Concurrent Programming
19530-V (WS01)

Lecture 9:
Safety, Progress, and Fairness Continued

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

Concurrent programming – December 18, 2001

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.

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}` ☑

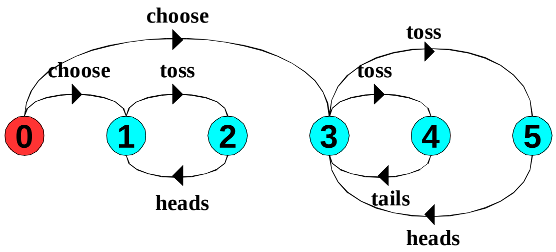
<u>LTSA check of COIN process with above progress properties</u>

```
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).
```

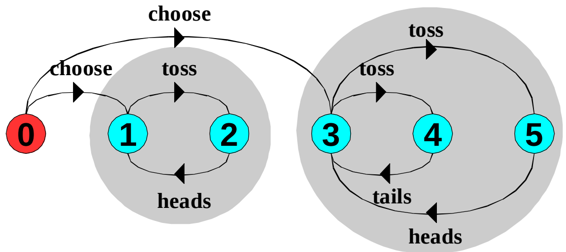


TWOCOIN process: `progress HEADS = {heads}` ☑
`progress TAILS = {tails}` ✗

Progress Analysis

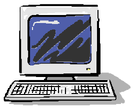
A *terminal set of states* is one in which every state is reachable from every other state in the set via one or more transitions and there is no transition from within the set to any state outside the set.

Terminal sets for **TWOCOIN**:
{1,2} and {3,4,5}



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

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



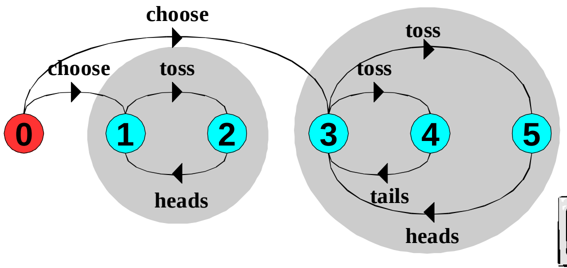
Progress Analysis

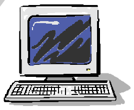
A progress property is *violated* if analysis finds a terminal set of states in which *none* of the progress set actions appear.

`progress TAILS = {tails}` (fails in {1,2})

Default analysis: Given fair choice, *every* action in the alphabet of the target system should execute infinitely often. This is equivalent to specifying a *separate progress property for every action*.

Default analysis for **TWOCOIN**?





Progress Analysis

Default analysis for **TWOCOIN**:

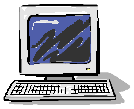
Terminal set {3,4,5}

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

Terminal set {1,2}

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

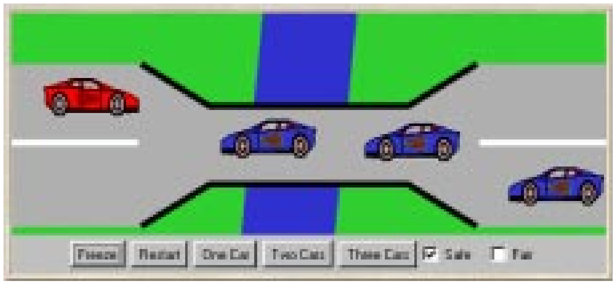
If the default holds, then every other progress property holds, i.e., every action is executed infinitely often and the system consists of a single terminal set of states.



Single-lane Bridge and Progress

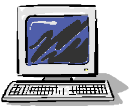
The Single Lane Bridge implementation can permit progress violations. However, if default progress analysis is applied to the model then *no* violations are detected!

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 superimpose some *scheduling policy* for actions, which models the situation in which the bridge is *congested*.



Action Priorities

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 all other actions 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***.

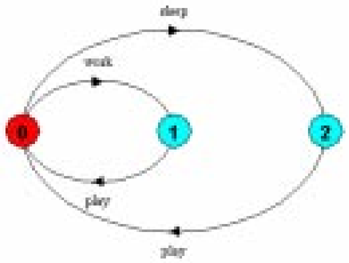
Low Priority (“>>”)

`||C = (P||Q)>>{a1,…,an}` specifies a composition in which the actions `a1,…,an` have ***lower*** priority than all other actions 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***.

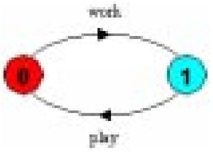
Progress Properties

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

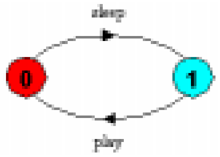


Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

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



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



Congested Single-lane Bridge Model

```
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

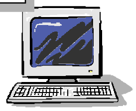
Congestion using action priority?

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}.
```

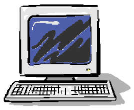
Progress Analysis ? LTS?



Congested Single-lane Bridge Analysis

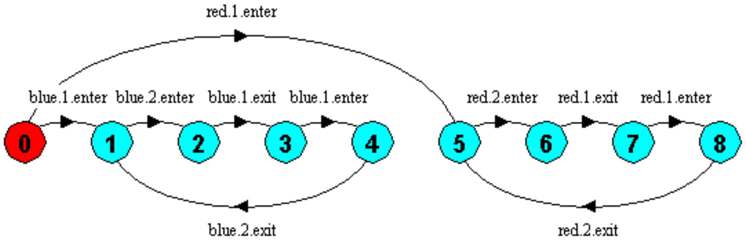
```
Progress violation: BLUECROSS
Path to terminal set of states:
      red.1.enter
      red.2.enter
Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter,
red.2.exit, red.3.enter, red.3.exit}

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 Analysis

```
||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?

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**:

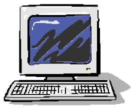
Red cars are only allowed to enter the bridge if there are no *blue* cars on the bridge *and there are no blue cars waiting* to enter the bridge.

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.

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]).
```

Is it okay now?



Revised Single-lane Bridge Analysis

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

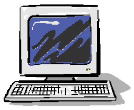
The trace is the scenario in which there are cars waiting at both ends, and consequently, the bridge does not allow either red or blue cars to enter.

Solution?

Introduce some ***asymmetry*** in the problem (e.g., dining philosophers).

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

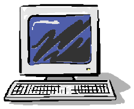
Arbitrarily, **`bt`** is set to true giving blue initial precedence.



Revised 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]
  ).
```

Is it okay now? Yes.



Revised Bridge Implementation

```
class FairBridge extends Bridge {

  private int nred  = 0; // number of red cars on bridge
  private int nblue = 0; // number of blue cars on bridge
  private int waitblue = 0; // number of blue cars waiting
  private int waitred = 0;  // number of blue cars waiting
  private boolean blueturn = true; // blue's turn

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

Revised Bridge Implementation

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

  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();
  }
}
```

Notice that we did not need to add a ***request*** monitor method; the existing enter methods were modified to increment wait counts before testing whether or not the caller can access the bridge.