Concurrent Programming 19530-V (WS01)

Lecture 8: Safety and Liveness Properties

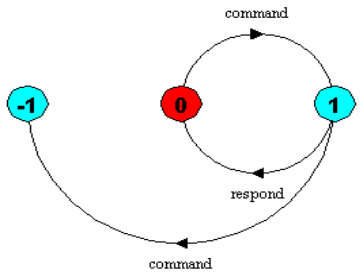
Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Safety Properties

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

- What ***bad*** things can happen?
  - **STOP** process or deadlocked states (i.e., no out-going arcs)
  - **ERROR** process (-1) used to detect erroneous behavior

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



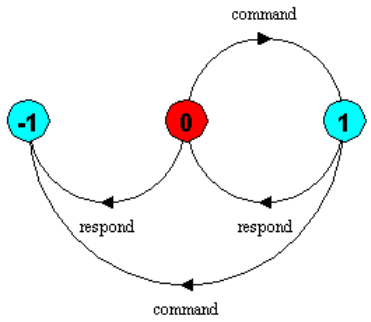
Safety analysis using LTSA

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

Safety Property Specification

- Explicit **ERROR** conditions in a process specify behavior that should ***not*** occur
- In complex systems, it is often better to specify safety properties by stating the behavior that ***should*** occur

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



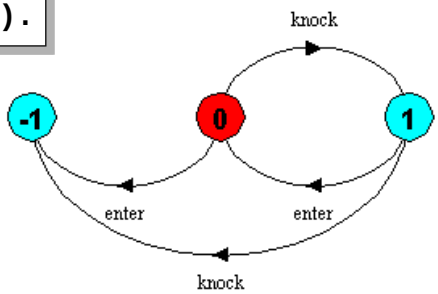
Can also use LTSA to analyze safety properties

Safety Property Specification

Consider a safety property **POLITE**, which asserts that is it 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**.

*This means that if **P** is composed with process **S**, then valid traces of actions in the alphabet of **S** that intersect the 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.

Mutual Exclusion Safety Example

How do we check that a process ensures mutual exclusion?

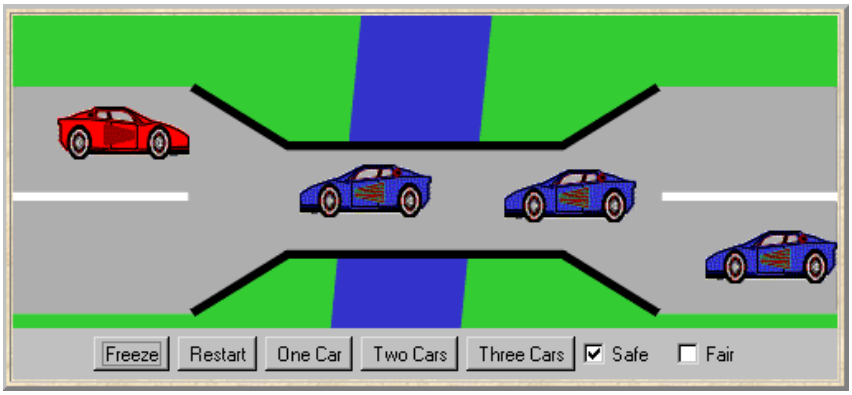
```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

We construct a safety property to verify mutual exclusion...

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

We can use LTSA to analyze this for correctness; 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.

Modeling Single-lane Bridge

- Events or actions of interest
  - **enter** and **exit**
- Identify processes
  - **CAR** and **BRIDGE**
- Identify properties
  - **ONEWAY**
- Define each process and property
  - Interactions and structure

Car 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).
```

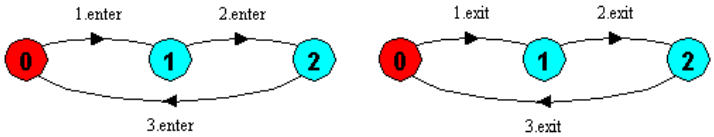
To model the fact that cars cannot pass eadch other on the bridge, we model a **CONVOY** of cars in the same direction. We will have a **red** and a **blue** convoy of up to **N** cars for each direction:

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

Convoy Model (No Passing Constraint)

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

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



Permitted: `1.enter->2.enter->1.exit->2.exit`

Not permitted: `1.enter->2.enter->2.exit->1.exit`

(i.e., no passing)

Bridge Model

Cars can move concurrently on the bridge only if they are going in the ***same direction***. The bridge counts the number 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 is initially empty,
// nr is red count, nb is blue count
BRIDGE = BRIDGE[0][0],
BRIDGE[nr:T][nb:T] =
  (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]).
```

Even when counters are 0, the **`exit`** can decrement counters. LTSA maps these undefined states to **ERROR**.

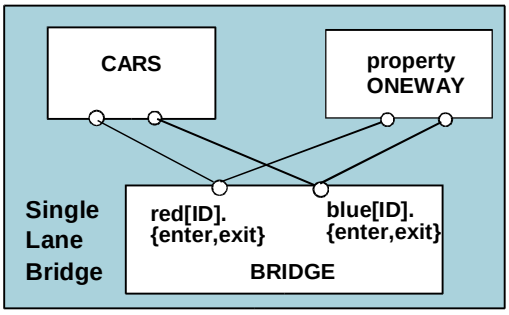
One-way Safety Property

Cars can move concurrently on the bridge only if they are going in the ***same direction***. The bridge counts the number of blue and red cars on the bridge. Red cars are ***only allowed to enter*** when the blue count is zero ***and vice-versa***.

```
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 Composition

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



Single-lane Bridge Analysis

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

Is safety property ***ONEWAY*** *violated?*

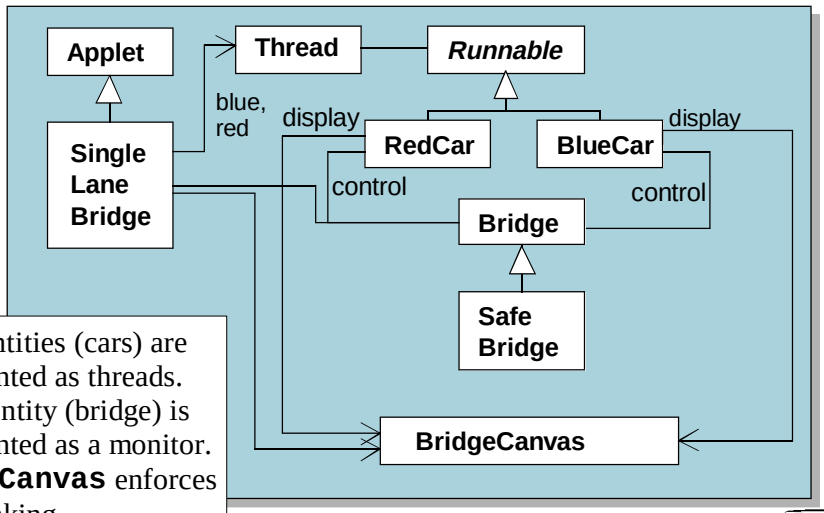
```
No deadlocks/errors
```

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

Without the ***BRIDGE*** *constraints, is safety property* ***ONEWAY*** *violated?*

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

Single-lane Bridge Implementation



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

Single-lane Bridge Implementation

An instance of **BridgeCanvas** class is created by **SingleLaneBridge** applet – reference is passed to **RedCar** and **BlueCar** objects.

```
class BridgeCanvas extends Canvas {

  public void init(int ncars) {…} // set number of cars

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

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

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

Single-lane Bridge Implementation

```
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 **BlueCar**

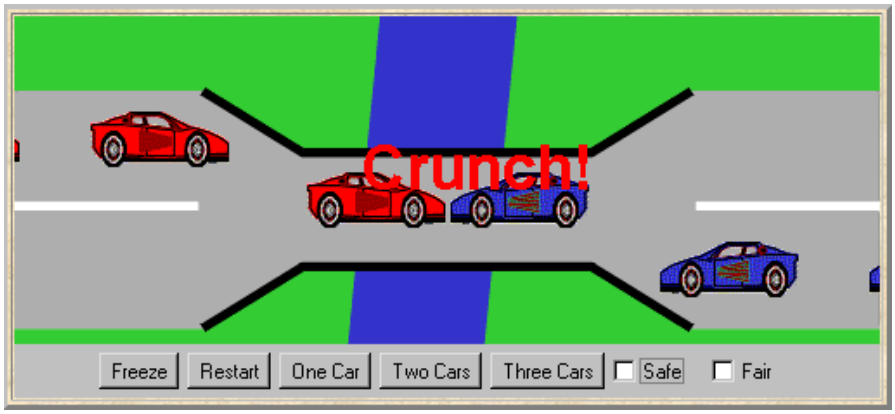
Single-lane Bridge Implementation

```
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.

What is the result?

Single-lane Bridge Implementation



How can we make the bridge safe?

Single-lane Bridge Implementation

```
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();
  }
  // continued on next slide...
```

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

Single-lane Bridge Implementation

```
  // continued from previous slide...

  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?

Liveness

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

A ***liveness property***, on the other hand, asserts that something ***good*** *eventually* happens.

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

A ***progress property*** is a restricted class of liveness properties; progress properties assert that an action will ***eventually be executed***. Progress is the ***opposite of starvation***, the name given to a concurrent programming situation in which an action is never executed.

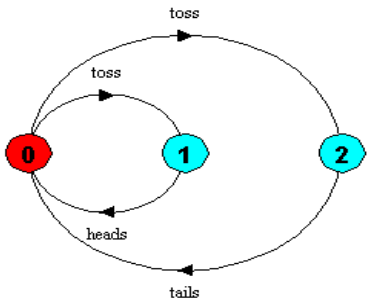
Progress and Fair Choice

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).
```



Specifying 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 process:
```
progress HEADS = {heads}
progress TAILS = {tails}
```

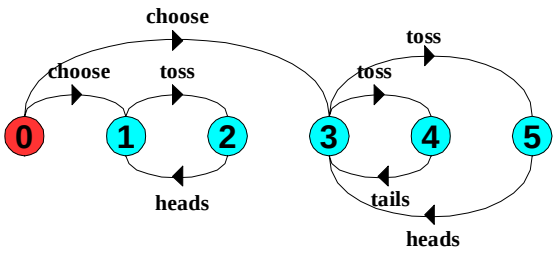
LTSA check of COIN process with above progress properties

```
No progress violations detected.
```

Progress Properties

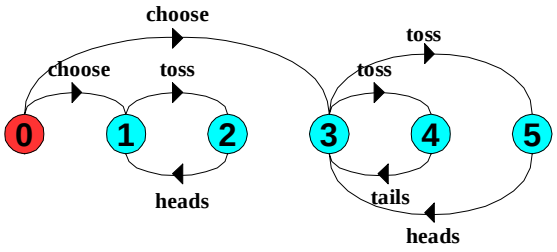
Suppose we choose from two coins, a *regular coin* and a *trick coin*...

```
TWOCOINS = (choose->COIN | choose->TRICK),
TRICK = (toss->heads->TRICK),
COIN = (toss->heads->COIN | toss->tails->COIN).
```



TWOCOIN process:
```
progress HEADS = {heads}
progress TAILS = {tails}
```

Progress Properties



LTSA finds progress violation:

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

This property is satisfied

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