*
- *How would you implement a Lock class in Java?*
- *Why is the Java Slot class so much more complex than the FSP Slot specification?*
- *How would you manually check a safety property?*
- *Why must safety properties be deterministic to be transparent?*