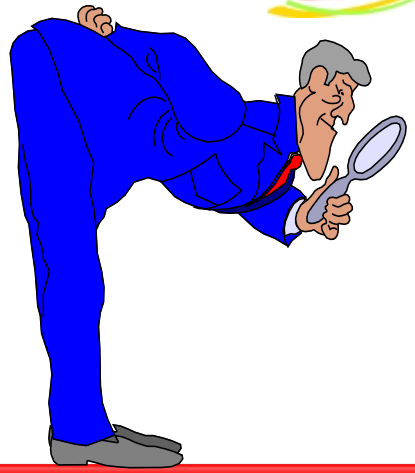
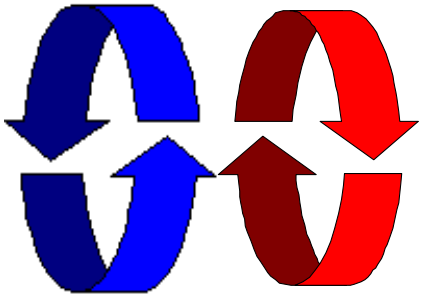
Propriété de sûreté et vivacité
Safety and liveness (part 2)

riveill@unice.fr
http://www.i3s.unice.fr/~riveill





Safety and liveness properties

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

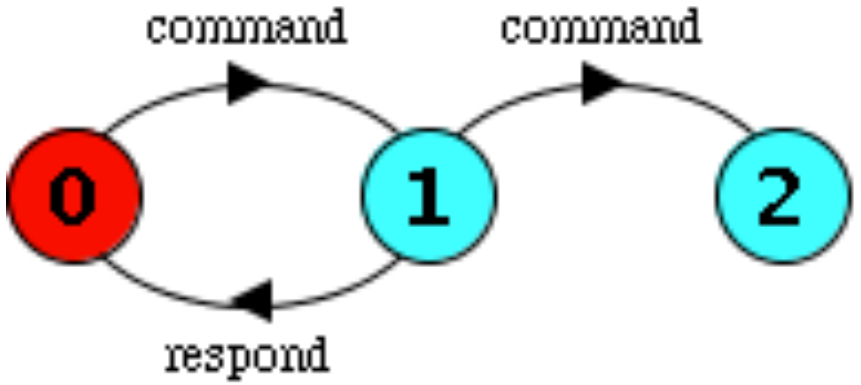
Practice: threads and monitors

Aim: property satisfaction.

Safety

STOP

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



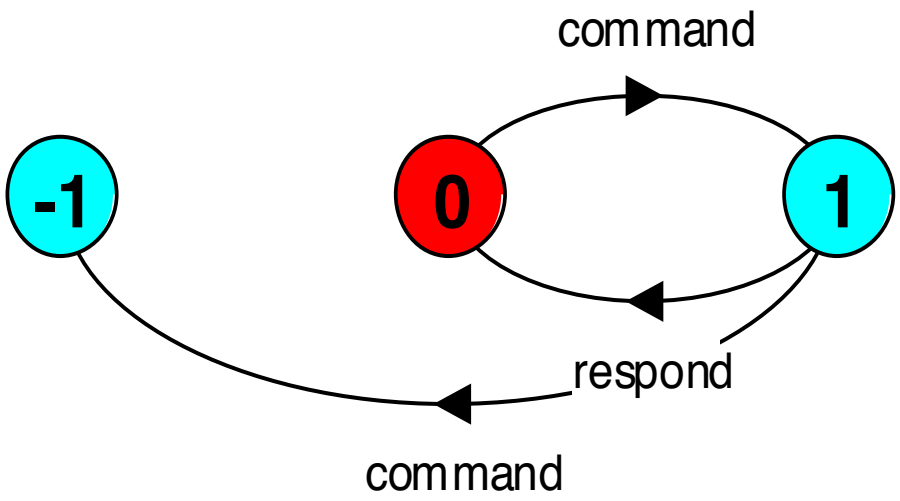
- analysis using LTSA:
  - Check -> Safety

Trace to DEADLOCK:
command
command

Give the shortest path

ERROR

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



- Analysis using LTSA:
  - Check -> Safety

Trace to ERROR:

command

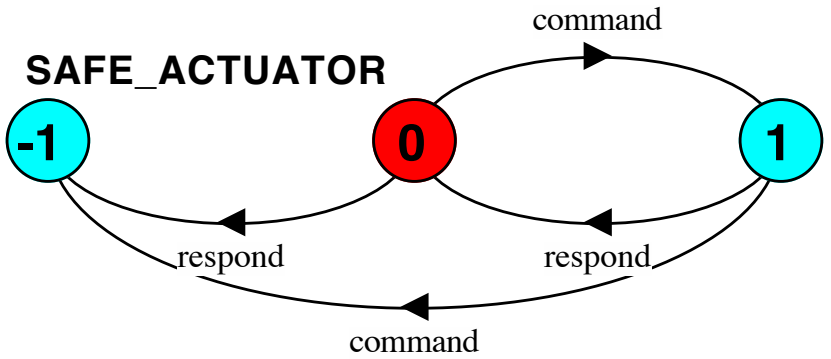
command

Give the shortest path

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
       ).
```

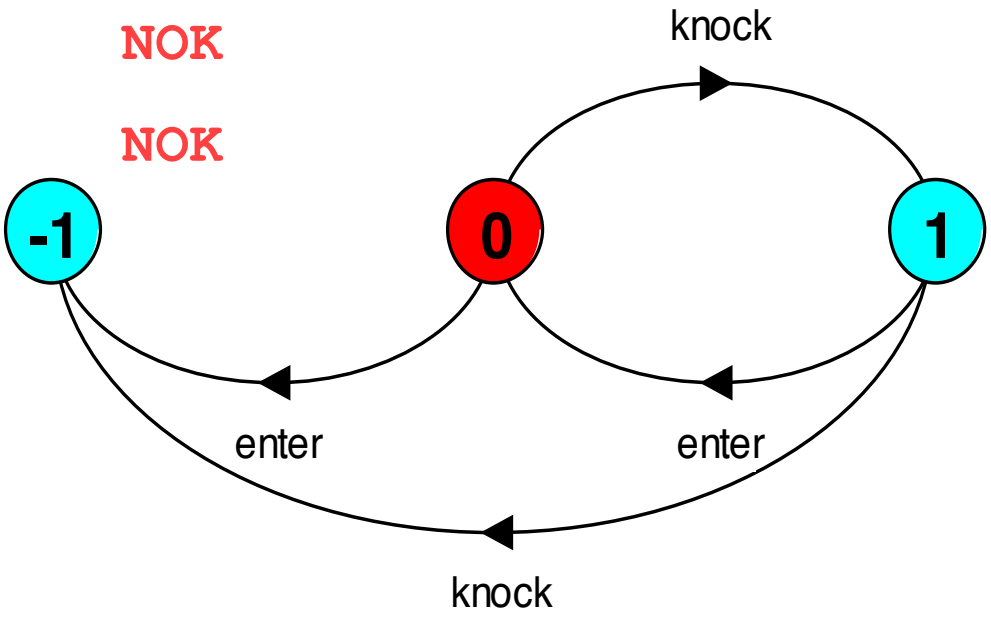


Safety properties - sample

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

Traces:

`knock→enter`	OK
`enter`	NOK
`knock→knock`	NOK



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

Safety property is a process which describe what is required.

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

Université Nice Sophia Antipolis — cnrs — i3S sophia antipolis

Safety properties

- If you want a PROC process checks the PROP property, proceed as follows

1. Describe PROC process
2. Describe PROP property
   - Safety **property PROP** defines a deterministic process that asserts that any trace including actions in the alphabet of **PROP**, is accepted by **PROP**.
3. Compose PROC process et PROP property
   - Traces of actions in the alphabet of **PROC** ∩ alphabet of **PROP** must also be valid traces of **PROP**, 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 - mutual exclusion

- How to prove that a mutex semaphore must be initialized to control a critical section

1. Describe PROC process

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

2. Describe PROP property

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

3. Compose PROC process et PROP property

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

Safety - mutual exclusion

- How to prove that a mutex semaphore must be initialized to control a critical section

4. *Verify the property*

   Check → safety

   No deadlocks/errors

- *What happens if semaphore is initialized to 2?*
  - *property MUTEX violation*

Trace to property violation in MUTEX:
p.1.mutex.down
p.1.enter
p.2.mutex.down
p.2.enter

- *What happens if semaphore is initialized to 0?*
  - *potential DEADLOCK*

Trace to DEADLOCK:

Liveness

Liveness

- A **safety** property asserts that nothing bad happens.
- A **liveness** property asserts that something good *eventually* happens.

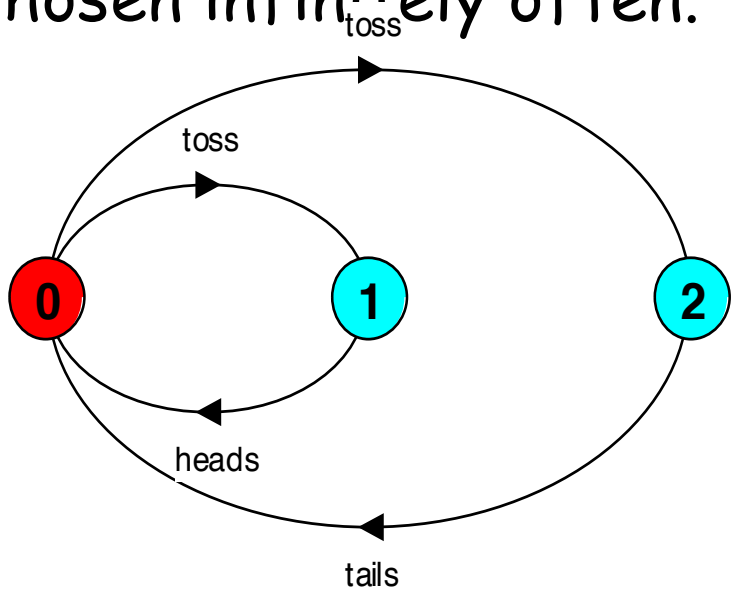
- Single Lane Bridge: *Does every car eventually get an opportunity to cross the bridge?*
  - ie. make **PROGRESS?**

- 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

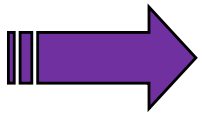
- **Fair Choice:** If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.
- If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.
- This requires **Fair Choice** !

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



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} ?`** ☑

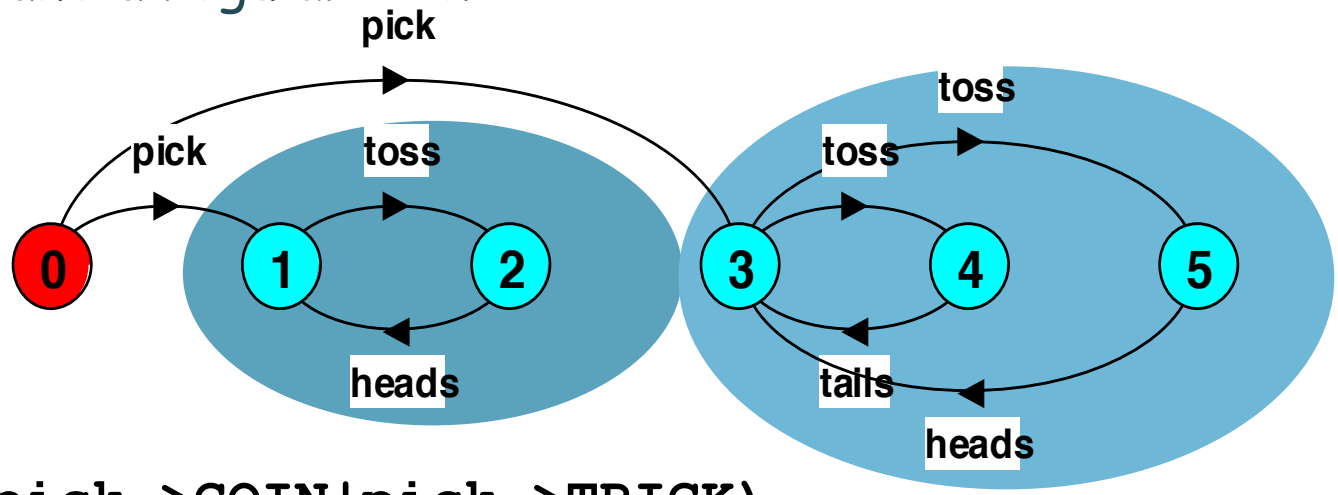
`progress TAILS = {tails} ?` ☑

LTSA check progress:

```
No progress violations
detected.
```

Progress properties

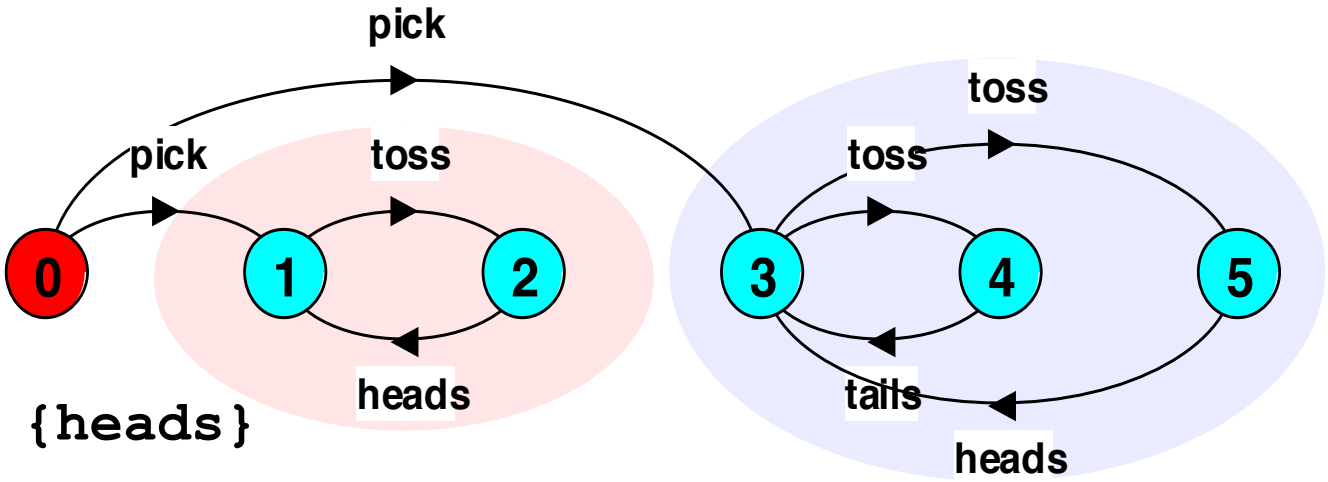
- Suppose that there were two possible coins that could be picked up:
  - a trick coin and a regular coin




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

- *TWOCOIN*:
  - `progress HEADS = {heads}` ? ☑
  - `progress TAILS = {tails}` ? ☒

Progress properties




```
progress HEADS = {heads}

progress TAILS = {tails}
```

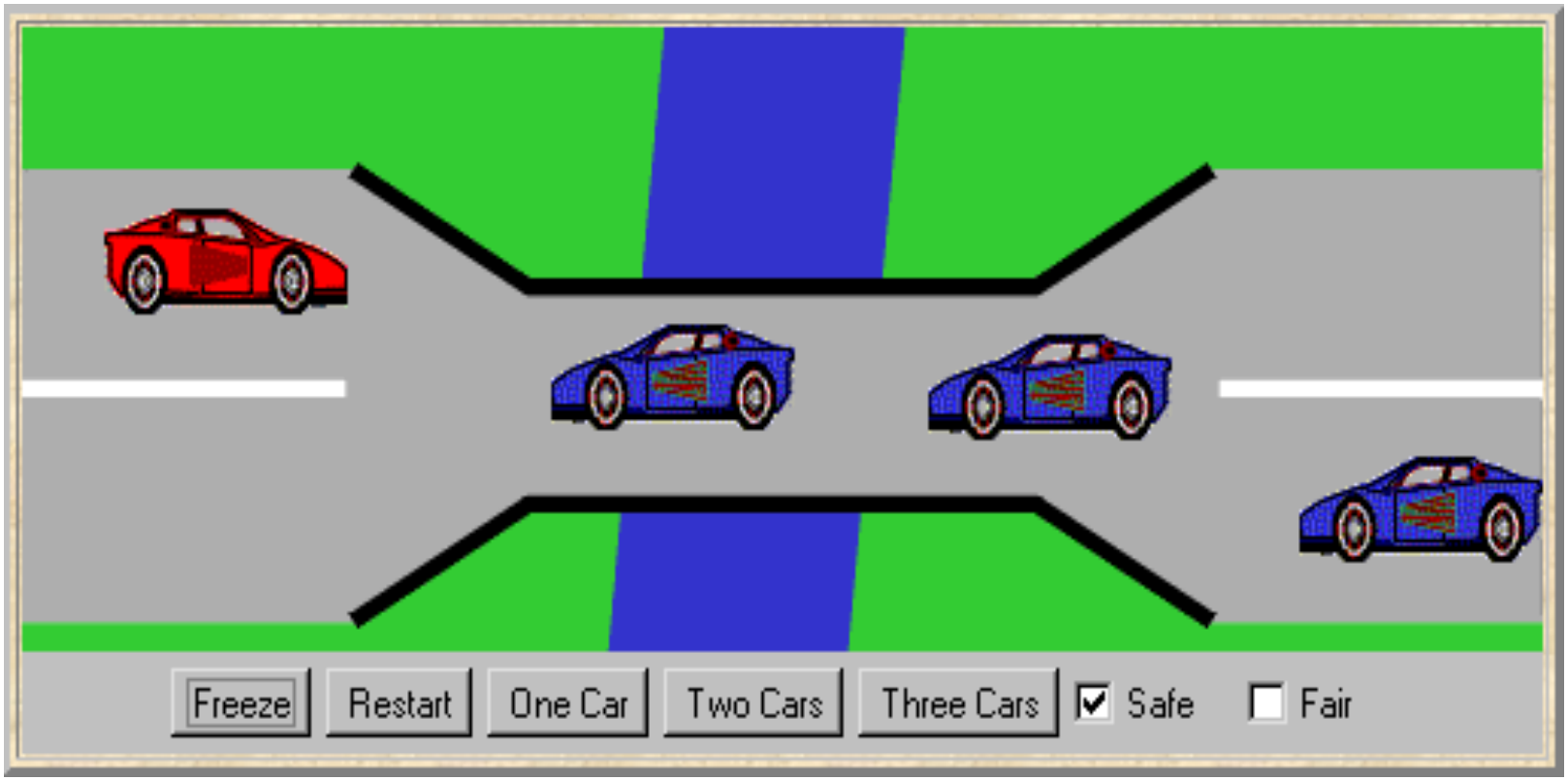
LTSA check progress

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

```
progress HEADSorTAILS = {heads,tails} ?
```

Part four: a full example (safety, deadlock and liveness)

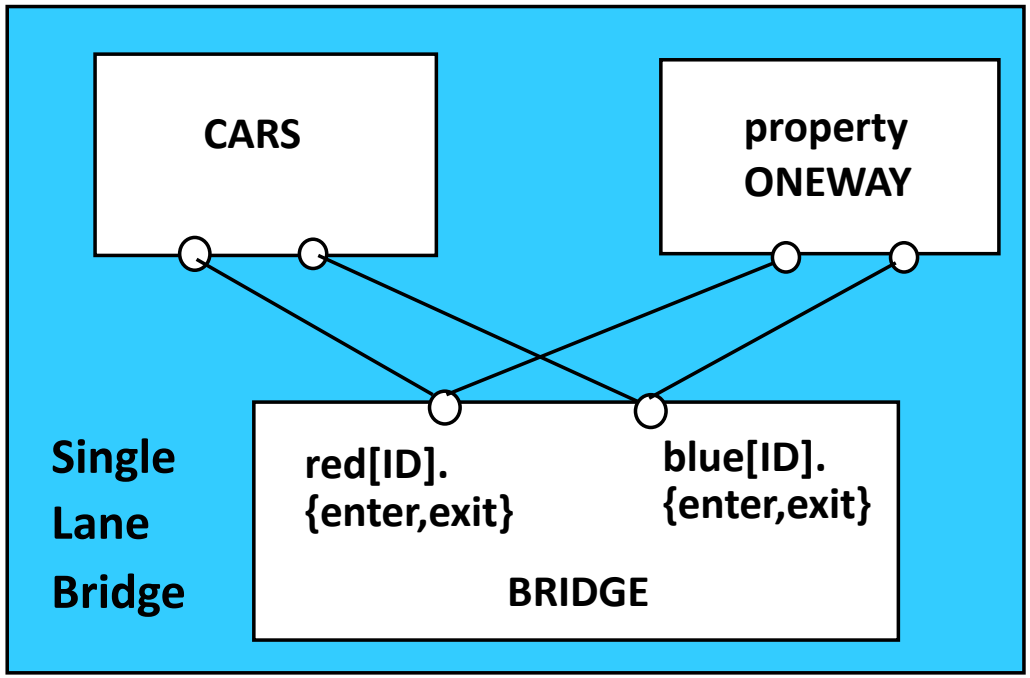
Single Lane Bridge problem



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

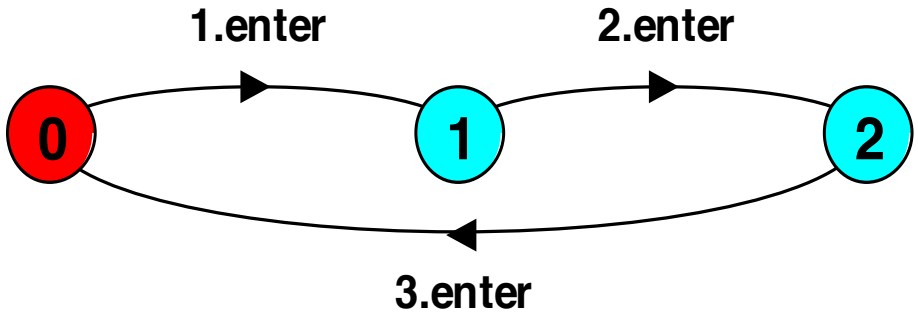
- Events or actions of interest?
  enter and exit
- Identify processes.
  cars and bridge
- Identify properties.
  oneway
- Define each process and interactions (structure).



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).

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



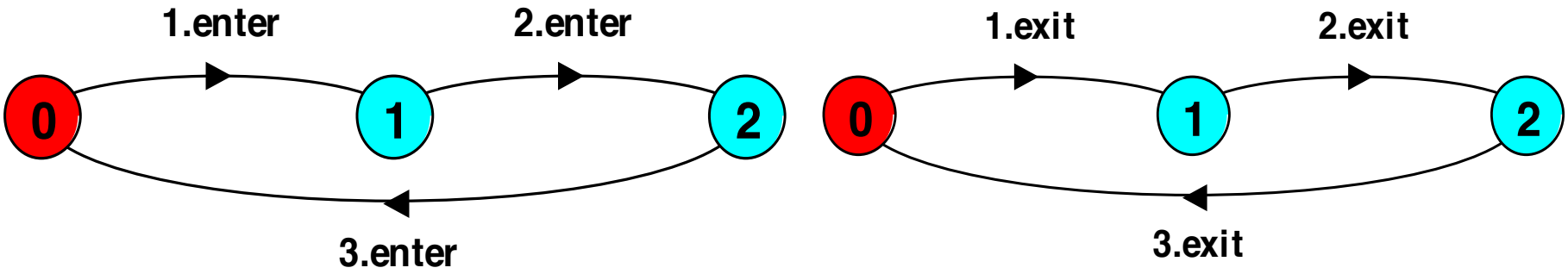
- We will have a red and a blue convoy of up to N cars for each direction:

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

- To model the fact that cars cannot pass each other on the bridge, we model a CONVOY of cars in the same direction.

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). // new
```



- *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 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)        //nb==0
          red[ID].enter -> BRIDGE[nr+1][nb]
    |     red[ID].exit  -> BRIDGE[nr-1][nb]
    |when(nr==0)        //nr==0
          blue[ID].enter-> BRIDGE[nr][nb+1]
    |     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, either a red or a blue 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

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

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

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

- *With the* **BRIDGE** *contraints, is the safety property* **ONEWAY** *violated?*

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

No deadlocks/errors
```

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) {}
  }
}
```

Université Nice Sophia Antipolis — cnrs

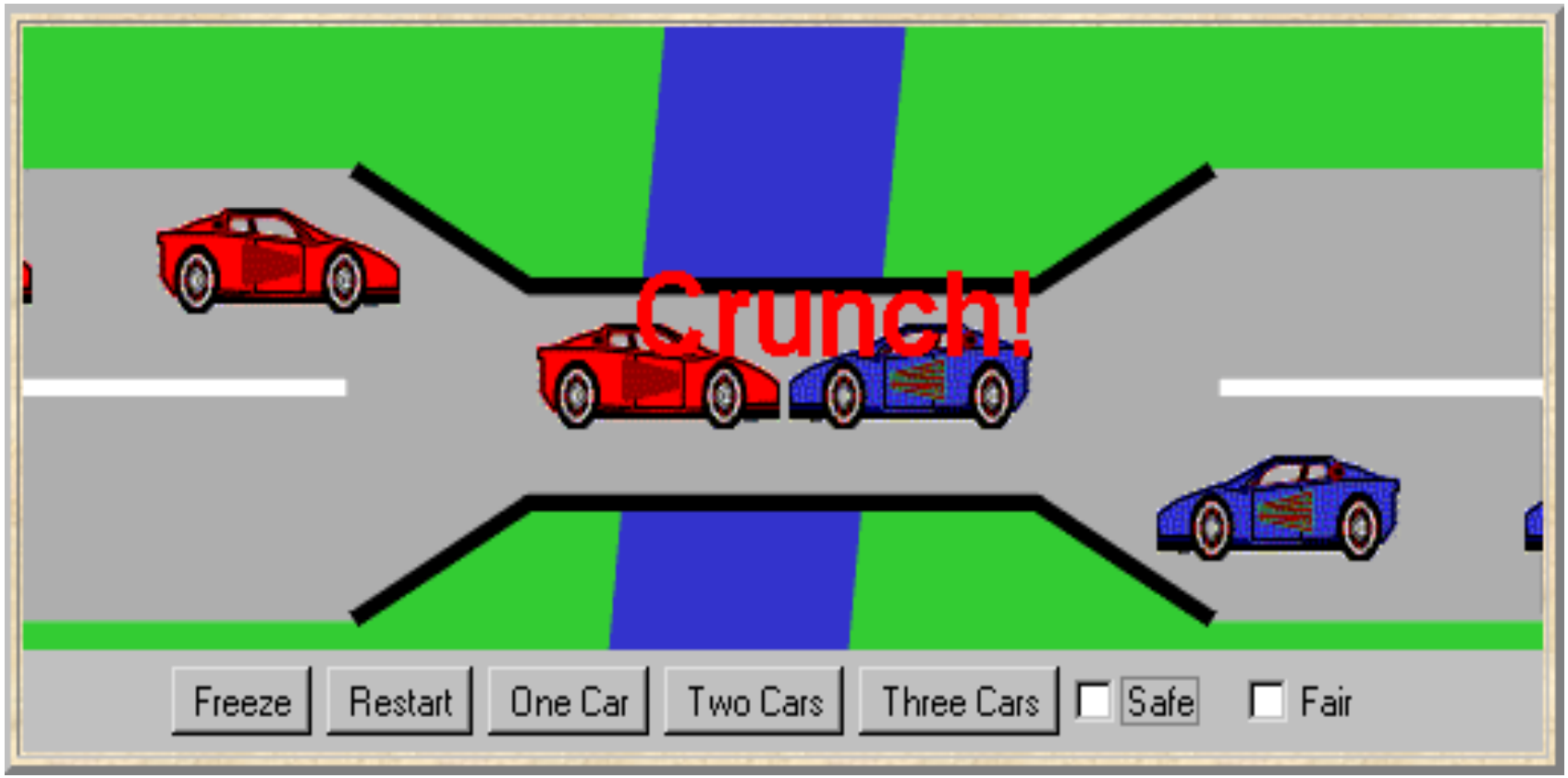
Similarly for the `BlueCar`

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.

Single Lane Bridge



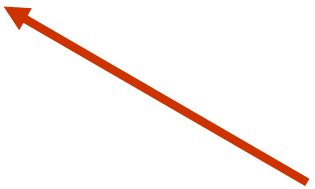
Single Lane Bridge - SafeBridge

```
classSafeBridge 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();
  }
}
```

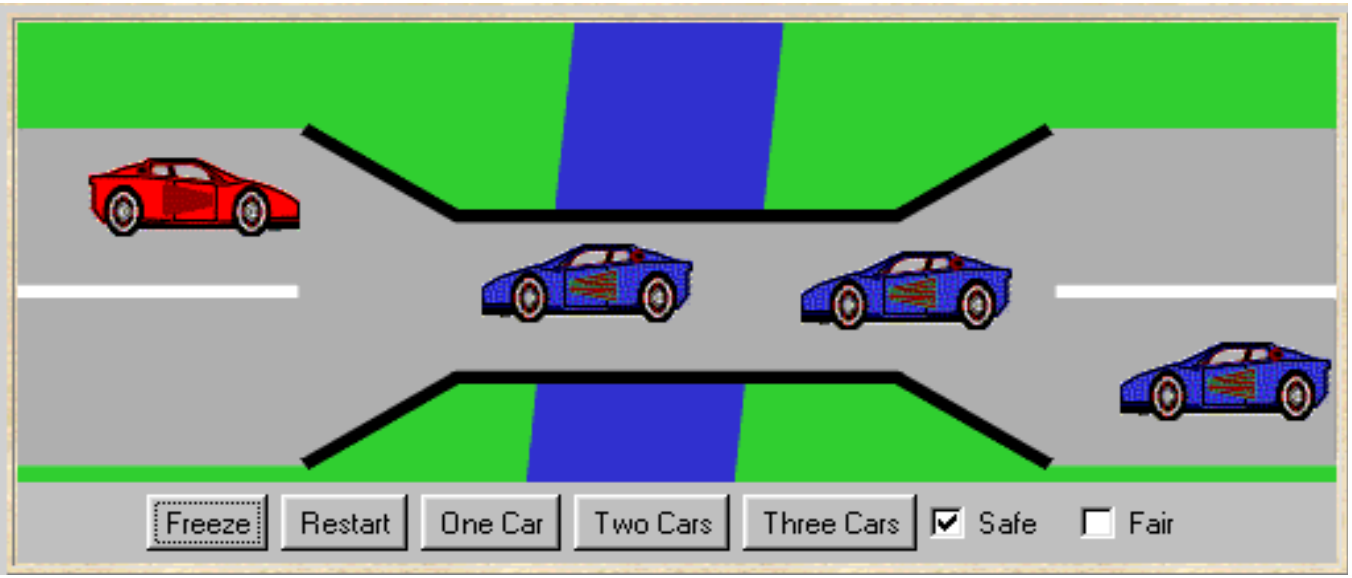


- To avoid unnecessary thread switches, we use *conditional notification* to wake up waiting threads only when the number of cars on the bridge is zero i.e. when the last car leaves the bridge.
- *But does every car eventually get an opportunity to cross the bridge? This is a liveness property.*

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 ("<<")

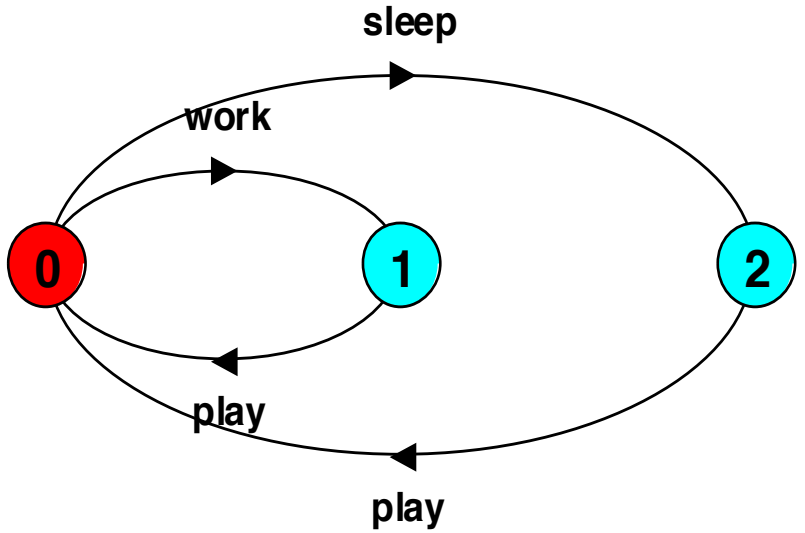
- `||C = (P||Q)<<{a1,…,an}` specifies a composition in which the actions `a1,..,an` have **higher** priority than any other action in the alphabet of `P||Q` including the silent action `tau`. *In any choice in this system which has one or more of the actions `a1,..,an` labeling a transition, the transitions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

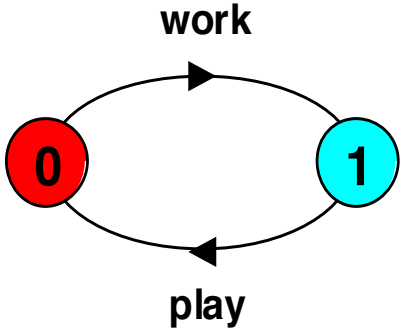
Progress - action priority

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

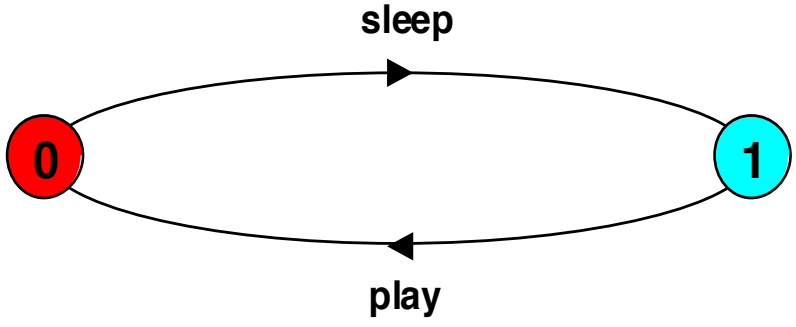


- Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

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




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



Université Nice Sophia Antipolis cnrs

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}.
```

- ***Progress Analysis? LTS?***

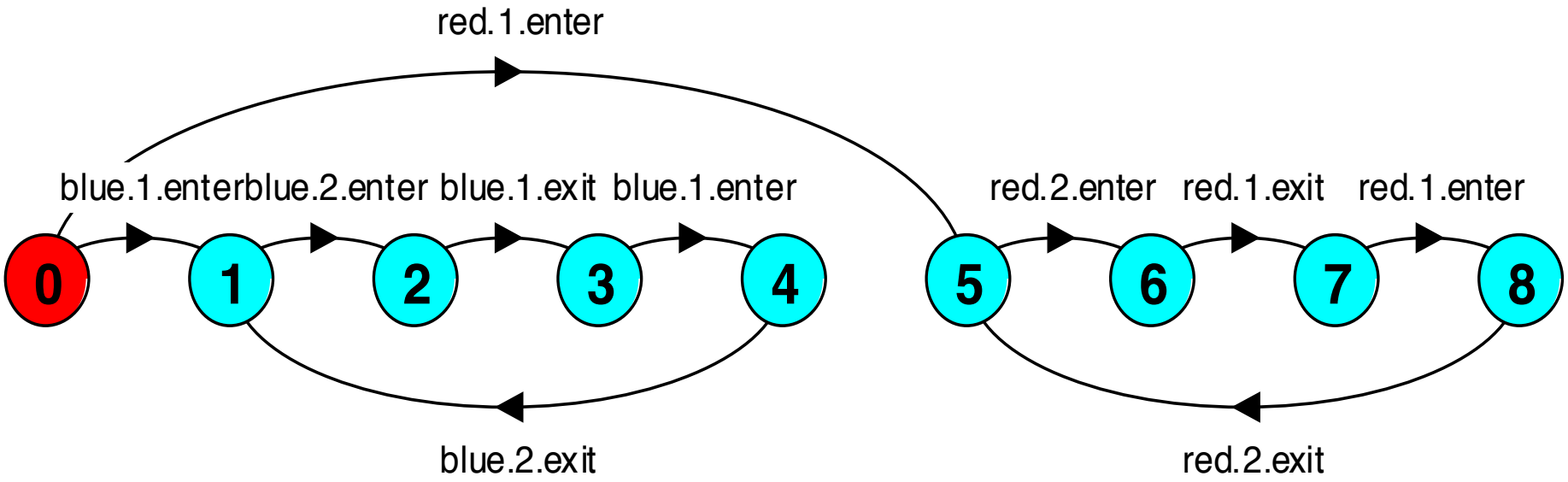
congested single lane bridge model

```
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 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 – number of red cars on the bridge
        wr – number of red cars waiting to enter
        nb – number of blue cars on the bridge
        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]
  ).
```

Université Nice Sophia Antipolis

cnrs

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 set **bt** to true initially giving blue initial precedence.

Progress - 2 nd 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; //count of red cars on the bridge
  private int nblue = 0; //count of blue cars on the bridge
  private int waitblue = 0;   //count of waiting blue cars
  private int waitred = 0;    //count of 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();
 }
}
```

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

Université Nice Sophia Antipolis

cnrs

i3s sophia antipolis

Q&A

http://www.i3s.unice.fr/~riveill

QCM

- Vous devez obligatoirement répondre en noircissant les cases sans utiliser le blanc masque.
  - Correct → 
  - Incorrect →   

- Barème prévisionnel :
  - Questions fermées simples : +3 bonne réponse, 0 pas de réponse, -0,5 mauvaise réponse, -0,5 si plus d'une case cochée.
    - Un et une seule bonne réponse
  - Questions fermées multiples : +1 bonne case cochée ou mauvaise case non cochée, -0,5 bonne case non cochée ou mauvaise case cochée, 0 pas de réponse, -1 seuil minimum
    - 0, 1 ou plusieurs bonnes réponses
  - Questions ouvertes : barème variable donné sur la copie. Pas de points négatifs.

- La note obtenue sera ramenée à 20 par règle de 3 et arrondi au ½ points