Safety & Liveness Properties

INF2140 Parallel Programming: Lecture 7

March 14, 2012

safety & liveness properties

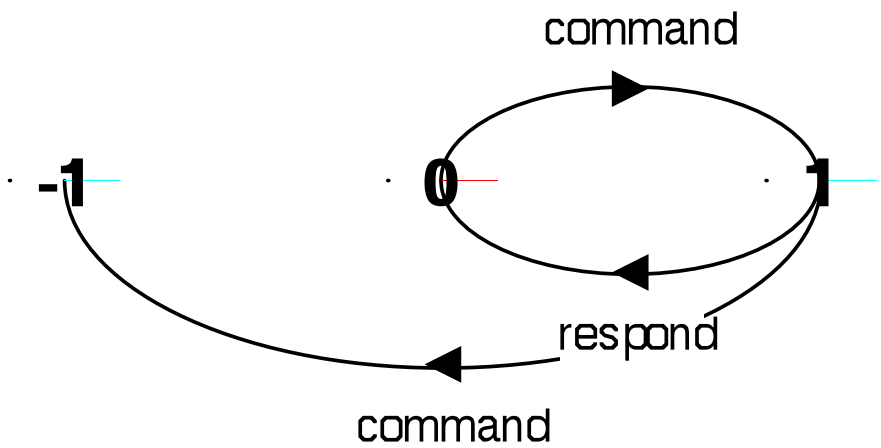
- Concepts
  - **properties:** true for every possible execution
  - **safety:** nothing bad happens
  - **liveness:** something good eventually happens
- Models
  - **safety:** no reachable `ERROR/STOP` state
  - **progress:** an action is eventually executed
    fair choice and action priority
- Practice
  - **threads and monitors**

Aim: property satisfaction

Safety

A safety property asserts that nothing bad happens

- STOP or deadlocked state (no outgoing transitions)
- ERROR process (-1) to detect erroneous behaviour



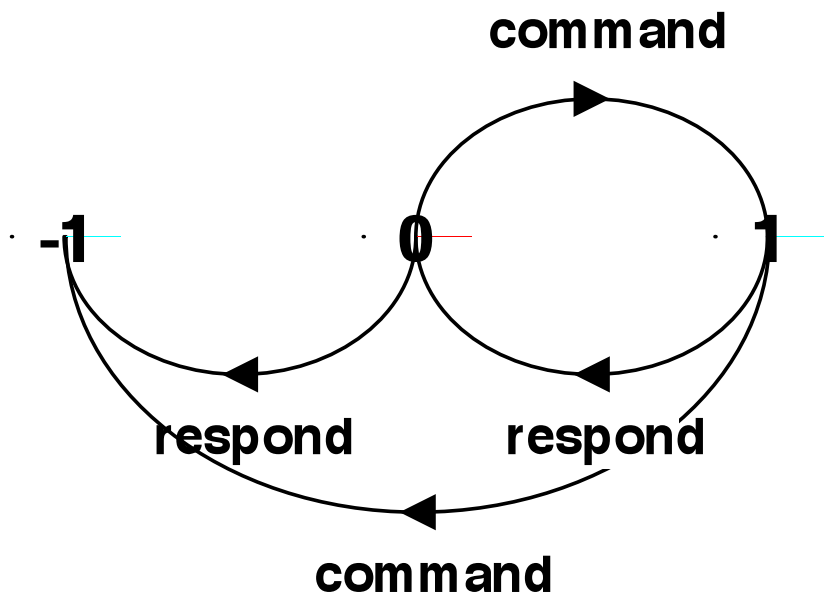

```
ACTUATOR
    =(command->ACTION),
ACTION
    =(respond->ACTUATOR
     |command->ERROR).
```

- analysis using LTSA: (shortest trace)

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

Safety - property specification

- ERROR conditions state what is not required (cf. 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
    ).
```

- analysis using LTSA as before.

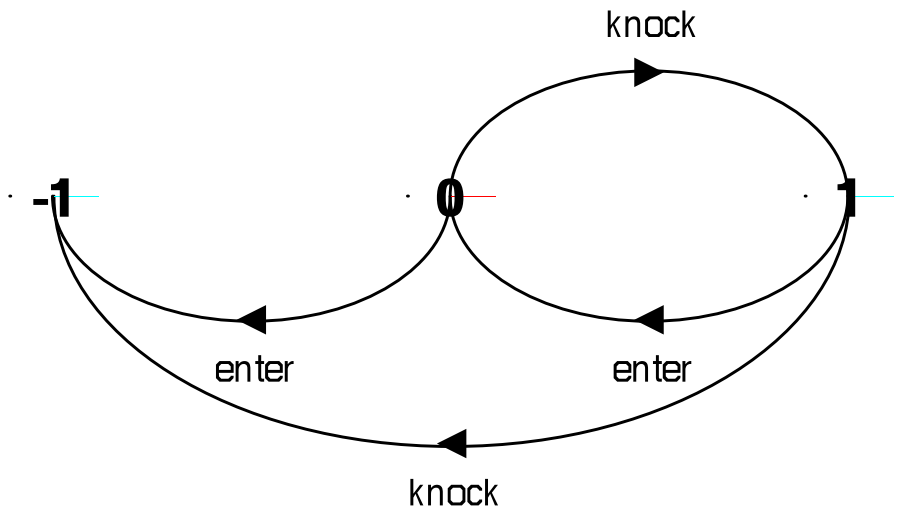
Safety properties

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

Traces: `knock->enter` `enter` `knock->knock`

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

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 of S ∩ alphabet of 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, composing a property with a set of processes does not affect their ***correct*** *behavior. However, if a behavior can occur which violates the safety property, then* `ERROR` *is reachable. Properties must be deterministic to be transparent.*

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 - mutual exclusion

```
LOOP = (mutex.down -> enter -> exit -> mutex.up -> LOOP).
||SEMADEMO = (p[1..3]:LOOP
              ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

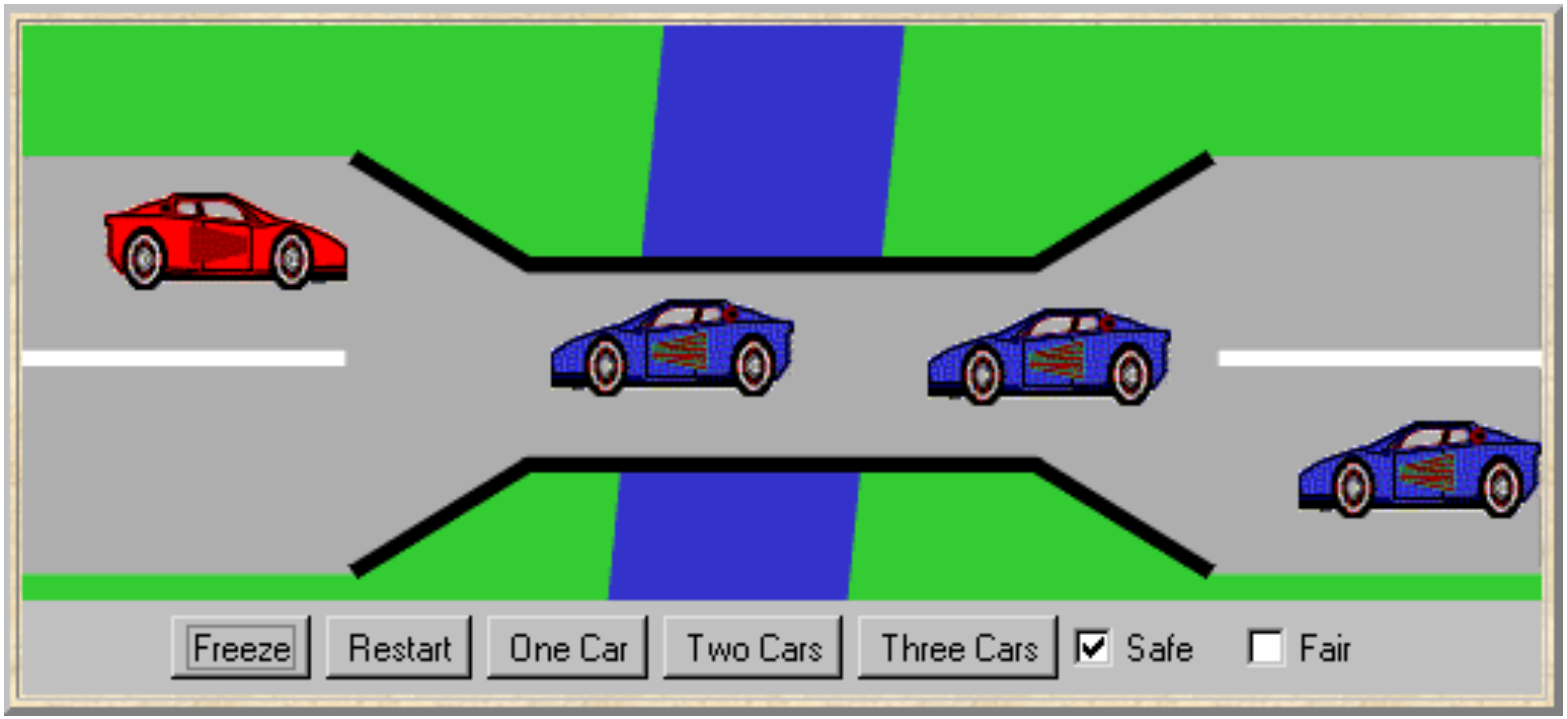
How do we check that this does indeed ensure mutual exclusion in the critical section?

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

Check safety using LTSA.

What happens if semaphore is initialized to 2?

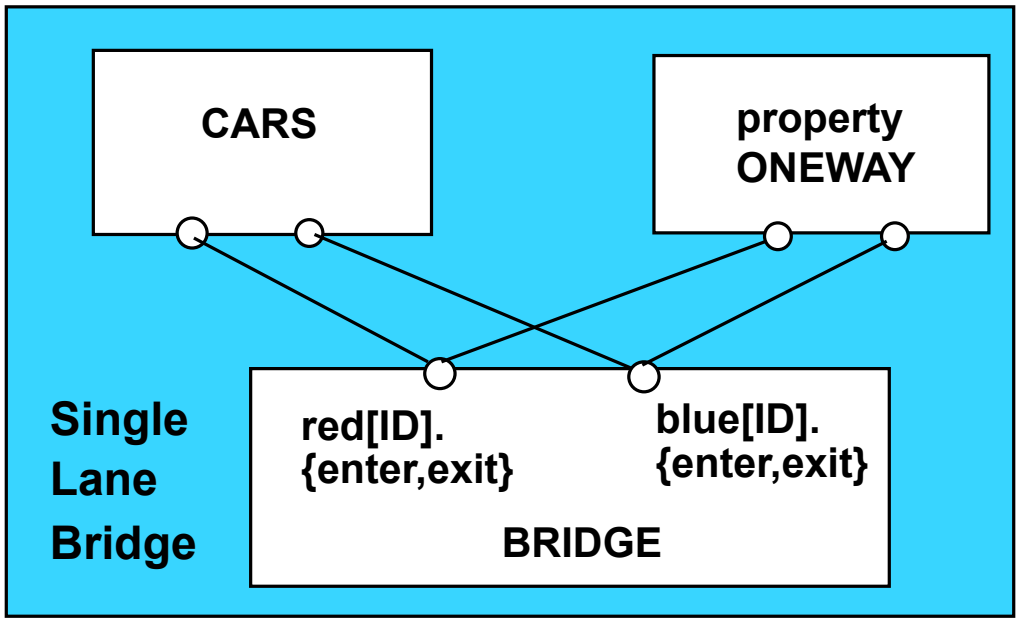
Single Lane Bridge problem



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`
- Define each process and interactions (structure).



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).
```

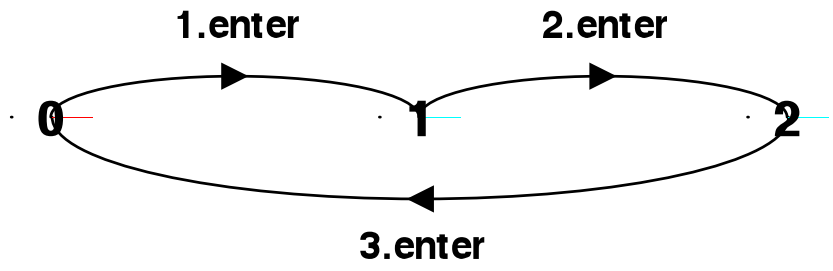
To model the fact that cars *cannot pass each other* on the bridge, we model a CONVOY of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

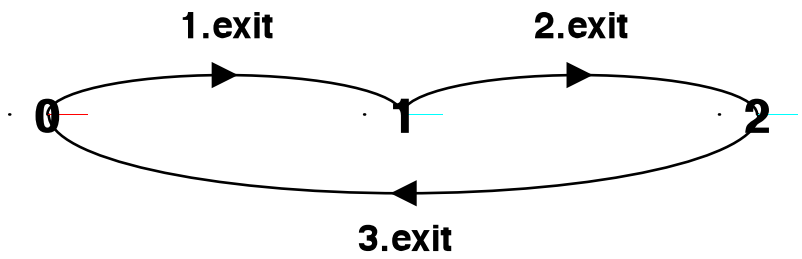
```
||CARS = (red:CONVOY || blue:CONVOY).
```

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 the bridge only if they drive in the same direction. The bridge maintains counts of blue and red cars on the bridge. Red cars are only allowed to enter when the blue count is zero and vice-versa.

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

Even when 0, exit actions permit the car counts to be decremented. LTSA maps these undefined states to ERROR.

Single Lane Bridge - safety property ONEWAY

We now specify a safety property to check that cars do not collide! While red cars are on the bridge only red cars can enter; similarly for blue cars. When the bridge is empty, any car may enter.

```
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]
            ),  //i is a count of red cars on the bridge

BLUE[i:ID]= (blue[ID].enter-> BLUE[i+1]
            |when(i==1)blue[ID].exit -> ONEWAY
            |when( i>1)blue[ID].exit -> BLUE[i-1]
            ).  //i is a count of blue cars on the bridge
```

Single Lane Bridge - model analysis

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

Is the safety property ONEWAY violated?

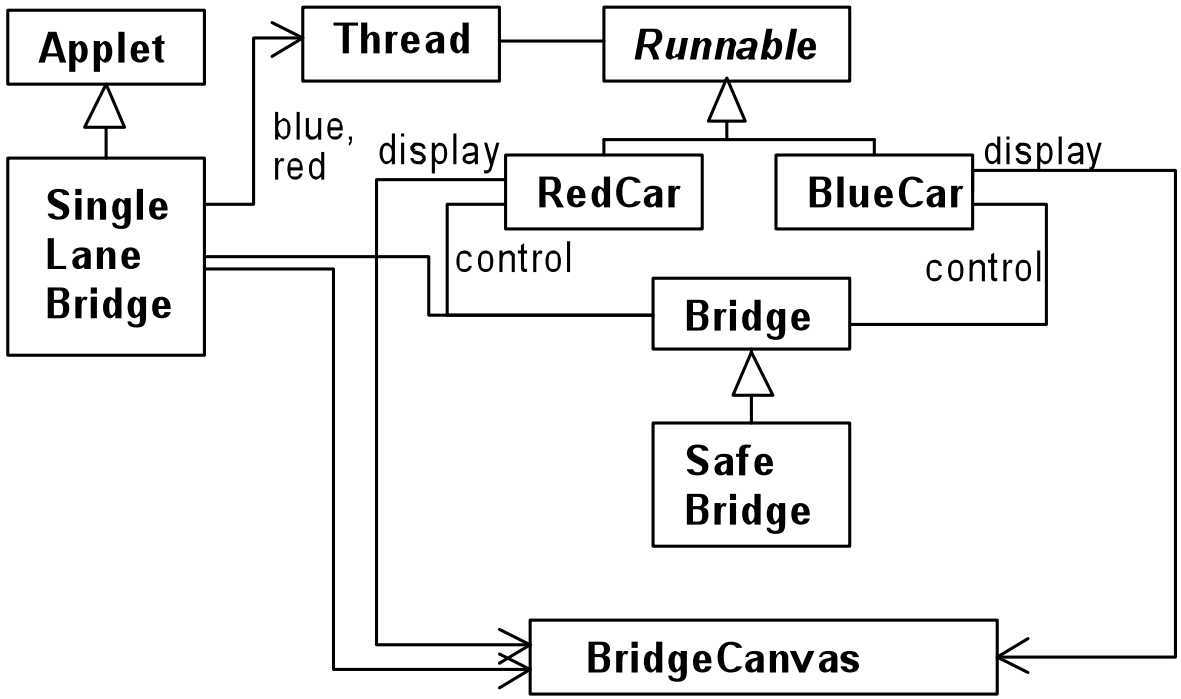
```
No deadlocks/errors
```

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

Without the BRIDGE contraints,
is the safety property ONEWAY violated?

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

Single Lane Bridge: Implementation in Java



- Active entities (cars) are implemented as threads.
- Passive entity (bridge) is implemented as a monitor.
- BridgeCanvas enforces no overtaking.

Single Lane Bridge: BridgeCanvas

A BridgeCanvas instance is created by the SingleLaneBridge applet—a ref is passed to each created RedCar and BlueCar object.

```
class BridgeCanvas extends Canvas {
  public void init(int ncars) {...} //set number of cars

  //move red car with identity i one step
  //returns true for the period on bridge, from just before
  public boolean moveRed(int i) // until just after
         throws InterruptedException{...}

  //move blue car with identity i one step
  //returns true for the period on bridge, from just before
  public boolean moveBlue(int i) // until just after
         throws InterruptedException{...}

  public synchronized void freeze(){...}// freeze display
  public synchronized void thaw(){...}  //unfreeze display
}
```

Single Lane Bridge - RedCar

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

  RedCar(Bridge b, BridgeCanvas d, int id) {
    display = d; this.id = id; control = b;
  }
  public void run() {
    try {
      while(true) {
        while (!display.moveRed(id));    // not on bridge
        control.redEnter();              // request access to bridge
        while (display.moveRed(id)); // move over bridge
        control.redExit();               // release access to bridge
      }
    } catch (InterruptedException e) {}
  }
}
                                    // Similarly for the BlueCar
```

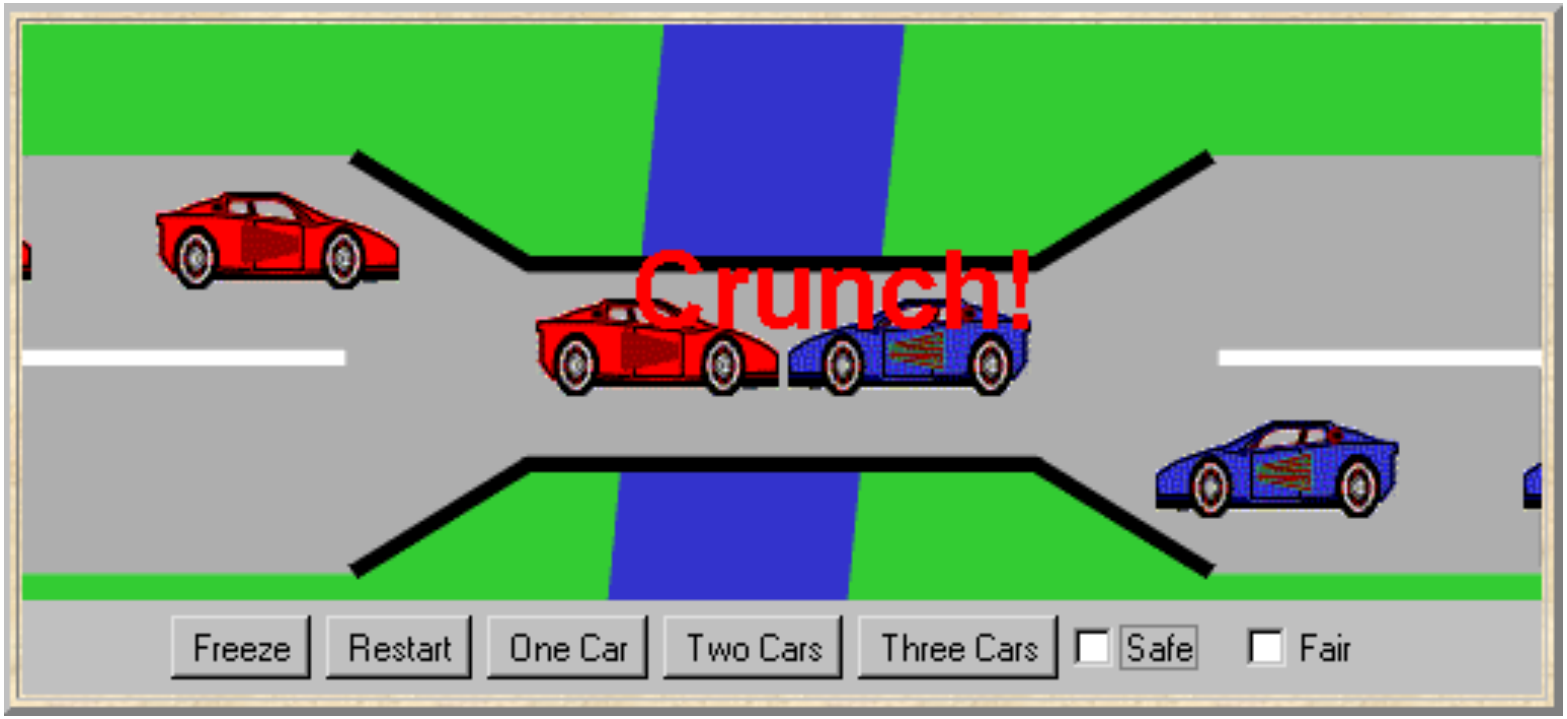
Single Lane Bridge - class Bridge

```
class Bridge {
  synchronized void redEnter()
    throws InterruptedException {}
  synchronized void redExit()  {}
  synchronized void blueEnter()
    throws InterruptedException {}
  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 {

  private int nred  = 0; //number of red cars on bridge
  private int nblue = 0; //number of blue cars on bridge

  // Monitor Invariant: nred≥0 and nblue≥0 and
  //                    not (nred>0 and nblue>0)

 synchronized void redEnter()
      throws InterruptedException {
    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 InterruptedException {
    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.

Liveness

A safety property asserts that nothing bad happens.

A liveness property asserts that something good eventually happens.

Single Lane Bridge: Does every car *eventually* get an opportunity to cross the bridge? ie. to 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.

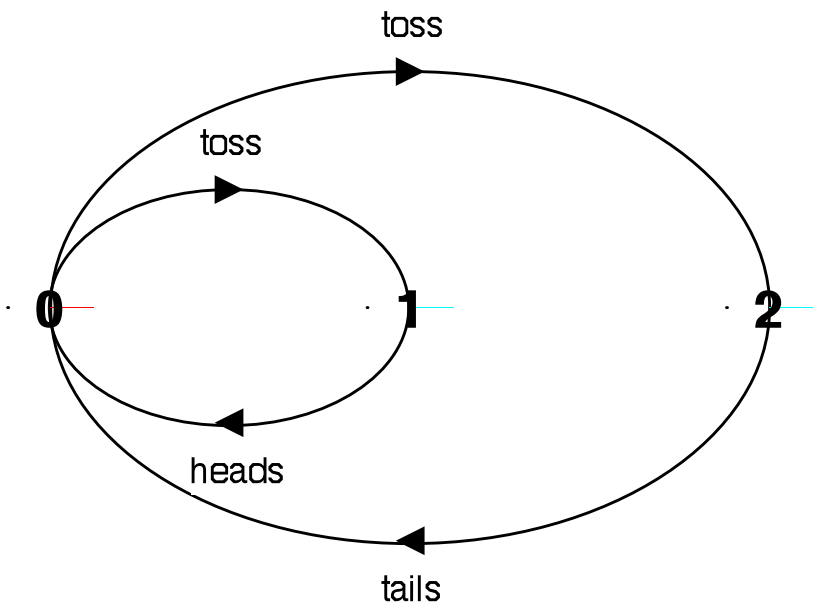
Progress properties - fair choice

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).
```

If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.

This requires **Fair Choice** !



Progress properties

`progress P = {a1,a2..an}` defines a progress property `P` which asserts that in an infinite execution of a target system, at least **one** of the actions `a1,a2..an` will be executed *infinitely often*.

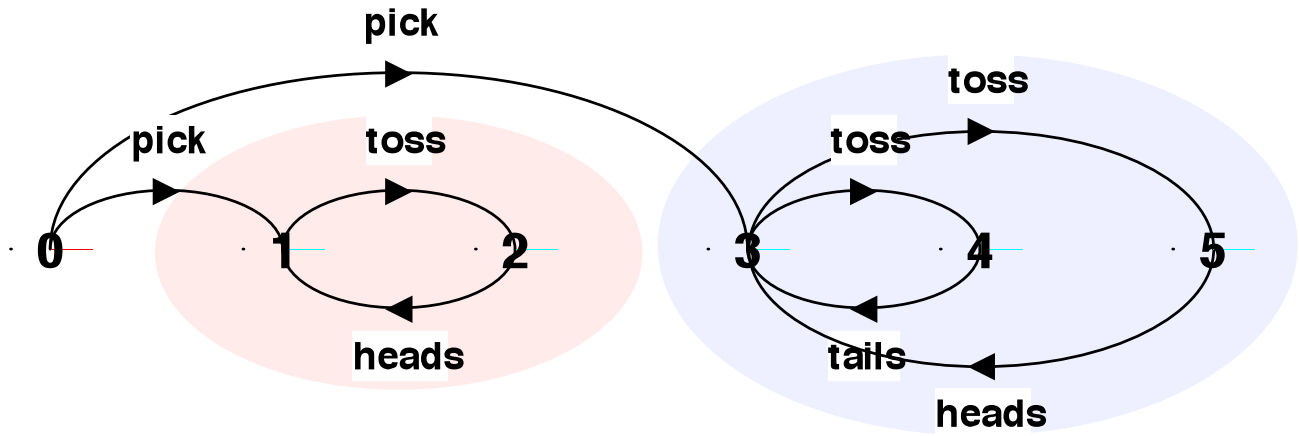
`COIN` system:
```
progress HEADS = {heads} ?
progress TAILS = {tails} ?
```

LTSA check progress: `No progress violations detected.`

Progress properties

Suppose that there were two possible coins that could be picked up:

a trick coin and
a regular coin



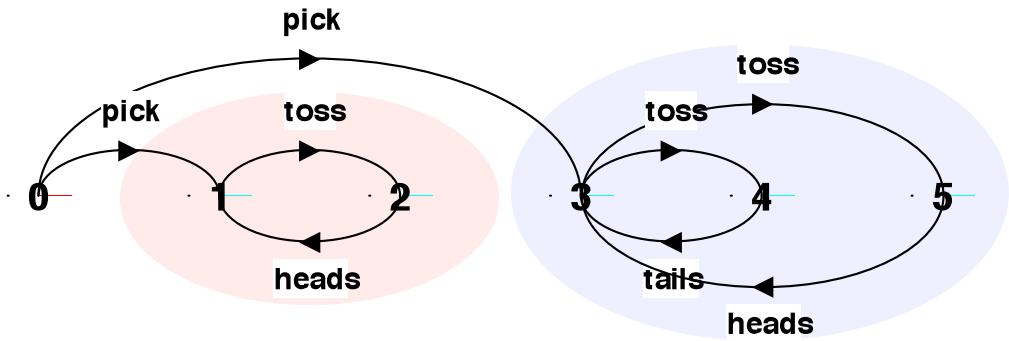

```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).
```

TWOCOIN system:

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

Progress properties

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



LTSA check progress

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

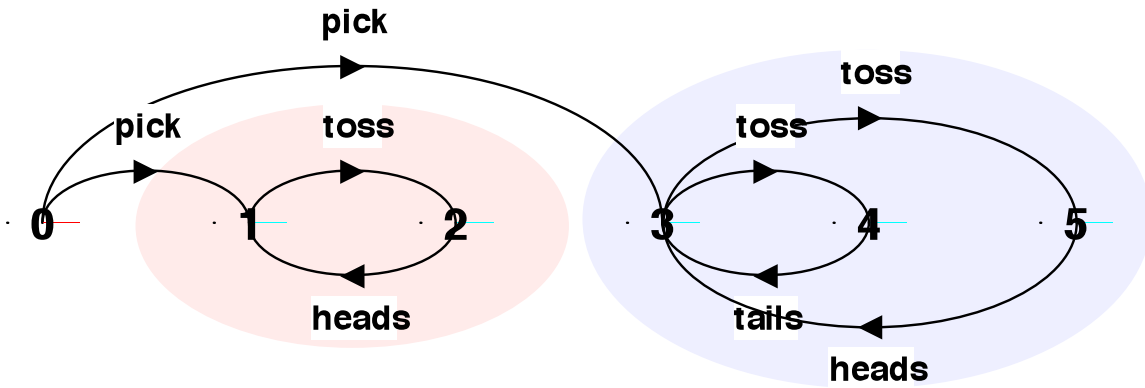
```
progress HEADSorTails = {heads,tails}
```

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 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!

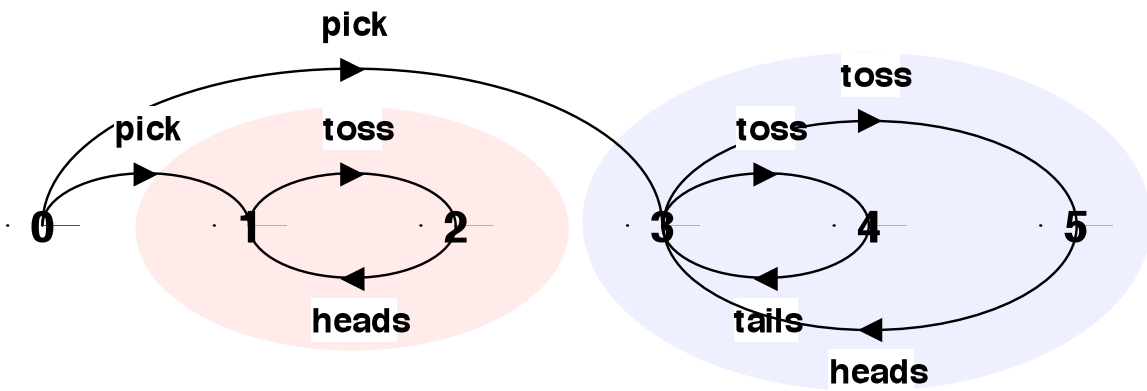
Progress analysis

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

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

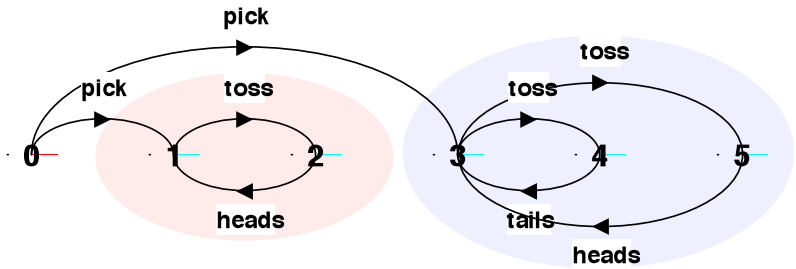
Default: given fair choice, *every* action in the alphabet of the target system will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.

Default analysis for `TWOCOIN`?



Progress analysis

Default analysis for TWOCOIN:
separate progress property
for every action.



If the default holds, then
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 violation for actions:
{pick}
Path to terminal set of states:
   pick
Actions in terminal set:
{toss, heads, tails}
```

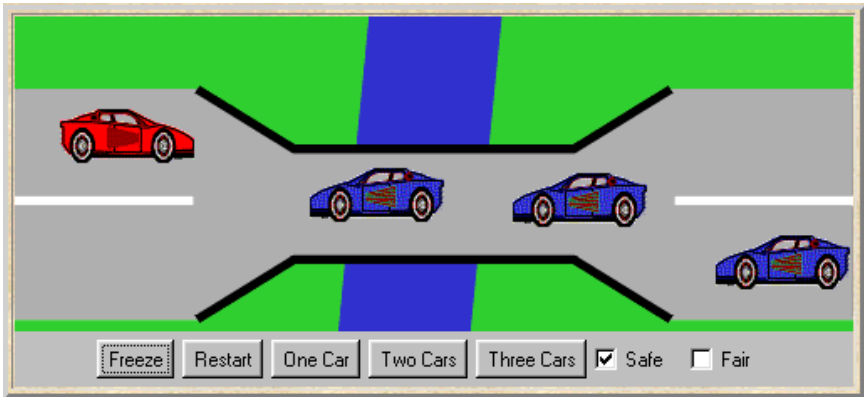
```
Progress violation for actions:
{pick, tails}
Path to terminal set of states:
   pick
Actions in terminal set:
{toss, heads}
```

Progress - single lane bridge

The Single Lane Bridge implementation can permit progress violations.

However, if default progress analysis is applied to the model then no violations are detected!

Why not?




```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
No progress violations detected.
```

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must check under adverse conditions. We 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* $a1, \ldots, an$ *labeling a transition, the transitions labeled with lower priority actions are discarded.*

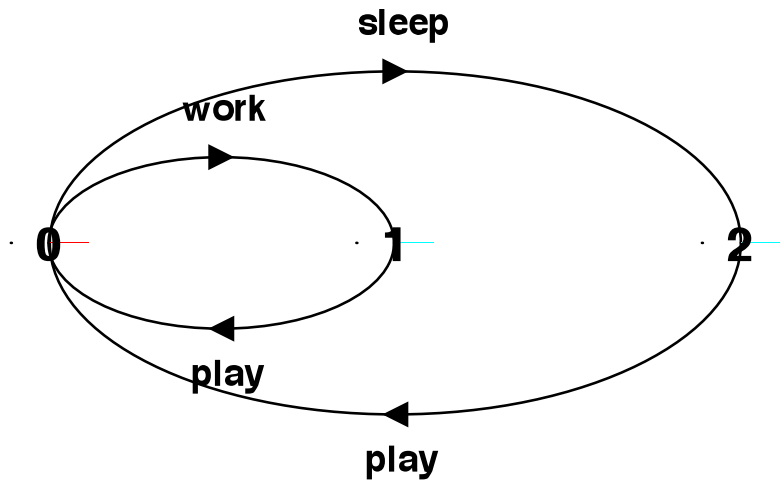
Low Priority (">>")

`||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* $a1, \ldots, an$*, the transitions labeled by* $a1, \ldots, an$ *are discarded.*

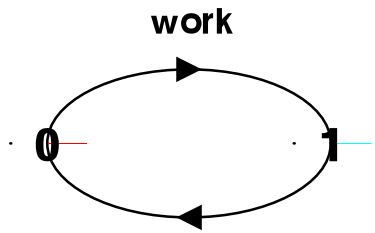
Progress - action priority

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

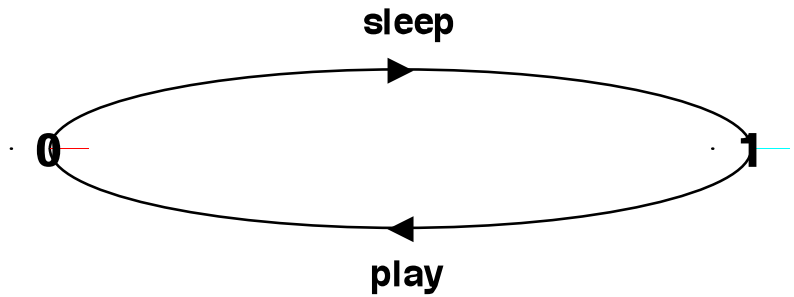
Action priority simplifies the resulting LTS by discarding lower priority actions from choices.




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




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



Congested single lane bridge

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

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

Progress Analysis ? LTS?

congested single lane bridge model

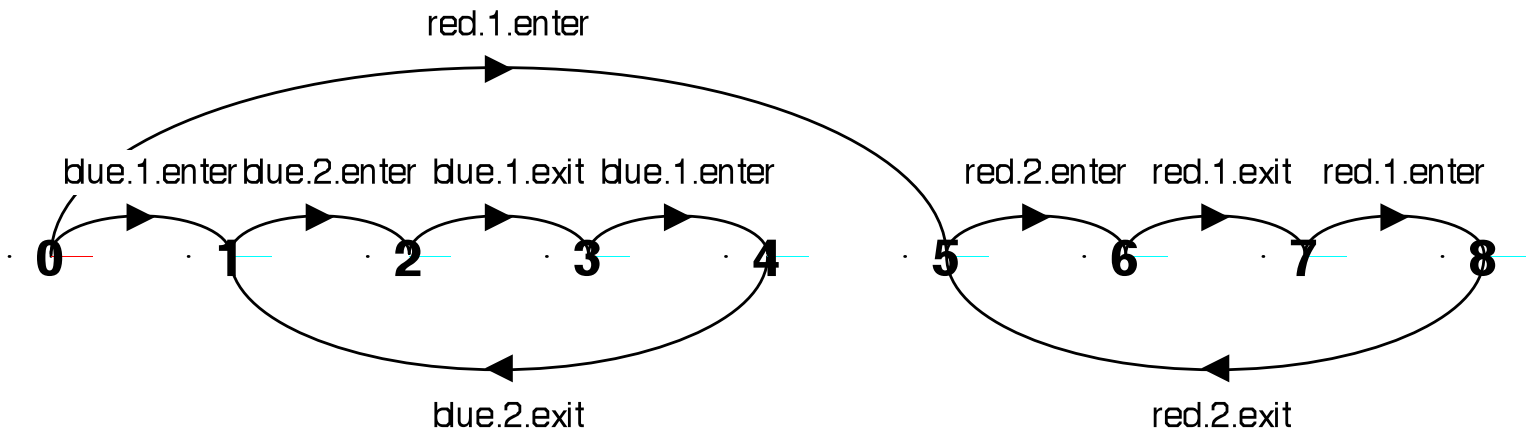
```
Progress violation: BLUECROSS
Path to terminal set of states:
     red.1.enter
     red.2.enter
Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter,
 red.2.exit, red.3.enter, red.3.exit}

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, this color 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 only one car moves 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

Revised version with 4 control variables:

nr: number of red cars on the bridge
nb: number of blue cars on the bridge
wr: number of red cars waiting to enter
wb: number of 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 **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 set `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 indicates blue turn, false indicates 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 {
  private int nred  = 0; // red cars on the bridge
  private int nblue = 0; // blue cars on the bridge
  private int waitblue = 0;  // waiting blue cars
  private int waitred = 0;   // waiting red cars
  private boolean blueturn = true;

  synchronized void redEnter()
      throws InterruptedException {
    ++waitred;
    while (nblue>0||(waitblue>0 && blueturn)) wait();
    --waitred; ++nred;
  }
  synchronized void redExit(){
    --nred; blueturn = true;
    if (nred==0)notifyAll();
  }
```

This is a direct translation from the model.

Revised single lane bridge implementation: FairBridge

```
  synchronized void blueEnter(){
      throws InterruptedException {
    ++waitblue;
    while (nred>0||(waitred>0 && !blueturn)) wait();
    --waitblue;
    ++nblue;
  }
  synchronized void blueExit(){
    --nblue;
    blueturn = false;
    if (nblue==0) notifyAll();
  }
}
```

The "fair" check box must be chosen to select the FairBridge implementation.

Note that 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.

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