Safety and Liveness Properties

SE 3BB4

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Safety and 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

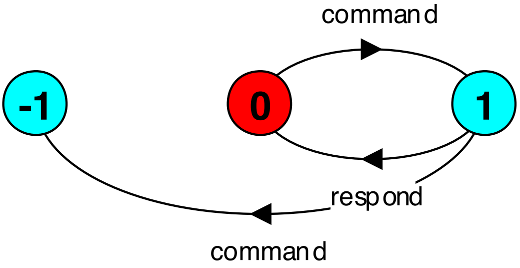
Practice: threads and monitors

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

$ACTUATOR = (command \rightarrow ACTION)$
$ACTION = (respond \rightarrow ACTUATOR \mid command \rightarrow ERROR)$



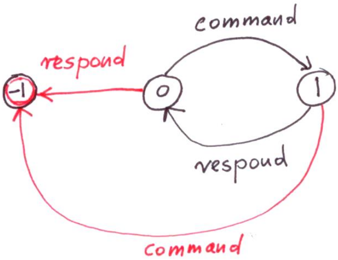
- Smallest trace to 'ERROR' : $command \rightarrow command$
- This smaller trace will be produced by LTSA with 'Safety Analysis'.

Safety Properties

- ERROR condition states what is not required (cf. exceptions).
- In complex systems, it is usually better to specify safety **properties** by stating directly what is required.
- Safety properties are specified in FSP by *property* process.
- Syntax of this process: just a prefix *property*
- Safety properties are composed with a target system to ensure that the specified property holds for that system.

property $SAFE\_ACTUATOR = (command \rightarrow respond \rightarrow SAFE\_ACTUATOR)$

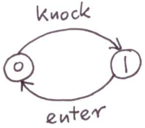
- Semantics differs from 'normal' FSP!



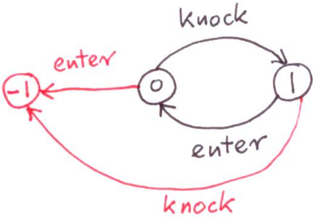
Semantics of Safety Property Processes

- In all states, all the actions in the alphabet of a property process are eligible choices, and everything that is not specified explicitly goes to *ERROR* state (process) (-1).

$POLITE = (knock \rightarrow enter \rightarrow POLITE)$



$property\ POLITE = (knock \rightarrow enter \rightarrow POLITE)$

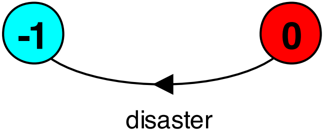


Safety Properties

- Safety *property* P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P. Those actions that are not part of the specified behaviour of P are transitions to the ERROR state.
- Thus, if P is composed with S, then traces of actions in $alphabet(S) \cap alphabet(P)$ 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* behaviour. However, if a behaviour can occur which violates the safety property, then ERROR is reachable.
- Properties must be deterministic to be transparent.
- The textbook states: '*Experience has shown that this is rarely a restriction in practice*', but not everyone agrees!

- 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 behaviours **in its alphabet.**

$LOOP = (mutex.down \rightarrow enter \rightarrow exit \rightarrow mutex.up \rightarrow LOOP)$
$\| \; SEMADEMO = (p[1..3] : LOOP \; \| \; \{p[1..3]\} :: mutex : SEMAPHORE(1))$
$SEMAPHORE(N) = SEMA(N)$
$SEMA[v : Int] = (when(v < N) \; up \rightarrow SEMA[v + 1] \; \|$
$\quad when(v > 0) \; down \rightarrow SEMA[v - 1])$
$property \; MUTEX = (p[i : 1..3].enter \rightarrow p[i].exit \rightarrow MUTEX),$
where $MUTEX$ expands to:
$MUTEX = (p[1].enter \rightarrow p[1].exit \rightarrow MUTEX \mid$
$\quad p[2].enter \rightarrow p[2].exit \rightarrow MUTEX \mid p[3].enter \rightarrow p[3].exit \rightarrow MUTEX)$

$$\| \; CHECK = (SEMADEMO \; \| \; MUTEX)$$

- The safety property $MUTEX$ specifies that when when a process enters the critical section ($p[i].enter$), the same process must exit the critical section ($p[i].exit$) before another process can enter.
- The solution works for the above case, but:

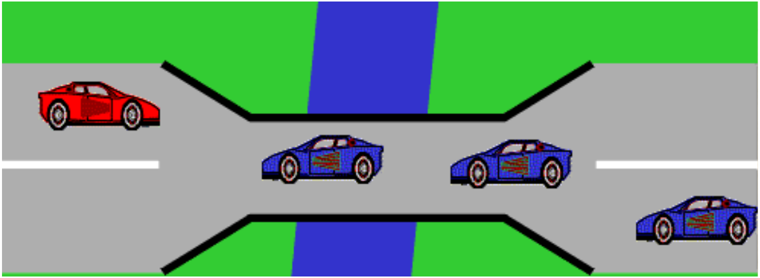
Safety: Mutual Exclusion

$\| \mathit{SEMADEMO\_2} = (p[1..3] : \mathit{LOOP} \;\|\; \{p[1..3]\} :: \mathit{mutex} : \mathit{SEMAPHORE}(2))$

$\| \mathit{CHECK\_2} = (\mathit{SEMADEMO\_2} \;\|\; \mathit{MUTEX})$

- The process $\| \mathit{CHECK}$ produces a trace: $p.1.\mathit{mutex}.\mathit{down} \rightarrow p.1.\mathit{enter} \rightarrow p.2.\mathit{mutex}.\mathit{down} \rightarrow p.2.\mathit{exit}$
- In this case $\mathit{SEMAPHORE}(2)$ can be interpreted that we allow two processes in the critical section. It does make sense, however it is then usually not called '*critical section*'.

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

'$i\%N+1$' means '$i \mod N+1$', i.e. i divided modulo N plus 1.

- Events or actions of interest?

  enter and exit

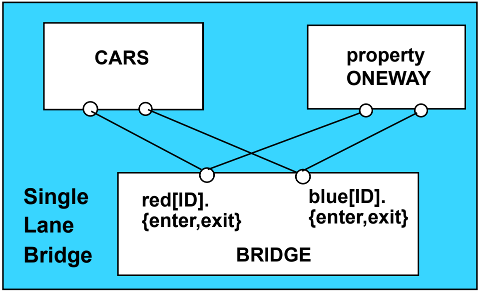
- Identify processes.

  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).
```

No overtaking constraints: 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:

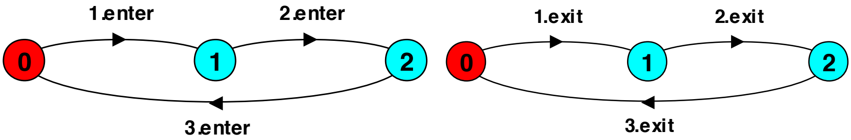
```
||CARS = (red:CONVOY || blue:CONVOY).
```

Single Lane Bridge - *CONVOY* model

'$i\%N+1$' means '$i \mod N+1$', i.e. i divided modulo N plus 1.

```
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.

- 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)
         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 uses this assumption.

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 - Analysis of This Simple Model

```
||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 - Analysis of This Simple Model

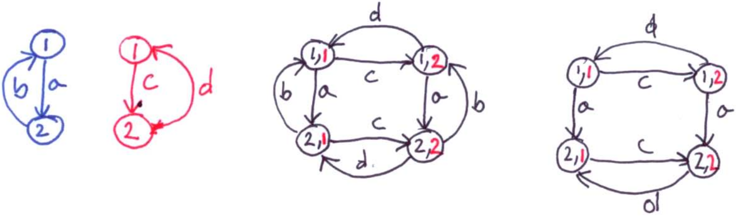
Question: How *ONEWAY* can be seen as *transparent* and *nondeterministic*? It contains '|' operator.

Answer: It could be interpreted as '*semantically deterministic*', but this concept is hidden in the textbook.

- *property P* is a *passive* process, but it is just a process with a little bit different semantics. However it can entirely be simulated by just a standard process.
- *Transparency* and *Determinism* are not defined very precisely in the textbook.
- *Determinism* is of semantical nature, it follows from the fact that every process involved in the definition of '*property P*' is considered to be passive and '*local*' to P.

Alphabet Extension: Processes

- Let: $P = (a \rightarrow b \rightarrow P)$, $Q = (c \rightarrow d \rightarrow Q)$ and
  $Qa = (c \rightarrow d \rightarrow Q) + \{b\}$.
- $alphabet(P) = \{a, b\}$, $alphabet(Q) = \{c, d\}$,
  $alphabet(Qa) = \{b, c, d\}$, so $alphabet(P) \cap alphabet(Q) = \emptyset$, while
  $alphabet(P) \cap alphabet(Qa) = \{b\}$.
- Define $||PQ = P||A$ and $PQa = P||Qa$.
- Labelled Transition Systems are:

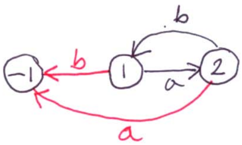


P $\quad Q = Qa \quad$ $||PQ$ $\quad$ $||PQa$

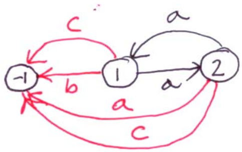
- Clearly $||PQ \neq ||PQa$!

Alphabet Extension: Properties

- Let *property* $P = (a \rightarrow b \rightarrow P)$ and
  property $Pc = (a \rightarrow b \rightarrow P) + \{c\}$.
- Labelled Transition Systems are:



P



Pc

- Clearly $P \not\equiv Q$!

Liveness

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

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

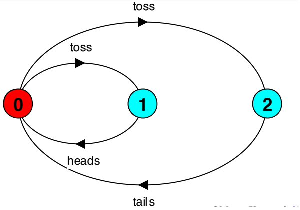
Single Lane Bridge: *Does every car eventually get an opportunity to cross the bridge?*

ie. to make ***PROGRESS?***

A progress property asserts that it is *always* the case that a particular 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.

- **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.

$$COIN = (toss \rightarrow heads \rightarrow COIN \mid toss \rightarrow tails \rightarrow COIN)$$



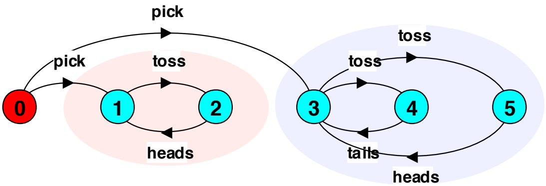
- If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.
- This requires Fair Choice !
- Ambiguity! What does it really mean?
  1. Always happens
  2. Sometimes Happens
  3. Is this an enforcement or just an observation?
- This is not discussed in the textbook!

Progress Properties

- *progress* $P = \{a_1, a_2, \ldots, a_n\}$ defines a *progress property* P which asserts that in an infinite execution of a target system, at least *one* of the actions $a_1, a_2, \ldots, a_n$ will be executed infinitely often.
- For *COIN* system we might have:
  progress $HEADS = \{heads\}$, *progress* $TAILS = \{tails\}$.
- Textbook says that is both cases '*no progress violations detected*'.
- **WHY??** What about a possible infinite traces:
  $heads \rightarrow heads \rightarrow \ldots$, and $tails \rightarrow tails \rightarrow \ldots$?
  They are not disallowed!
- Is the definition of "*progress property*" correct?
- Before answering the last question, let us consider another example from the textbook.

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}` ✗

- According to the textbook, for *TWOCOINS* the *progress HEADS* property is satisfied, while the *progress TAILS* property is not!
- **This means the definition of progress property in the textbook is wrong!**

Progress Property: Correct Version

- The correct definition could be the following one:
  progress $P = \{a_1, a_2, \dots, a_n\}$ defines a *progress property P* which asserts that in any state of a target system, there is always a **continuation** trace which contains at least one element of $\{a_1, a_2, \dots, a_n\}$.
- The above definition does not need the concept of infinite trace.
- For *TWOCOIN* we have:
  progress $HEADS = \{heads\} \leftarrow$ YES
  progress $TAILS = \{tails\} \leftarrow$ NO
  progress $HEADSorTAILS = \{heads, tails\} \leftarrow$ YES
- *progress property* is not a process-like structure, its semantics is in reality defined in terms of '*terminal sets of states*', which is implicit in the textbook.

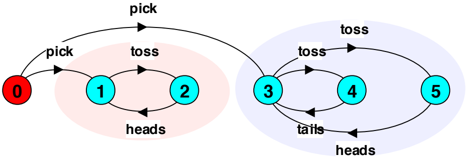
Definition (Terminal Set of States)

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 Set for *COIN*: $\{1, 2, 3\}$

Terminal sets for TWOCOIN:

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



- According to the textbook:
  Given *fair choice*, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often.

- This means that **fair choice** policy **enforces** some rules. These rules could be explained better had **infinite** traces been introduced, as for example in most temporal logics and model checking.
  Intuitively $(toss \rightarrow heads)^{\infty}$ or even
  $(toss \rightarrow tails) \rightarrow (toss \rightarrow heads)^{\infty}$ are **illegal** infinite behaviours.

- **In other words, there is a DEMON (ORACLE) which controls executions (runs) and after a finite numbers of choices in one direction, it stops it and says 'Listen lad, enough is enough, move over and let the other guy to move'.**
- *Fair choice* semantics is **not** a standard semantics. There are cases when it is just not needed!
- 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 it represents a *potential progress violation*!

Total Progress

- Why in the definition of *progress* $P = \{a_1, a_2, \ldots, a_n\}$ we have a statement: '*at least one*'?
- Because in many applications all $\{a_1, a_2, \ldots, a_n\}$ can be considered as '*equivalent*'. For example we have $\{get_1, get_2, \ldots, get_n\}$, but we really want any get_i to be executed infinitely many times.
- However having also for instance:

$$total\_progress\ P = \{a_1, a_2, \ldots, a_n\},$$

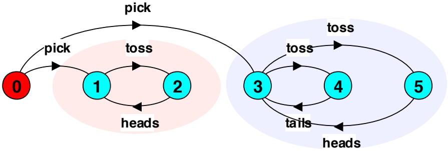
where '**at least one**' is replaced by '**all**' is a valid, and often very useful, option!

Another Definition of *Progress Property*

- A *progress property* is **violated** if analysis finds a terminal set of states in which *none* of the progress set action appears.
- The above statement can also be a definition of progress properties, and it is much more precise than the original (wrong as we showed) definition given in the textbook.

Default

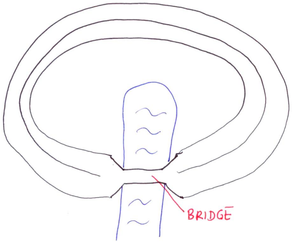
- **Default**: given fair choice, for every action in the alphabet of the target system, that action will be executed infinitely often. This is equivalent to specifying a *separate progress property for every action.*
- *Default* $\iff$ *total_progress P = Alphabet of All Processes*
- Default analysis for TWOCOIN: violation for *pick* and *tails*



- *DEFAULT* assumes the existence of a *Demon* that enforces *Fair Choice*.
- If the *DEFAULT* holds, then every other *progress property* holds, i.e. every action is executed infinitely often and system has a single terminal set of states.
- All these concepts could be defined more precisely if the concept of *infinite* traces were introduced!

Single Lane Bridge Again

- The Single Lane Bridge implementation **can** permit *progress* violations (assuming infinite number of cars, or a cycle like blow, not clear from the textbook statement).



- However, if *default* progress analysis is applied (i.e. a 'DEMON' that enforces FAIR CHOICE exists) to the model, then no violations are detected.

Priority

- **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**.
- Possible tool: Action Priority.
- Action priority expressions describe scheduling properties.
- **Mixing priority and concurrency is a well known problem!**

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 other, lower priority actions are discarded.*

Low Priority (">>")

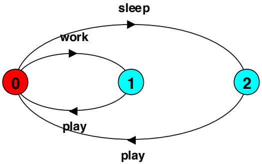
||C = (P||Q)>>{a1,…,an} specifies a composition in which the actions **a1,...,an** have **lower** priority than any other action in the alphabet of **P||Q** including the silent action **tau**.

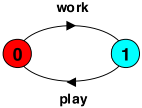
In any choice in this system which has one or more transitions not labeled by ***a1,...,an****, the transitions labeled by* ***a1,...,an*** *are discarded.*

- Is this definition clear to you?
- Where this '*discarding*' occurs?
  On the FSP level or LTS level?
- The statement '*including the silent action* τ' gives some hint, but it is still unclear!

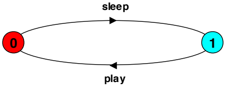
Priority: Textbook Examples

$NORMAL = (work \rightarrow play \rightarrow NORMAL \mid sleep \rightarrow play \rightarrow NORMAL)$





$\| \ HIGH = (NORMAL) << \{work\}$
$WORKAHOLIC = (work \rightarrow play \rightarrow WORKAHOLIC)$
$\| \ HIGH \approx WORKAHOLIC$, so '|' is just not used.



$\| \ LOW = (NORMAL) >> \{work\}$
$MY\_DREAM = (sleep \rightarrow play \rightarrow MY\_DREAM)$
$\| \ LOW \approx MY\_DREAM$, so '|' is just not used.

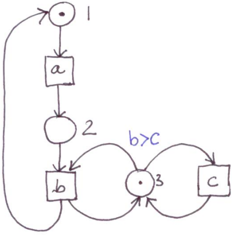
- Oversimplification? YES!

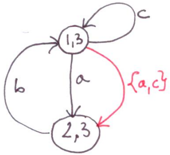
Famous Priority Example

The concurrent system *PRIORITY* comprises two sequential subsystems, such that:

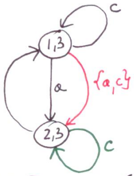
- the first subsystem can cyclically engage in event *a* followed by event *b*;
- the second subsystem can cyclically engage in event *c* or in event *b*;
- the two subsytems synchronize by means of handshake communication;
- there is a *priority constraint* stating that if it is possible to execute event *b* then *c* must not be executed.

Petri Net Solution





Full Transition Graph



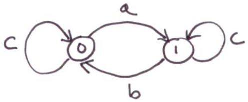
Full Transition Graph
without b>c

- It models '*not later than*' as $\{a, c\}$ and $c \rightarrow a$ are allowed but $a \rightarrow c$ is not!

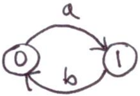
FSP Solution

$P1 = (a \rightarrow b \rightarrow P1)$
$P2 = (b \rightarrow P2 \mid c \rightarrow P2)$
$PRIORITY = (P1 \parallel P2) << \{b\}$

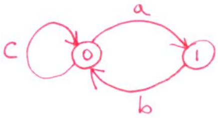
- LTS for $(P1 \parallel P2)$:



- Let $P2' = (b \rightarrow P2')$, LTS for $(P1 \parallel P2')$ is:



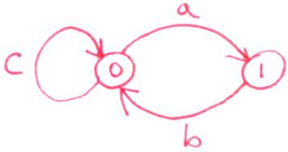
- Is LTS for *PRIORITY* the same as for $(P1 \parallel P2')$, or"



??

Priority

- The solution **must be**:



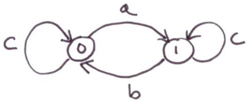
  otherwise there might be a serious problem.
- Each model has at least two levels:
  1. Syntax level.
  2. Semantics level.
- All definitions should clarify which level is dealt with!

Priority: New Version

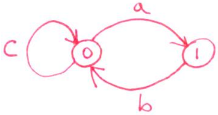
- The following addition of priority to FSP most likely works better:
  $P = (P_1 \parallel \ldots \parallel P_n) + < \{a_1 < b_1, \ldots, a_k < b_k\}$ specifies a composition in which b_i has **higher** priority than a_i, for all $i = 1, \ldots, k$.
  LTS of P is constructed as follows. First we construct LTS for $(P_1 \parallel \ldots \parallel P_n)$ is a standard manner, and next, in every state where we have a choice between a_i and b_i, we discard a_1, and remove useless states if needed.

Priority: New Version

- For our previous example we have now:
  $P1 = (a \to b \to P1)$
  $P2 = (b \to P2 \mid c \to P2)$
  $PRIORITY = (P1 \parallel P2) + < \{c < b\}$
- LTS for $(P1 \parallel P2)$:



- LTS for *PRIORITY*:



- So we are done!

```
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

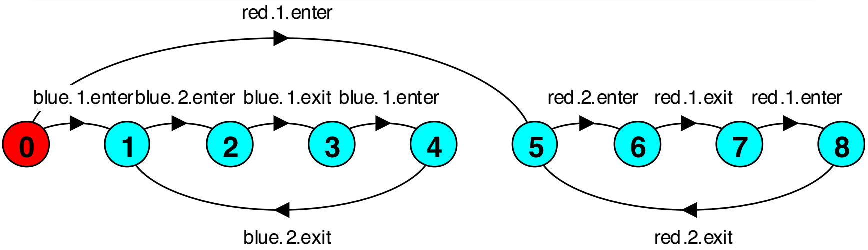
```
Progress violation: REDCROSS
Trace to terminal set of states:
     blue.1.enter
Cycle in terminal set:
     blue.2.enter
     blue.1.exit
     blue.1.enter
     blue.2.exit
Actions in terminal set:
     blue[1..2].{enter, exit}
```

```
Similarly for BLUECROSS
```

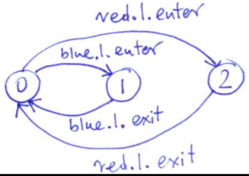
This corresponds with the observation that, with ***more than one car (N=2 say)***, it is possible that whichever colour car enters the bridge first could continuously occupy the bridge preventing the other colour from ever crossing.

The Case of Two Cars

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



- The same story if we give car entry higher priority.
- With only one car moving in each direction it is OK.



Revised Model

- The bridge needs to know whether or not cars are waiting to cross.
- Modified CAR:

$$CAR = (request \rightarrow enter \rightarrow exit \rightarrow CAR)$$

- Modified $BRIDGE$:
    - 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 blue cars waiting** to enter the bridge.

Revised 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]
   ).
```

- **Problem**:
  red.1.request → *red2.request* → *red3.request* → *blue.1.request* → *blue.2.request* → *blue.3.request*
  and **deadlock**.
- 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 to the Revised Model

- Introduce some asymmetry and alternation in the problem (cf. Dining philosophers).
- We introduce 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.

➡ **Analysis ?**

```
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]
   ).
```

- No deadlock, *BLUECROSS* and *REDCROSS* properties are not violated, no starvation.