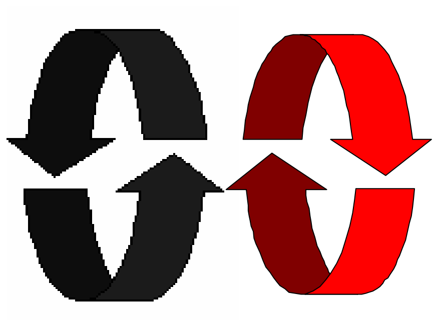
7 - Safety & Liveness Properties



Alexandre David
adavid@cs.aau.dk

Credits for the slides:
Claus Braband
Jeff Magee & Jeff Kramer

Repetition - Deadlock

- Concepts
  - deadlock (no further progress)
  - 4x necessary and sufficient conditions

- Models
  - no eligible actions (analysis gives shortest path trace)

- Practice
  - blocked threads

Aim - deadlock avoidance:

"Design systems where deadlock cannot occur".

Repetition - Necessary and Sufficient Conditions

1. **Serially reusable resources:**

the processes involved share resources which they use under mutual exclusion.

2. **Incremental acquisition:**

processes hold on to resources already allocated to them while waiting to acquire additional resources.

3. **No pre-emption:**

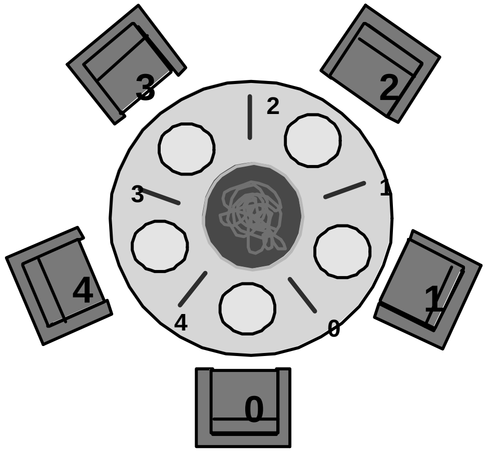
once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

4. **Wait-for cycle:**

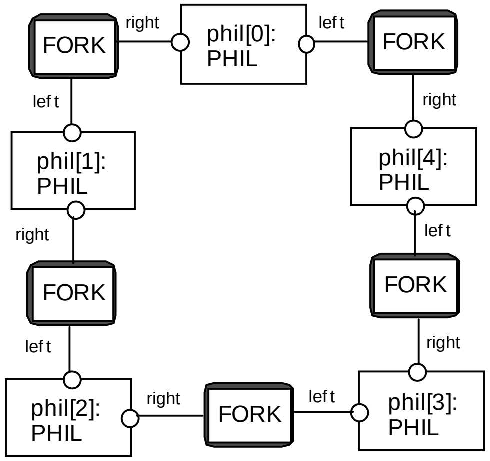
a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Repetition - Dining Philosophers

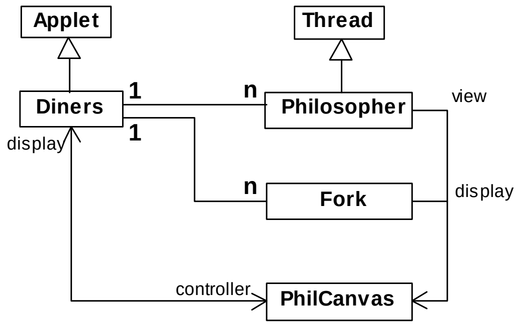
- Concepts

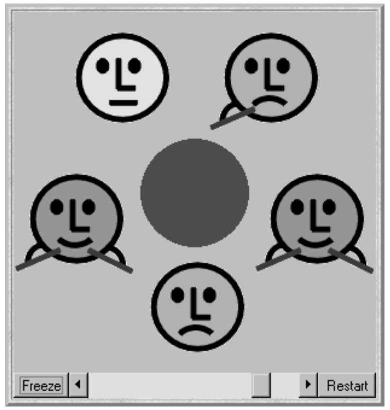


- Models



- Practice





Safety & Liveness Properties

Concepts:

Properties:	true for every possible execution
Safety:	nothing bad happens
Liveness:	something good *eventually* happens

Models:

Safety:	no reachable ERROR/STOP state
Progress:	an action is *eventually* executed
	(fair choice and action priority)

Practice:

Threads and monitors

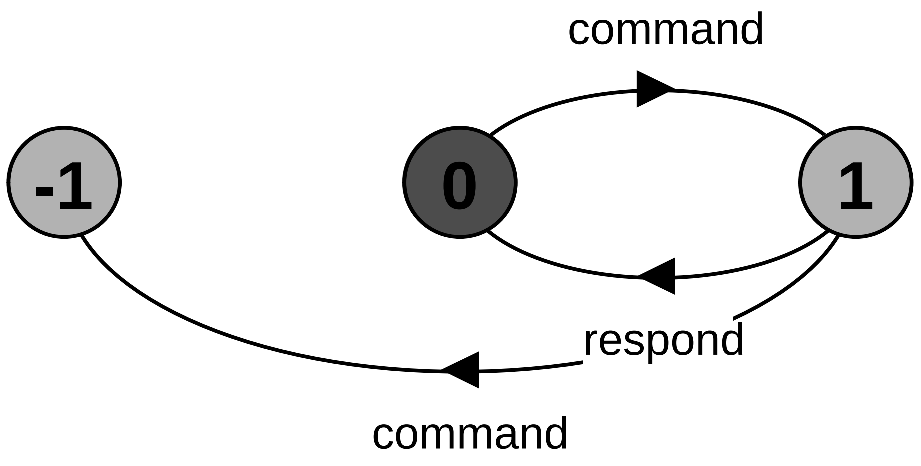
Aim: property satisfaction.

7.1 Safety

A safety property asserts that nothing bad happens.

♦ STOP or deadlocked state (no outgoing transitions)

♦ ERROR process (-1) to detect erroneous behaviour



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

♦ Analysis using LTSA:

(shortest trace)

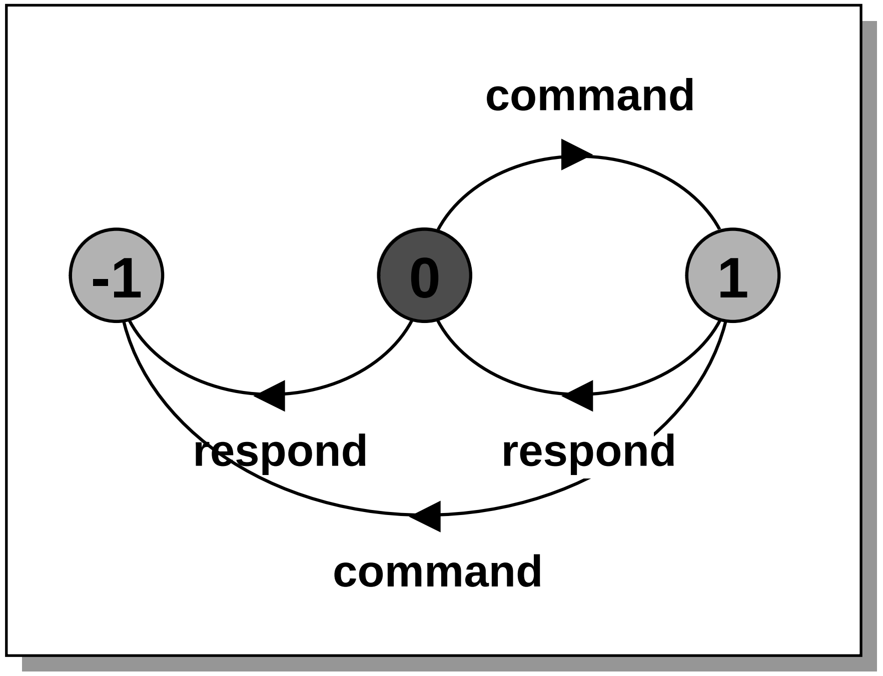
```
Trace to ERROR:
  command
  command
```

Safety - Property Specification

- **ERROR** conditions state what is **not** required (~ exceptions).
- In complex systems, it is usually better to specify **safety properties** by stating directly what **is** required.

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

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



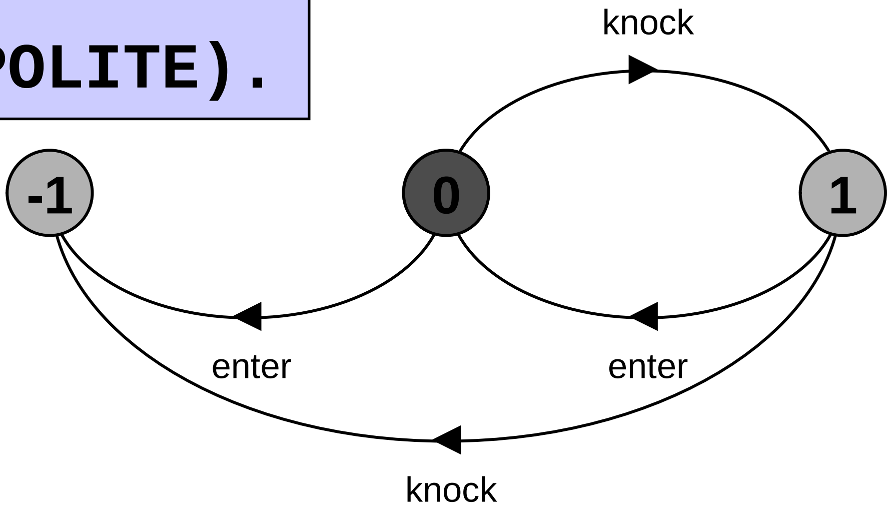
Safety Properties

Property that it is polite to knock before entering a room.

Traces:

```
        knock->enter     ☺
        enter            ☹
        knock->knock     ☹
```

```
property POLITE
  = (knock -> enter -> POLITE).
```



Note: *In **all** states, **all** the actions in the alphabet of a property are eligible choices.*

Safety Properties

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

Thus, if **P** is composed with **S**, then *traces of actions* in the alphabet $\alpha(\mathbf{S}) \cap \alpha(\mathbf{P})$ must also be valid *traces* of **P**, otherwise **ERROR** is reachable.

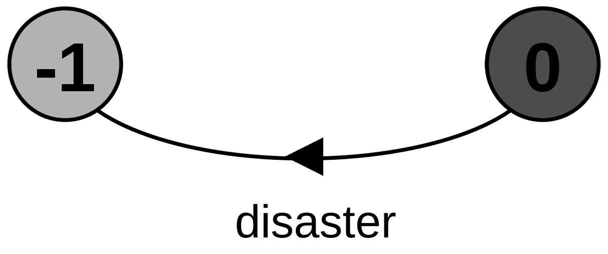
Transparency of safety properties:

Since all actions in the alphabet of a property are eligible choices => composition with S ***does not affect*** *its correct behavior.*

However, if a bad behavior can occur (violating the safety property), then **ERROR** *is reachable.*

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 behaviors in its alphabet.

Safety Property – How to See Them

- The model is for the ***implementation***
- The property is for the ***specification***
- The implementation must meet its specification

- In the LTS tool both models and properties are described using FSP.
- They will be similar because they are using the same basic language but they do not represent the same thing.
- In our simple examples, the specification is so simple that it may be enough to describe the implementation – don't be fooled by this.
- Normally: model = implementation, formula = specification. "Difference" here: you don't implement the specification!

Safety - Mutual Exclusion

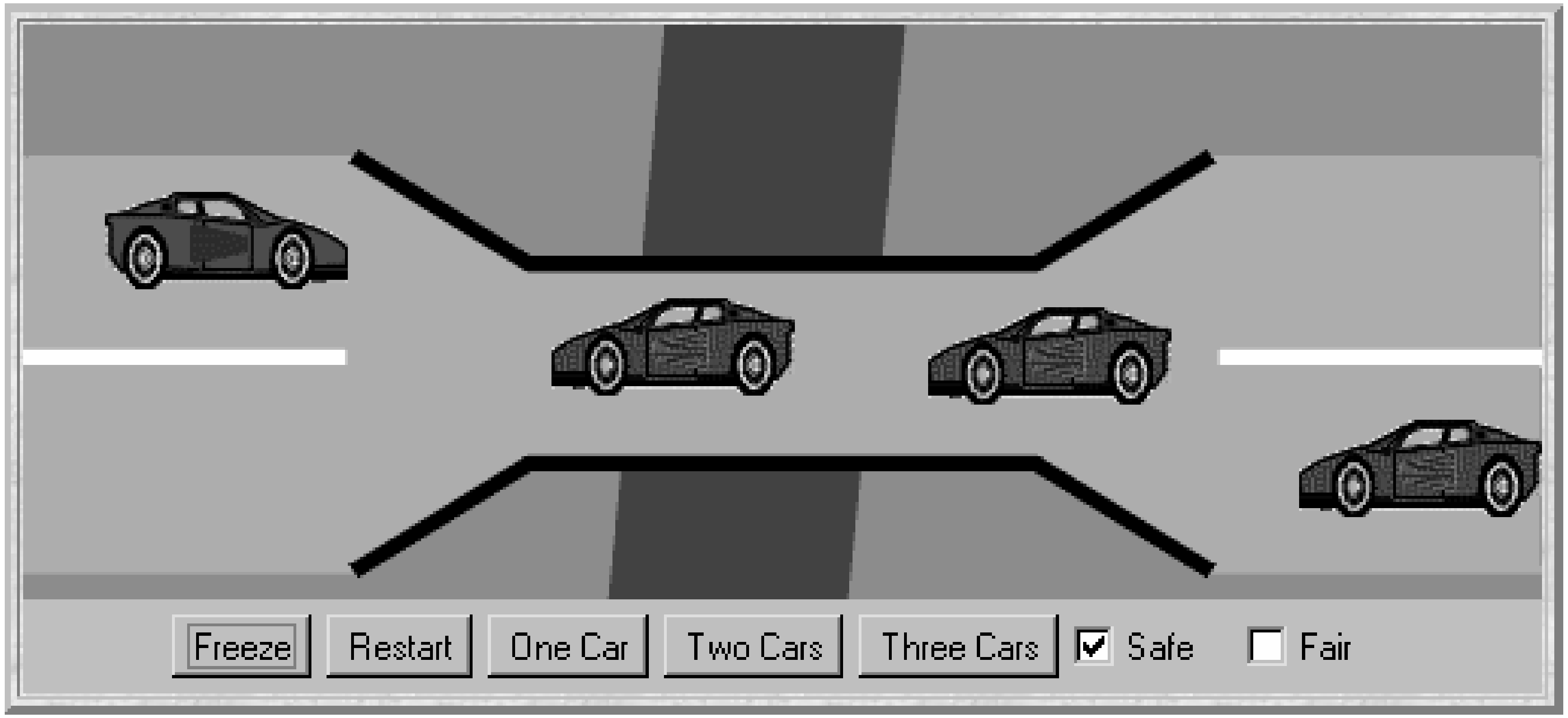
```
LOOP =
  (mutex.down->enter->exit->mutex.up->LOOP).

||SEMADEMO = (p[1..3]:LOOP ||
             {p[1..3]}::mutex:SEMAPHORE(1)).
```

How do we check that this does indeed ensure mutual exclusion in the critical section?

```
property MUTEX =
  (p[i:1..3].enter -> p[i].exit -> MUTEX).

||CHECK = (SEMADEMO || MUTEX).
```

Check safety using **LTSA**!   What if **SEMAPHORE(2)**?

7.2 Single Lane Bridge Problem



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 different directions enter the bridge at the same time.

Single Lane Bridge - Model

- Events or actions of interest?

  enter and exit

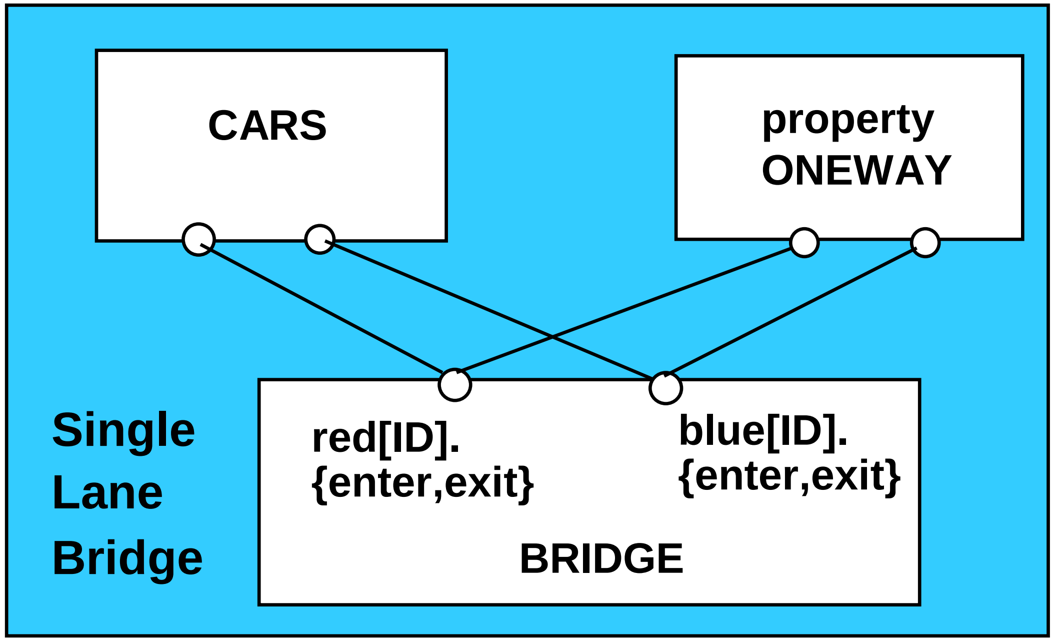
- Identify processes?

  cars and bridge

- Identify properties?

  "oneway"

Structure diagram:



Single Lane Bridge - CARS model

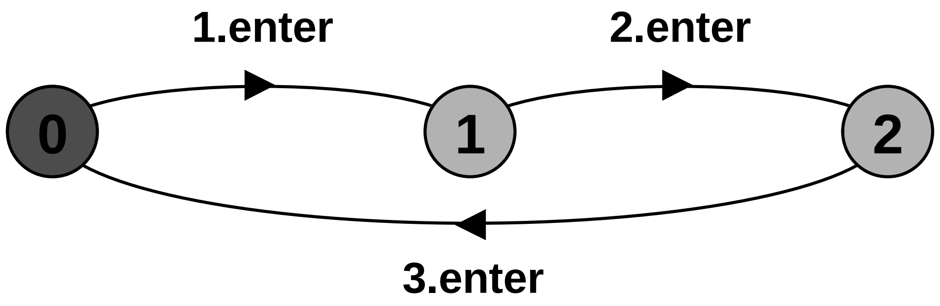
```
const N = 3         // number of each type of car
range T = 0..N      // type of car count
range ID= 1..N      // car identities

CAR = (enter->exit->CAR).
```

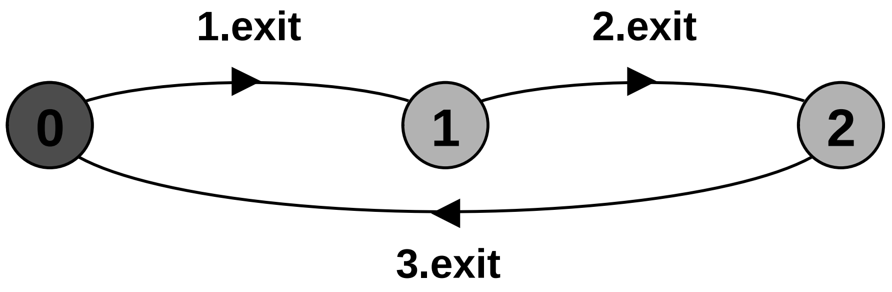
To model the fact that cars cannot pass each other on the bridge, we model a **CONVOY** of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

```
||CARS = (red:CONVOY || blue:CONVOY).
```

Single Lane Bridge - CONVOY model

```
NOPASS1  = C[1],                //preserves entry order
C[i:ID]  = ([i].enter-> C[i%N+1]).
NOPASS2  = C[1],                //preserves exit order
C[i:ID]  = ([i].exit-> C[i%N+1]).

||CONVOY = ([ID]:CAR||NOPASS1||NOPASS2).
```





Permits 1.enter→ 2.enter→ 1.exit→ 2.exit

but not 1.enter→ 2.enter→ 2.exit→ 1.exit

ie. no overtaking.

Single Lane Bridge - BRIDGE Model

Cars can move concurrently on bridge, but only in the same direction.

The bridge maintains a count of blue and red cars on it.

Red cars are only allowed to enter when the blue count is 0 (and vice-versa).

```
BRIDGE = BRIDGE[0][0], // initially empty bridge
BRIDGE[nr:T][nb:T] =   // nr: #red; nb: #blue
  (when (nb==0) red[ID].enter -> BRIDGE[nr+1][nb]
  |             red[ID].exit  -> BRIDGE[nr-1][nb]
  |when (nr==0) blue[ID].enter-> BRIDGE[nr][nb+1]
  |             blue[ID].exit -> BRIDGE[nr][nb-1]
  ).
```

Single Lane Bridge - **BRIDGE** Model

Even when 0, **exit** actions permit the car counts to be decremented:

```
Warning - BRIDGE.-1.0 defined to be ERROR
Warning - BRIDGE.0.-1 defined to be ERROR
Warning - BRIDGE.-1.1 defined to be ERROR
Warning - BRIDGE.-1.2 defined to be ERROR
Warning - BRIDGE.-1.3 defined to be ERROR
Warning - BRIDGE.0.4 defined to be ERROR
Warning - BRIDGE.1.-1 defined to be ERROR
Warning - BRIDGE.2.-1 defined to be ERROR
Warning - BRIDGE.4.0 defined to be ERROR
Warning - BRIDGE.3.-1 defined to be ERROR
Compiled: BRIDGE
```

Recall that **LTSA** maps such *undefined states* to **ERROR**.

Single Lane Bridge - Safety Property “ONEWAY”

We now specify a **safety property** to check that cars do not collide!:

```
property ONEWAY = (red[ID].enter  -> RED[1]
                  |blue[ID].enter -> BLUE[1]),

RED[i:ID] = (red[ID].enter  -> RED[i+1]
            |when (i==1) red[ID].exit  -> ONEWAY
            |when (i>1)  red[ID].exit  -> RED[i-1]),

BLUE[j:ID]= (blue[ID].enter -> BLUE[j+1]
            |when (j==1) blue[ID].exit -> ONEWAY
            |when (j>1)  blue[ID].exit -> BLUE[j-1]).
```

When the bridge is empty, either a red or a blue car may enter. While red cars are on the bridge only red cars can enter; similarly for blue cars.

Single Lane Bridge - Model Analysis

```
||SingleLaneBridge = (CARS||BRIDGE||ONEWAY).
```

Is the safety property **"ONEWAY"** *violated?*

```
No deadlocks/errors
```

...And ***without*** *the BRIDGE:*

```
||SingleLaneBridge = (CARS || ONEWAY).
```

Is the safety property **"ONEWAY"** *violated?*

```
Trace to property violation
                   in ONEWAY:
     red.1.enter
     blue.1.enter
```

Implementation/Property?

```
BRIDGE = BRIDGE[0][0], // initially empty bridge
BRIDGE[nr:T][nb:T] =   // nr: #red; nb: #blue
  (when (nb==0) red[ID].enter -> BRIDGE[nr+1][0]
  |             red[ID].exit  -> BRIDGE[nr-1][nb]
  |when (nr==0) blue[ID].enter-> BRIDGE[0][nb+1]
  |             blue[ID].exit -> BRIDGE[nr][nb-1]).
```

```
property ONEWAY = (red[ID].enter  -> RED[1]
                  |blue[ID].enter -> BLUE[1]),

RED[i:ID] = (red[ID].enter  -> RED[i+1]
            |when (i==1) red[ID].exit  -> ONEWAY
            |when (i>1)  red[ID].exit  -> RED[i-1]),

BLUE[j:ID]= (blue[ID].enter -> BLUE[j+1]
            |when (j==1) blue[ID].exit -> ONEWAY
            |when (j>1)  blue[ID].exit -> BLUE[j-1]).
```

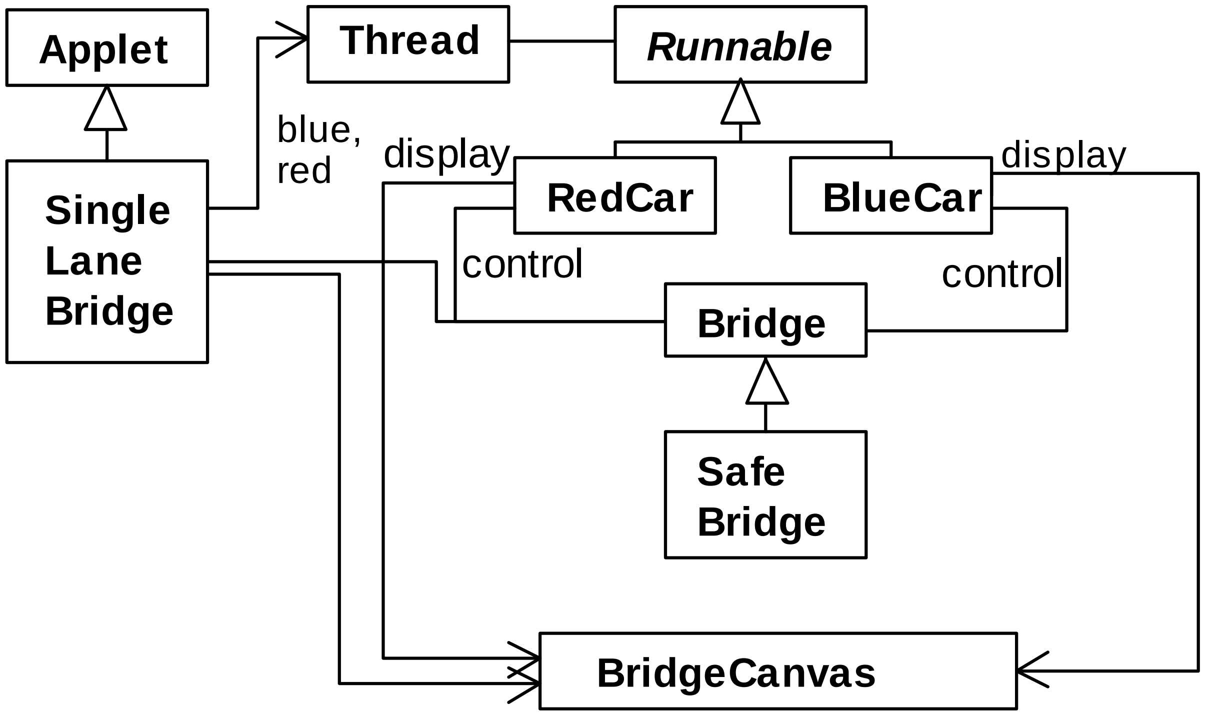
Implementation/Property?

- Controller implementation: from one “state”, we allow actions.
- Property observer: specify all the valid states and the sequences of actions from them.

- Our controller meets its specification (although it is sloppy).
- You cannot cheat here and use the controller as your specification (add property) because it allows wrong behaviours. The wrong behaviours do not occur because we have good cars. The specification describes good behaviours for the **controller with the cars**.

Single Lane Bridge - Implementation in Java (UML)

CAR (active => thread) ; BRIDGE (passive => monitor)



BridgeCanvas enforces no overtaking (~ ENTER_SEQ).

Single Lane Bridge - BridgeCanvas

An instance of **BridgeCanvas** class is created by the SingleLaneBridge applet.

```
class BridgeCanvas extends Canvas {
    public void init(int ncars) {…}  // set #cars

    public boolean moveRed(int i)  throws Int’Exc’{…}
    // moves red car #i a step  (if possible)
    // returns true if on bridge

    public boolean moveBlue(int i) throws Int’Exc’{…}
    // moves blue car #i a step (if possible)
    // returns true if on bridge

    public synchronized void freeze() {…}
    public synchronized void thaw()   {…}
}
```

Each Car object is passed a reference to the BridgeCanvas.

Single Lane Bridge - RedCar

```
class RedCar implements Runnable {
    Bridge control; BridgeCanvas display; int id;

    RedCar(Bridge b, BridgeCanvas d, int i) {
        control = b; display = d; id = i;
    }

    public void run() {
        try {
            while (true) {
                while (!display.moveRed(id)) ; // not on br.
                control.redEnter(); // req access to br.
                while (display.moveRed(id)) ;  // move on br
                control.redExit(); // release access to br.
            }
        } catch (InterruptedException _) {}
    }
}
```

Similarly for the `BlueCar`...

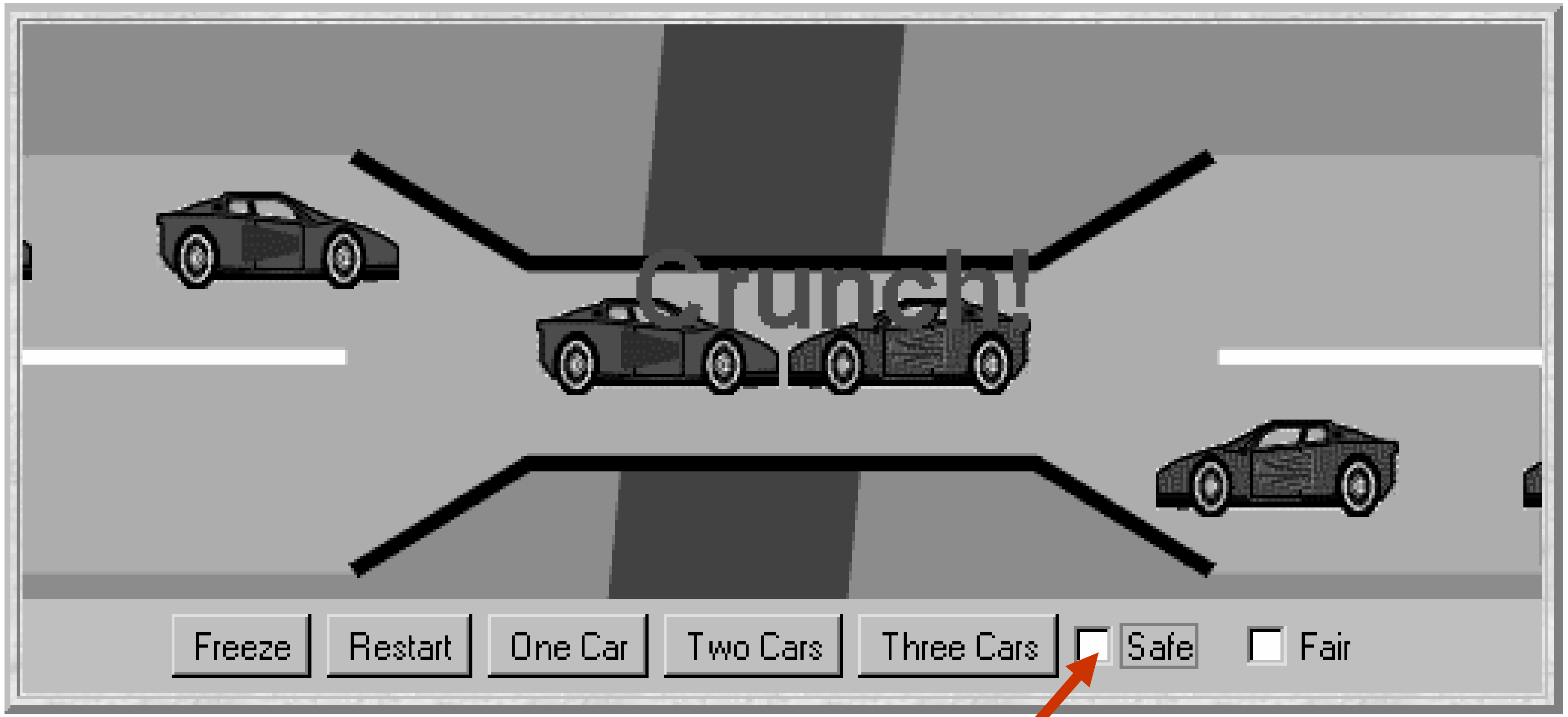
Single Lane Bridge - Class Bridge

```
class Bridge {
    synchronized void redEnter() throws Int’Exc’ {…}
    synchronized void redExit()  {…}
    synchronized void blueEnter() throws Int’Exc’ {…}
    synchronized void blueExit() {…}
}
```

Class **Bridge** provides a *null implementation* of the access methods i.e. no constraints on the access to the bridge.

Result………. ?

Single Lane Bridge



To ensure safety, the "safe" check box must be chosen in order to select the `SafeBridge` implementation.

Single Lane Bridge - SafeBridge

```
class SafeBridge extends Bridge {
    protected int nred  = 0; // #red cars on br.
    protected int nblue = 0; // #blue cars on br.

    // monitor invariant: nred≥0 ∧ nblue≥0 ∧
    //                    ¬(nred>0 ∧ nblue>0)

    synchronized void redEnter() throws Int’Exc’ {
        while (nblue>0) wait();
        ++nred;
    }

    synchronized void redExit() {
        --nred;
        if (nred==0) notifyAll();
    }
}
```

This is a direct translation from the **BRIDGE** model.

Single Lane Bridge - SafeBridge

```
synchronized void blueEnter() throws Int’Exc’ {
    while (nred>0) wait();
    ++nblue;
}

synchronized void blueExit() {
    --nblue;
    if (nblue==0) notifyAll();
}
```

To avoid unnecessary thread switches, we use *conditional notification* to wake up waiting threads only when the number of cars on the bridge is zero (i.e., when the last car leaves the bridge).

*But does every car **eventually** get an opportunity to cross the bridge? This is a **liveness** property.*

7.3 Liveness

A safety property **asserts** that nothing bad happens.

A liveness property **asserts** that something good ***eventually*** happens.

*Does every car eventually get an opportunity to cross the bridge (i.e., make **progress**)?*

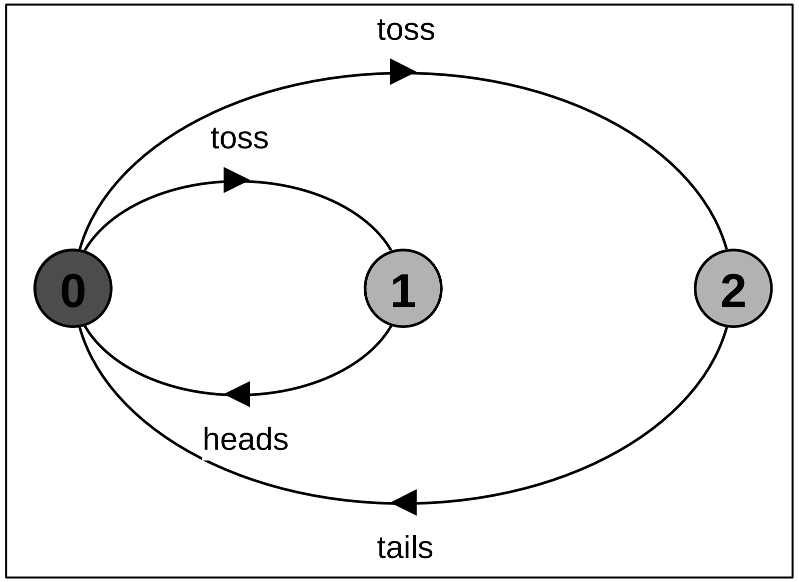
A progress property **asserts** that it is *always* the case that an action is *eventually* executed.

Progress is the opposite of *starvation* = the name given to a concurrent programming situation in which an action is never executed.

Progress Properties - Fair Choice

Fair Choice: If a choice over a *set of transitions* is executed infinitely often, then every transition in the set will be executed infinitely often.

```
COIN = (toss->heads->COIN
       |toss->tails->COIN).
```



How about if we chose
  toss(1) 100.000x;then
  toss(2) 1x; then
  toss(1) 100.000x; then
  toss(2) 1x; then …

Fair?

Progress Properties

```
progress P = {a1, a2, …, an}
```

This defines a *progress property*, **P**, which **asserts** that in an infinite execution, at least one of the actions **a_1**, **a_2**, …, **a_n** will be executed infinitely often.

```
COIN = (toss->heads->COIN | toss->tails->COIN).
```

```
progress HEADS = {heads} ?   ☺
progress TAILS = {tails} ?   ☺
```

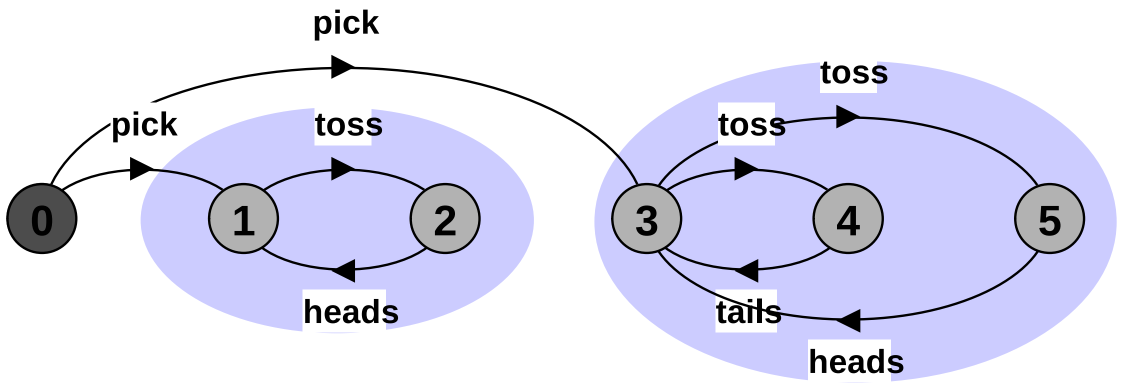
LTSA check progress:

```
No progress violations detected
```

Progress Properties

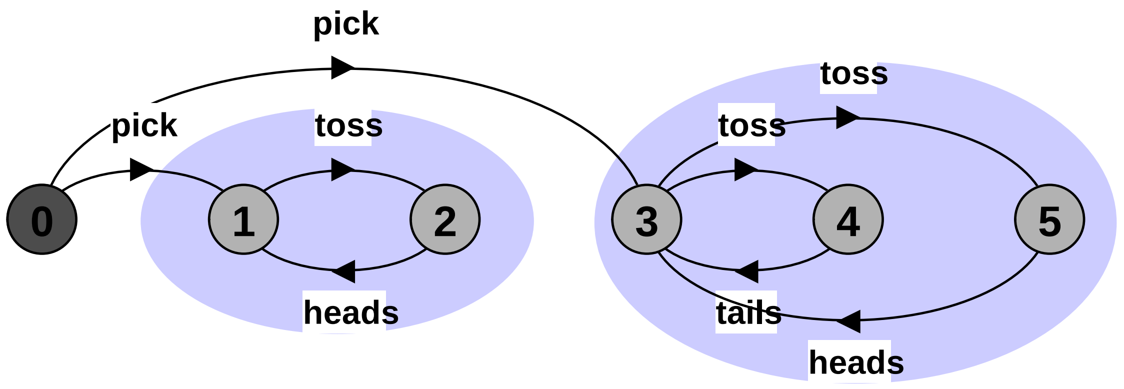
Suppose that there were **two** possible coins that could be picked up: *a regular coin* and *a **trick** coin*

```
TWOCOIN = (pick->COIN | pick->TRICK),
COIN    = (toss->heads->COIN | toss->tails->COIN).
TRICK   = (toss->heads->TRICK),
```



```
progress HEADS = {heads} ?   ☺
progress TAILS = {tails} ?   ☹
```

Progress Properties



```
progress HEADS = {heads}

progress TAILS = {tails}
```

```
Progress violation: TAILS
Path to terminal set of states:
     pick
Actions in terminal set:
{toss, heads}
```

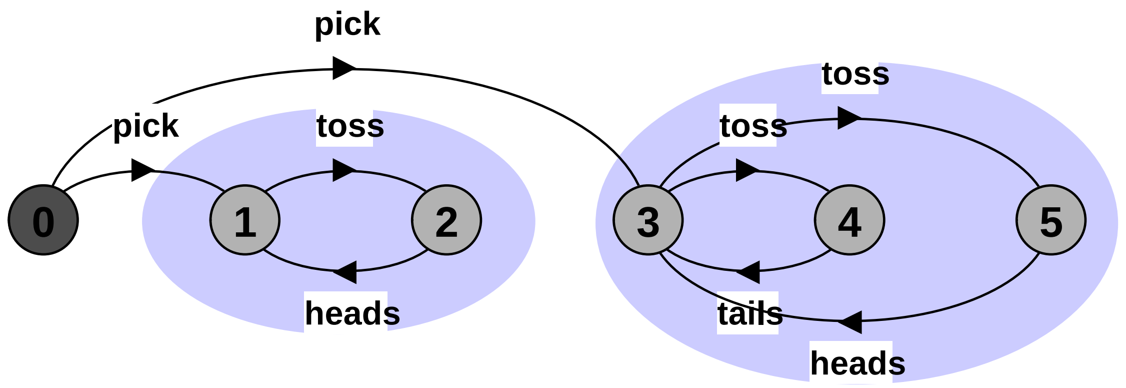
`progress HEADSorTails = {heads,tails} ?` ☺

Progress Analysis

A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for TWOCOIN:

{1,2} and {3,4,5}



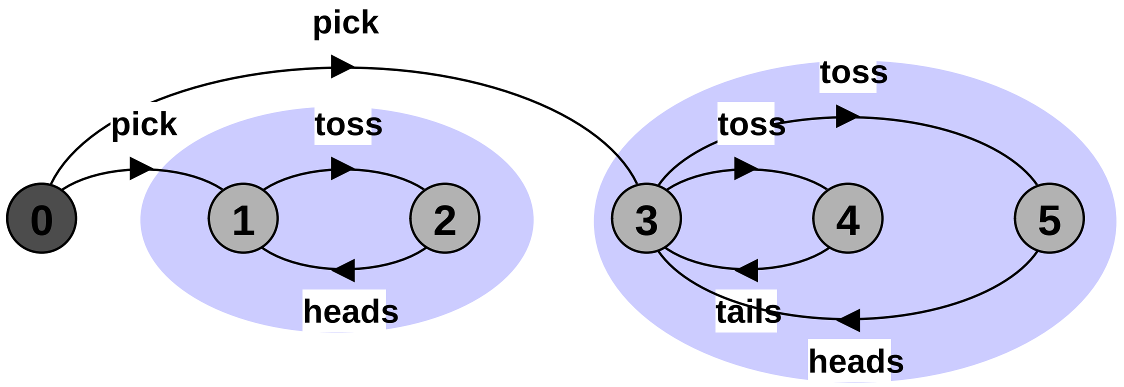
Given fair choice, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often.

Since there is no transition out of a terminal set, any action that is not used in the set cannot occur infinitely often in all executions of the system - and hence represents a potential progress violation!

Progress Analysis

A progress property is violated if analysis finds a *terminal set of states* in which **none** of the progress set actions appear.

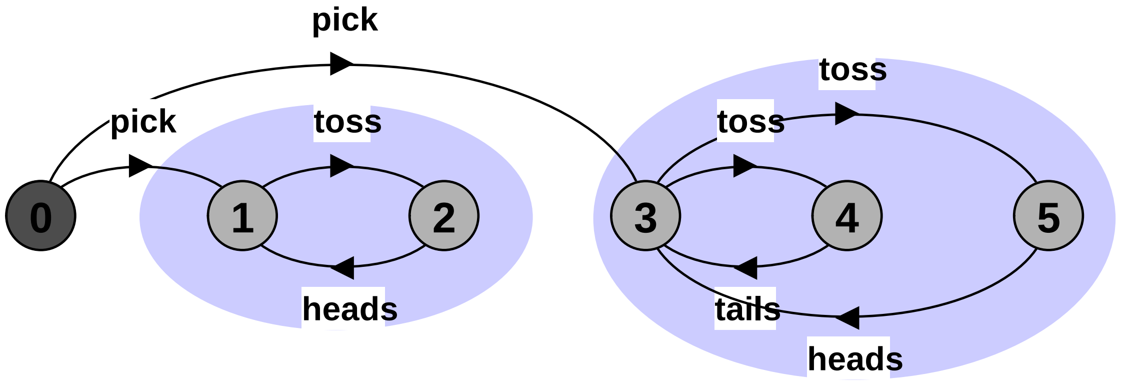
```
progress TAILS
   = {tails}
     in {1,2} ☹
```



Default progress: for *every* action in the alphabet, that action will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.

Progress Analysis – Default Progress

Default progress:



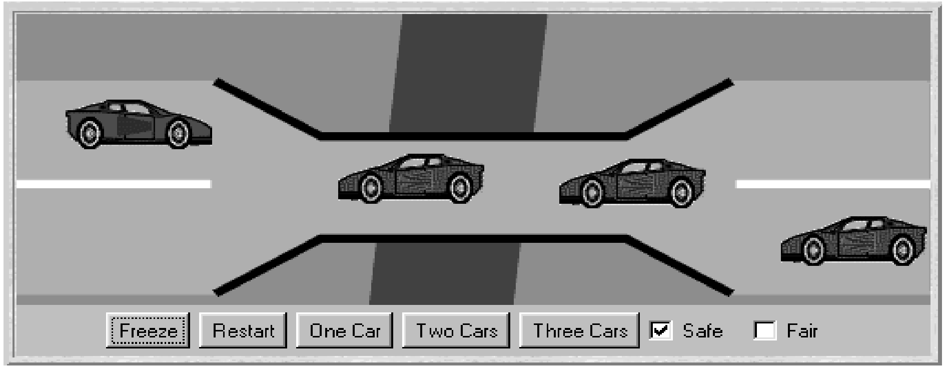
```
Progress violation for actions:
{pick}
Path to terminal set of states:
     pick
Actions in terminal set:
{toss, heads, tails}
```

```
Progress violation for actions:
{pick, tails}
Path to terminal set of states:
     pick
Actions in terminal set:
{toss, heads}
```

Note: default holds => every other progress property holds (i.e., every action is executed infinitely often and the system consists of a single terminal set of states).

Progress – Return of the Single Lane Bridge

Implementation exhibits progress violations:



```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS  = {red[ID].enter}
```

```
No progress violations detected.
```

In fact, no violations of default progress!

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must **superimpose** some scheduling policy for actions, which models the situation in which the bridge is congested.

Progress - Action Priority

Action priority expressions describe scheduling properties:

High Priority ("<<")

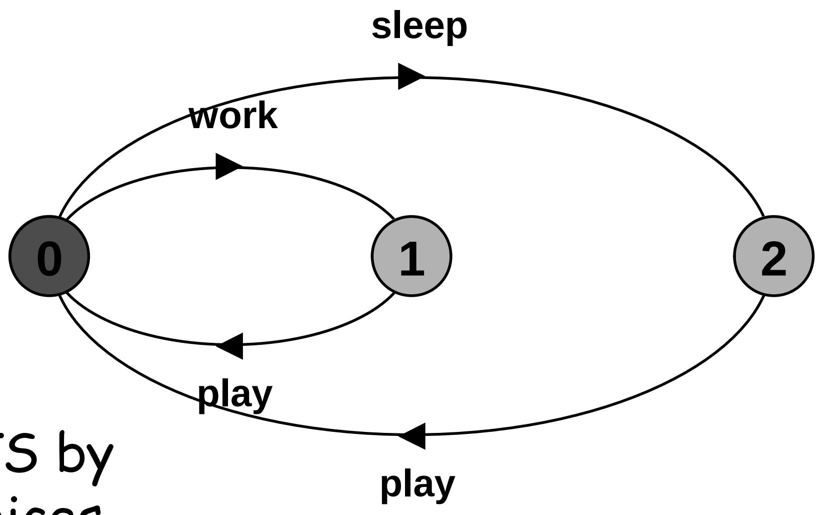
`||C = (P||Q)<<{a1,…,an}` specifies a composition in which the actions **`a1,..,an`** have higher priority than any other action in the alphabet of **`P||Q`** including the silent action **`tau`**. *In any choice in this system which has one or more of the actions* **`a1,..,an`** *labeling a transition, the transitions labeled with lower priority actions are discarded.*

Low Priority (">>")

`||C = (P||Q)>>{a1,…,an}` specifies a composition in which the actions **`a1,..,an`** have lower priority than any other action in the alphabet of **`P||Q`** including the silent action **`tau`**. *In any choice in this system which has one or more transitions not labeled by* **`a1,..,an`**, *the transitions labeled by* **`a1,..,an`** *are discarded.*

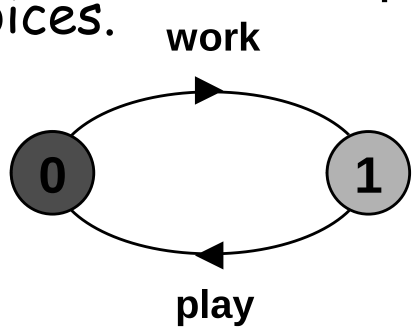
Progress - action priority

```
NORMAL =(work->play->NORMAL
        |sleep->play->NORMAL).
```

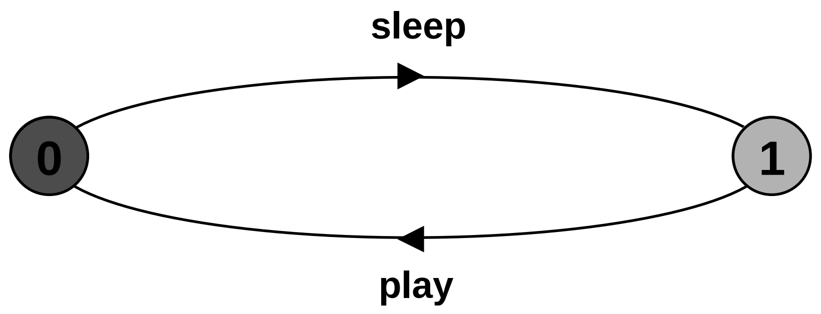


Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

```
||HIGH =(NORMAL)<<{work}.
```



```
||LOW  =(NORMAL)>>{work}.
```



7.4 Congested Single Lane Bridge

```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

BLUECROSS - eventually one of the blue cars will be able to enter

REDCROSS - eventually one of the red cars will be able to enter

Congestion using action priority?

Could give red cars priority over blue (or vice versa) ?
In practice neither has priority over the other.

Instead we merely encourage congestion by *lowering the priority of the exit actions of both cars from the bridge.*

```
||CongestedBridge = (SingleLaneBridge)
                    >>{red[ID].exit,blue[ID].exit}.
```

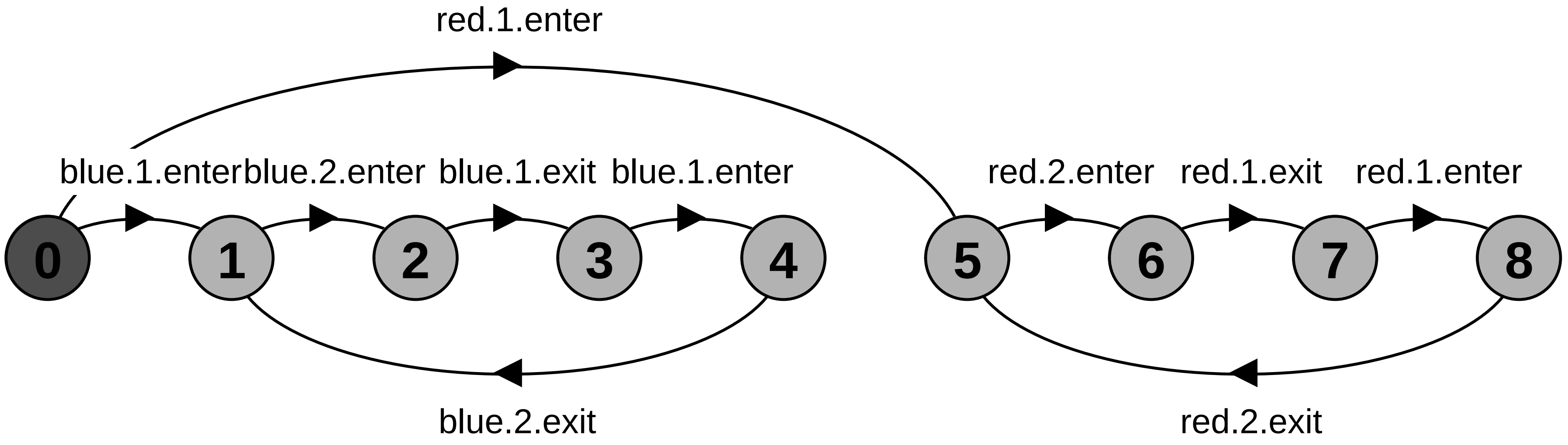
Congested Single Lane Bridge Model

```
Progress violation: BLUECROSS
Path to terminal set of states:
     red.1.enter
     red.2.enter
Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter,
red.2.exit, red.3.enter, red.3.exit}

Progress violation: REDCROSS
Path to terminal set of states:
     blue.1.enter
     blue.2.enter
Actions in terminal set:
{blue.1.enter, blue.1.exit, blue.2.enter,
blue.2.exit, blue.3.enter, blue.3.exit}
```

This corresponds with the observation that, with ***more than one car***, it is possible that whichever color car enters the bridge first will continuously occupy the bridge preventing the other color from ever crossing.

Congested Single Lane Bridge Model

```
||CongestedBridge = (SingleLaneBridge)
                   >>{red[ID].exit,blue[ID].exit}.
```



Will the results be the same if we model congestion by giving car ***entry*** *to the bridge* ***high*** *priority?*

Can congestion occur if there is only one car moving in each direction?

Progress - Revised Single Lane Bridge Model

The bridge needs to know whether or not cars are **waiting** to cross.

Modify **CAR**:

```
CAR = (request->enter->exit->CAR).
```

Modify **BRIDGE**:

Red cars are only allowed to enter the bridge if there are no blue cars on the bridge **and** there are ***no blue cars waiting*** to enter the bridge.

Blue cars are only allowed to enter the bridge if there are no red cars on the bridge **and** there are ***no red cars waiting*** to enter the bridge.

Progress - Revised Single Lane Bridge Model

```
// nr: #red cars on br.;  wr: #red cars waiting to enter
// nb: #blue cars on br.; wb: #blue cars waiting to enter

BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] = (
     red[ID].request          -> BRIDGE[nr][nb][wr+1][wb]
    |when (nb==0 && wb==0)
             red[ID].enter    -> BRIDGE[nr+1][nb][wr-1][wb]
    |red[ID].exit             -> BRIDGE[nr-1][nb][wr][wb]
    |blue[ID].request         -> BRIDGE[nr][nb][wr][wb+1]
    |when (nr==0 && wr==0)
             blue[ID].enter   -> BRIDGE[nr][nb+1][wr][wb-1]
    |blue[ID].exit            -> BRIDGE[nr][nb-1][wr][wb]
).
```

OK now?

Progress - Analysis of Revised Single Lane Bridge Model

```
Trace to DEADLOCK:
   red.1.request
   red.2.request
   red.3.request
   blue.1.request
   blue.2.request
   blue.3.request
```

The trace is the scenario in which there are cars waiting at both ends, and consequently, the bridge does not allow either red or blue cars to enter.

Solution?

Introduce some asymmetry in the problem (cf. Dining philosophers).

This takes the form of a boolean variable (**bt**) which breaks the deadlock by indicating whether it is the turn of blue cars or red cars to enter the bridge.

Arbitrarily initialize **bt** to true initially giving blue initial precedence.

Progress - 2nd Revision of Single Lane Bridge Model

```
const True = 1   const False = 0   range B = False..True

    //  bt: true ~ blue turn; false ~ red turn

BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] = (
   red[ID].request      -> BRIDGE[nr][nb][wr+1][wb][bt]
  |when (nb==0 && (wb==0||!bt))
        red[ID].enter   -> BRIDGE[nr+1][nb][wr-1][wb][bt]
  |red[ID].exit         -> BRIDGE[nr-1][nb][wr][wb][True]
  |blue[ID].request     -> BRIDGE[nr][nb][wr][wb+1][bt]
  |when (nr==0 && (wr==0||bt))
        blue[ID].enter  -> BRIDGE[nr][nb+1][wr][wb-1][bt]
  |blue[ID].exit        -> BRIDGE[nr][nb-1][wr][wb][False]
).
```

Analysis ?

Revised Single Lane Bridge Implementation - FairBridge

```
class FairBridge extends Bridge {
    protected int nred, nblue, wblue, wred;
    protected boolean blueturn = true;

    synchronized void request() {
        ++wred;
    }

    synchronized void redEnter() throws Int’Exc’ {
        while (!(nblue==0 && (waitblue==0 || !blueturn)))
             wait();
        --wred;
        ++nred;
    }
}
```

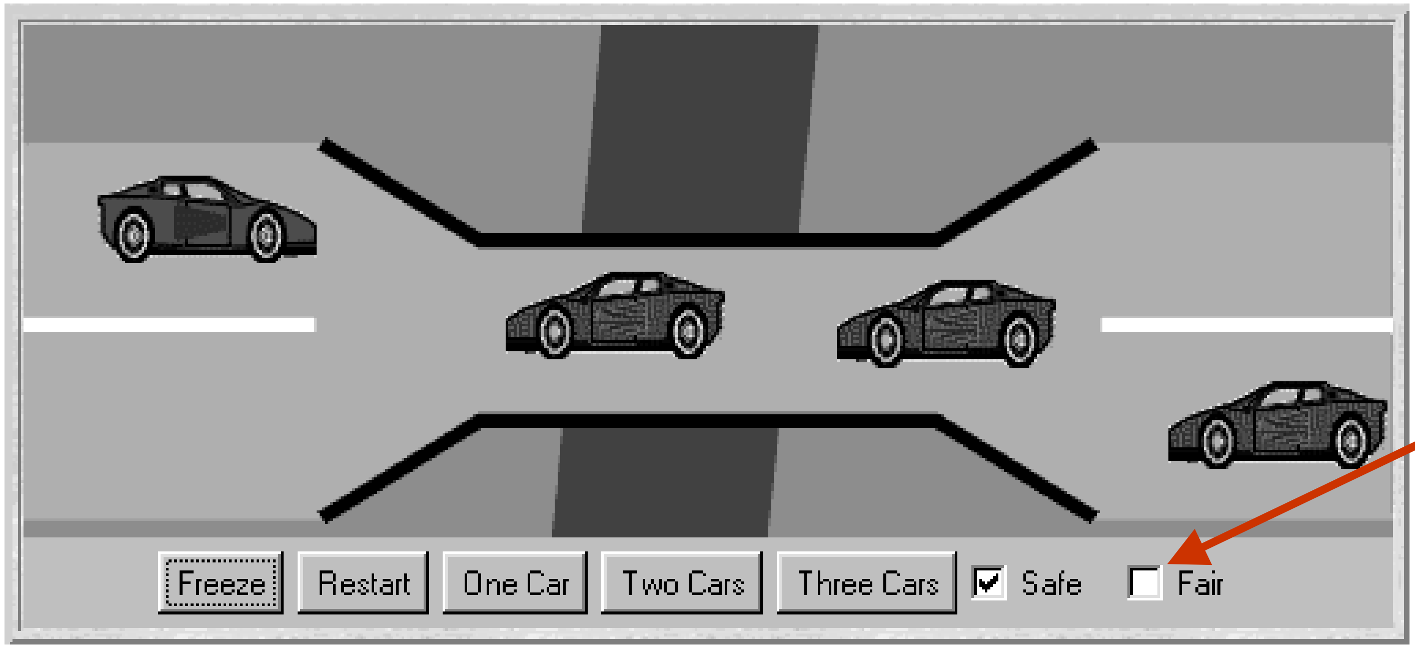
This is a direct translation from the model.

Revised Single Lane Bridge Implementation - FairBridge

```
class FairBridge extends Bridge {

    …

    synchronized void redExit(){
        --nred;
        blueturn = true;
        if (nred==0) notifyAll();
    }
}
```

This is a direct translation from the model.

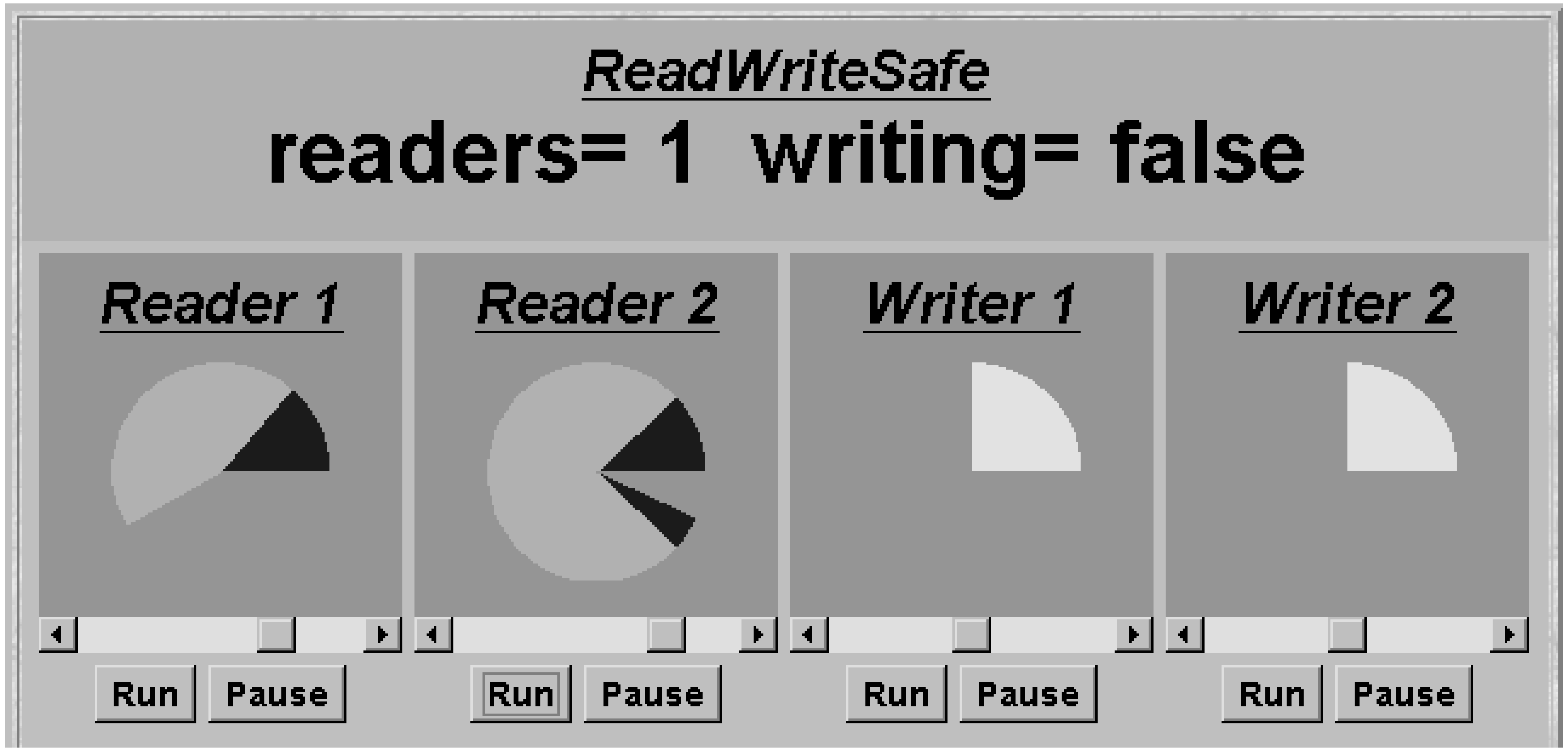
Revised single lane bridge implementation - FairBridge



Use **FairBridge** monitor

Note: we did not need to introduce a new *request* monitor method. The existing enter methods can be modified to increment a wait count before testing whether or not the caller can access the bridge.

7.5 Readers and Writers



Light blue indicates database access.

A shared database is accessed by two kinds of processes. **Readers** execute transactions that examine the database while **Writers** both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

Readers and Writers Model

- Events or actions of interest?

  acquireRead, releaseRead, acquireWrite, releaseWrite

- Identify processes.

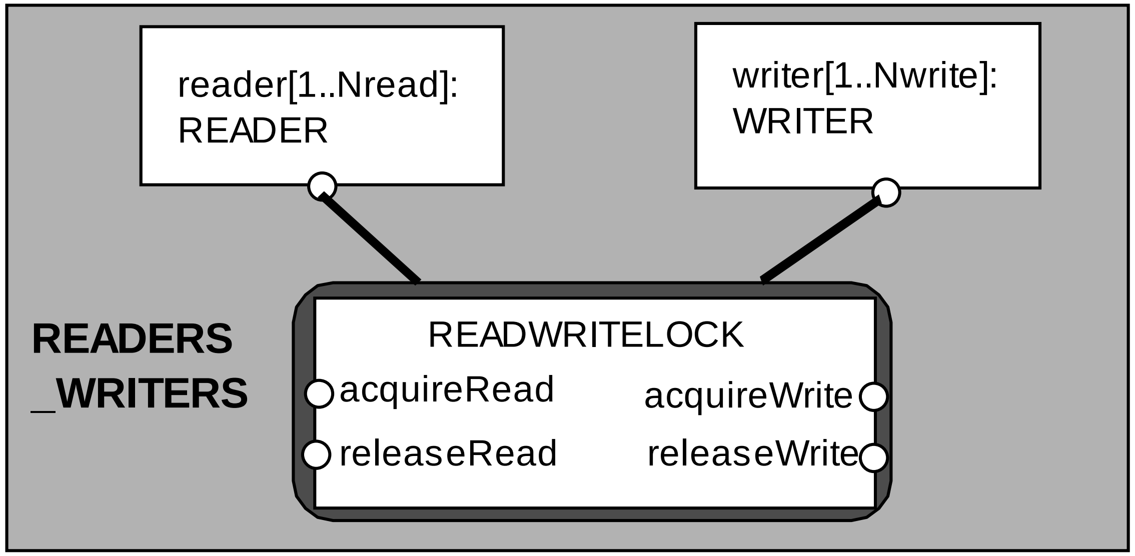
  Readers, Writers & the RW_Lock

- Identify properties.

  RW_Safe

  RW_Progress

- Structure diagram:



Readers/Writers Model - READER & WRITER

```
set Actions =
 {acquireRead,releaseRead,acquireWrite,releaseWrite}

READER = (acquireRead->examine->releaseRead->READER)
  + Actions
  \ {examine}.

WRITER = (acquireWrite->modify->releaseWrite->WRITER)
  + Actions
  \ {modify}.
```

Alphabet extension is used to ensure that the other access actions cannot occur freely for any prefixed instance of the process (as before).

Action hiding is used as actions **examine** and **modify** are not relevant for access synchronisation.

Readers/Writers Model - RW_LOCK

```
const False = 0   const True = 1   range Bool = False..True
const Nread = 2   // #readers
const Nwrite= 2   // #writers

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] = (
     when (!writing) acquireRead  -> RW[readers+1][writing]
    |releaseRead                  -> RW[readers-1][writing]
    |when (readers==0 && !writing)
                    acquireWrite -> RW[readers][True]
    |releaseWrite                 -> RW[readers][False]
).
```

The lock maintains a count of the number of readers, and a boolean for the writers.

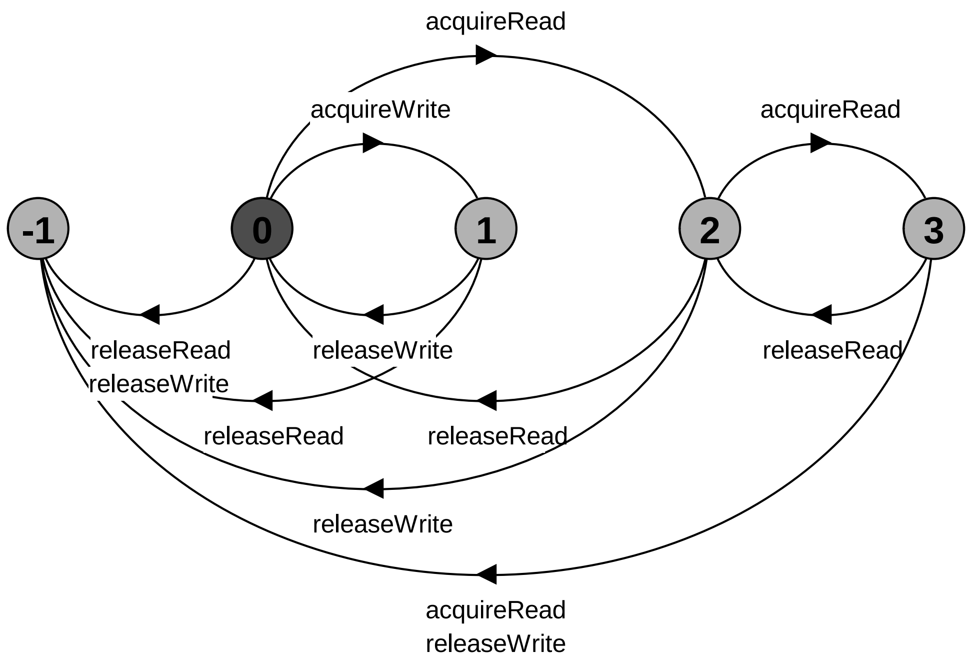
Readers/Writers Model - Safety

```
property SAFE_RW =
    (acquireRead                 -> READING[1]
    |acquireWrite                -> WRITING),

READING[i:1..Nread] =
    (acquireRead                 -> READING[i+1]
    |when (i>1)  releaseRead -> READING[i-1]
    |when (i==1) releaseRead -> SAFE_RW
    ),

WRITING = (releaseWrite      -> SAFE_RW).
```

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```

We can check that RW_LOCK satisfies the safety property……

Readers/Writers Model



An **ERROR** occurs if a reader or writer is badly behaved (**`release`** before **`acquire`** or more than two readers).

We can now compose the **READWRITELOCK** with **READER** and **WRITER** processes according to our structure…

```
||READERS_WRITERS
   = (reader[1..Nread]:READER
     || writer[1..Nwrite]:WRITER
     || {reader[1..Nread],
         writer[1..Nwrite]}::READWRITELOCK).
```

Safety and Progress Analysis ?

Readers/Writers Model - Progress

```
progress WRITE = {writer[1..Nwrite].acquireWrite}
progress READ  = {reader[1..Nread].acquireRead}
```

WRITE - eventually one of the writers will acquireWrite

READ - eventually one of the readers will acquireRead

Action priority (to simulate intensive use)?

we lower the priority of the release actions for both readers and writers.

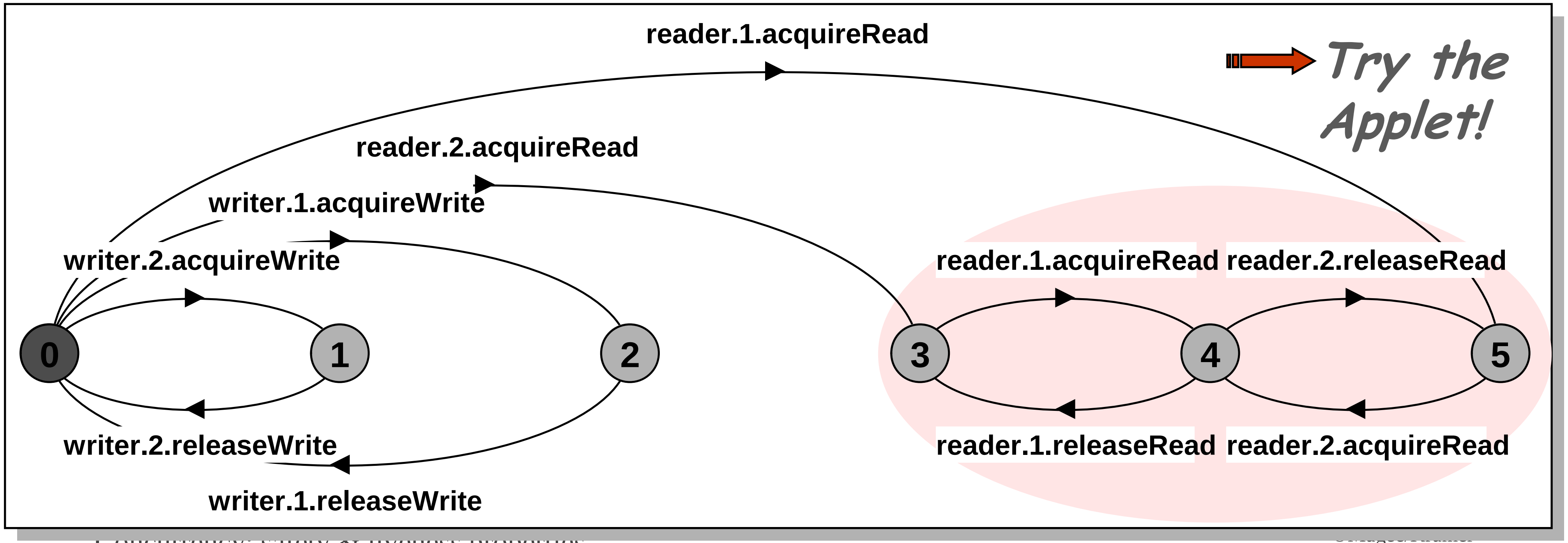
```
||RW_PROGRESS = READERS_WRITERS
                >>{reader[1..Nread].releaseRead,
                   writer[1..Nread].releaseWrite}.
```

Progress Analysis ? LTS?

Readers/Writers Model - Progress

```
Progress violation: WRITE
Path to terminal set of states:
     reader.1.acquireRead
Actions in terminal set:
{reader.1.acquireRead, reader.1.releaseRead,
 reader.2.acquireRead, reader.2.releaseRead}
```

Writer starvation: The number of readers never drops to zero.



Readers/Writers Implementation - Monitor Interface

We focus on the monitor implementation:

```
interface ReadWrite {
    void acquireRead() throws Int'Exc';
    void releaseRead();
    void acquireWrite() throws Int'Exc';
    void releaseWrite();
}
```

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface.

Firstly, the safe **READWRITELOCK.**

Readers/Writers Implementation - `ReadWriteSafe`

```
class ReadWriteSafe implements ReadWrite {
    protected int readers = 0;
    protected boolean writing = false;

    synchronized void acquireRead() throws Int’Exc’ {
        while (writing) wait();
        ++readers;
    }

    synchronized void releaseRead() {
        --readers;
        if(readers==0) notify();
    }
}
```

Unblock a single writer *when no more readers.*

Readers/Writers Implementation - ReadWriteSafe

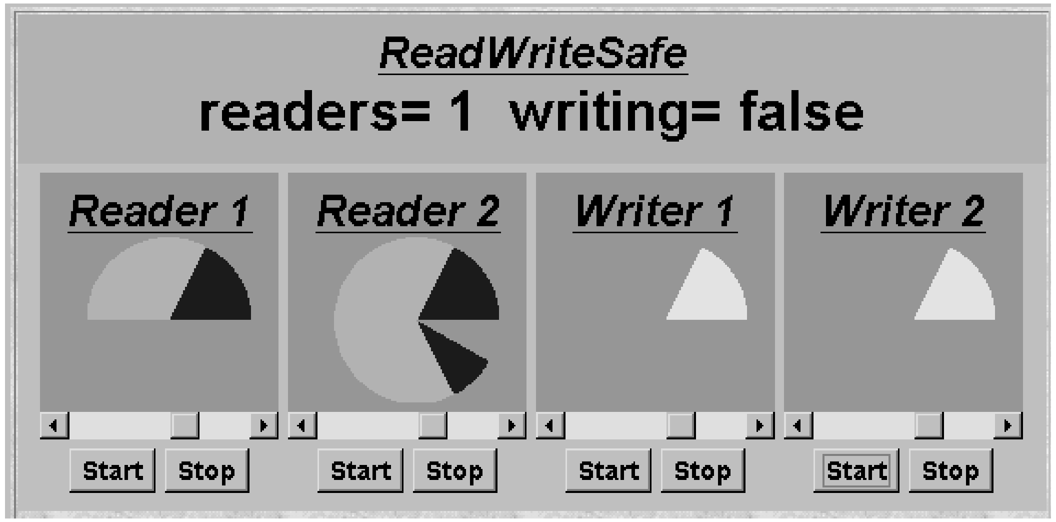
```
synchronized void acquireWrite() throws Int'Exc' {
    while (readers>0 || writing) wait();
    writing = true;
}

synchronized void releaseWrite() {
    writing = false;
    notifyAll();
}
```

Unblock all readers

However, this monitor implementation suffers from the WRITE progress problem: possible ***writer starvation*** if the number of readers never drops to zero.

Solution?

Readers/Writers - Writer Priority



Strategy: **Block** readers if there is a writer *waiting*.

```
set Actions = {acquireRead,releaseRead,acquireWrite,
               releaseWrite,requestWrite}
WRITER = ( requestWrite ->
             acquireWrite ->
               modify ->
                 releaseWrite -> WRITER)+Actions
           \{modify}.
```

Readers/Writers Model - Writer Priority

```
RW_LOCK = RW[0][False][0],
RW[readers:0..Nread][writing:Bool][waitingW:0..Nwrite] = (

   when (!writing && waitingW==0)
            acquireRead -> RW[readers+1][writing][waitingW]
  |releaseRead          -> RW[readers-1][writing][waitingW]

  |when (readers==0 && !writing)
           acquireWrite -> RW[readers][True][waitingW-1]
  |releaseWrite         -> RW[readers][False][waitingW]
  |requestWrite         -> RW[readers][writing][waitingW+1]
).
```

```
|| RW_P = R_W >>{*.release*}.  // simulate Intensive usage
```

Safety and Progress Analysis ?

Readers/Writers Model - Writer Priority

property RW_SAFE:

```
No deadlocks/errors
```

progress READ **and** WRITE:

```
Progress violation: READ
Path to terminal set of states:
     writer.1.requestWrite
     writer.2.requestWrite
Actions in terminal set:
{writer.1.requestWrite, writer.1.acquireWrite,
 writer.1.releaseWrite, writer.2.requestWrite,
 writer.2.acquireWrite, writer.2.releaseWrite}
```

Reader starvation: *if* always a writer waiting.

In practice: this may be satisfactory as is usually more read access than write, and readers generally want the most up to date information.

Readers/Writers Implementation - ReadWritePriority

```
class ReadWritePriority implements ReadWrite {
    protected int readers = 0;
    protected boolean writing = false;
    protected int waitingW = 0; // #waiting writers

    synchronized void acquireRead() throws Int’Exc’ {
        while (writing || waitingW>0) wait();
         ++readers;
    }

    synchronized void releaseRead() {
        --readers;
        if (readers==0) notify();
    }
}
```

Readers/Writers Implementation - ReadWritePriority

```
synchronized void acquireWrite() throws Int’Exc’ {
    ++waitingW;
    while (readers>0 || writing) wait();
    --waitingW;
    writing = true;
}

synchronized void releaseWrite() {
    writing = false;
    notifyAll();
}
```

Both **READ** and **WRITE** progress properties can be satisfied by introducing a **turn** variable as in the Single Lane Bridge.

Summary

- Concepts
  - **properties:** true for every possible execution
  - **safety:** nothing bad happens
  - **liveness:** something good *eventually* happens
- Models
  - **safety:** no reachable ERROR/STOP state
    compose safety properties at appropriate stages
  - **progress:** an action is eventually executed
    fair choice and action priority
    apply progress check on the final target system model
- Practice
  - threads and monitors

Aim: property satisfaction