CIS 721 - Real-Time Systems

Quiz #2 - Review

Mitch Neilsen
neilsen@ksu.edu

Outline

- Quiz #2: Wed., Dec. 2
- Topics
  - Real-Time Scheduling Theory and Algorithms
    - Priority Inheritance, NPCS, PCP, etc.
  - Real-Time Design
  - Validation and Verification (Spin, UPPAAL)

Common Approaches For Real-Time Scheduling

- **Clock-Driven (Time-Driven) Approach –** scheduling decisions are made at specific time instants.
- **Weighted Round-Robin Approach -** every job joins a FIFO queue; when a job reaches the front of the queue, its weight refers to the fraction of processor time (number of time slices) allocated to the job.
- **Priority-Driven (Event-Driven) Approach -** ready jobs with highest priorities are scheduled for execution first.
  - Scheduling decisions are made when particular events occur; e.g., a job is released or a processor becomes idle. A **work-conserving** processor is busy whenever there is work to be done.

Utilization-Based Test

- **A sufficient, but not necessary,** test that can be used to test the schedulability of a task set that is assigned priorities using the rate-monotonic algorithm.
- Compute total task utilization U(n) = U.
- Compare with worst-case utilization bound (also called **schedulable utilization**) $U_{RM}(n) = U_{RM}$:
  - If **U > 1**, then the task set is not schedulable.
  - If **$U \leq U_{RM}$**, then the task set is schedulable.
  - Otherwise, no conclusion can be made.

Schedulability Analysis

- **Utilization-Based Tests**
  - Not exact (sufficient, but not necessary).
  - Not applicable to more general task models.
- **Time-Based Tests (Response Time Analysis)**
  - Use analytic approach to predict worst-case response time of each task.
  - Compare computed worst-case response times with deadlines.
  - Exact (sufficient and necessary)

Example

Task τ_i	Period T_i	Deadline D_i	Run-Time C_i	Phase φ_i
A (High Priority)	7	7	3	0
B	12	12	3	0
C (Low Priority)	20	20	5	0

- $U = 3/7 + 3/12 + 5/20 = 13/14 \approx 0.93$
- $U_{RM} = 3\,(2^{1/3} - 1) \approx 0.78$
- Since $U_{RM} < U \leq 1.0$, no conclusion can be drawn using the Utilization-Based Test.

Response Time

- The **response time** (R_i) for task T_i is given by $R_i = e_i + I_i$ where:
  - e_i is the execution time of each job in T_i , and
  - I_i is the **maximum interference** caused by higher priority tasks in any interval [t, t + R_i).

Maximum Interference (I_i)

- The number of releases of task T_j in [t, t + R_i) is given by $\left\lceil \frac{R_i}{p_j} \right\rceil$
- The interference caused by task T_j is $\left\lceil \frac{R_i}{p_j} \right\rceil * e_j$
- The maximum interference caused by all higher priority tasks is given by

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil * e_j$$

where *hp(i)* = set of all tasks with priority greater than task T_j .

Response Time Analysis

- The (**worst-case**) **response time** (w_i) for task T_i is given by the implicit equation:

$$R_i = e_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{p_j} \right\rceil * e_j$$

- Solve by forming a recurrence relation:

$$w_i^{n+1} = e_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{p_j} \right\rceil * e_j$$

$$w_i^0 = e_i$$

Solving Recurrence

- The sequence $w_i^0, w_i^1, w_i^2, \ldots, w_i^n$

  is non-decreasing:

  - If $w_i^{n+1} = w_i^n$, then a fixed point (solution) has been found.

  - If $w_i^{n+1} > D_i$, then no solution exists.

Algorithm

Input: $e_1,..,e_m$, $p_1,..,p_m$, $D_1,..,D_m$

Output: $R_1, R_2,..., R_m$

for $i = 1$ to m

$n = 0$

$w_i^n = e_i$

loop

$$w_i^{n+1} = e_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{p_j} \right\rceil * e_j$$

if $w_i^{n+1} = w_i^n$ then

$R_i = w_i^n$

break out of loop {solution found}

if $w_i^{n+1} > D_i$ then

break out of loop {no solution}

$n = n + 1$

end loop

end for

Response Time Analysis

- For each task, T_i , compute worst-case response time (R_i).
- If ($R_i \leq D_i$) for each task T_i **,** then the task set is feasible (schedulable).
- Response Time Analysis is both necessary and sufficient.

Preemption Thresholds Task Model

- Task Set $\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$
  - Each task τ_i is characterized by (C_i, T_i, D_i), denoted $\tau_i \sim (C_i, T_i, D_i)$.
  - Each task τ_i is assigned a priority $\pi_i \in \{1,2,..,n\}$
  - and a preemption threshold $\gamma_i \in \{\pi_i, \pi_i + 1,..,n\}$.
- **Notes:**
  - 1 = lowest priority, n = highest priority.
  - π_i = static priority.
  - γ_i = dynamic priority.

Run-Time Model

- Modified fixed-priority, preemptive scheduling.
- When task τ_i is released, it is scheduled using its static priority π_i .
- After task τ_i starts executing, another task τ_j can preempt τ_i only if $\pi_j > \gamma_i \geq \pi_i$.

Extremes

- If $\gamma_i = \pi_i$ for each i, then the result is preemptive, priority-based scheduling.
- If γ_i = n (max. priority) for each i, then the result is non-preemptive, priority-based scheduling.

Sharing Resources

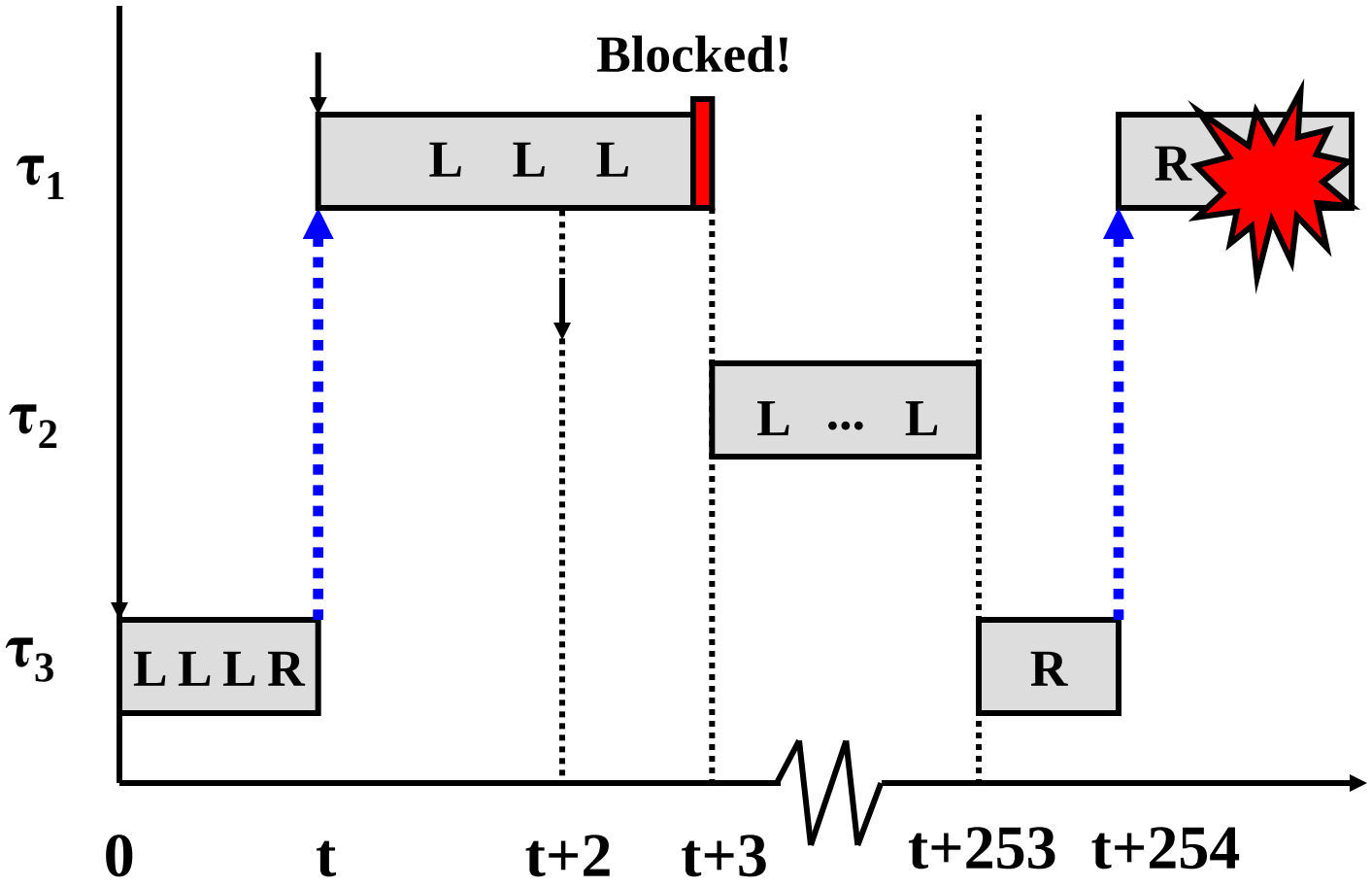
- Many applications require concurrent access to shared resources.
- **One solution:** To synchronize access to a shared resource, **semaphores** or **monitors** can be used.
- If a task tries to lock a binary semaphore that is already locked, then the task is **blocked**.
- Our prior analysis assumed that there is no blocking due to resource sharing.
- **Q:** How does resource sharing impact feasibility? How can scheduling algorithms be modified to support sharing of resources?

Tasks

- **Task τ_1 L L L R L**
- **Task τ_2 L L ... L**
- **Task τ_3 L L L R R L ... L**

R = access resource in critical section

Priority Inversion



Blocking

- **Blocking** occurs when a higher priority task is waiting on a lower priority task; e.g., for example, blocking occurs when a lower priority task is holding a resource that is requested by a higher priority task.

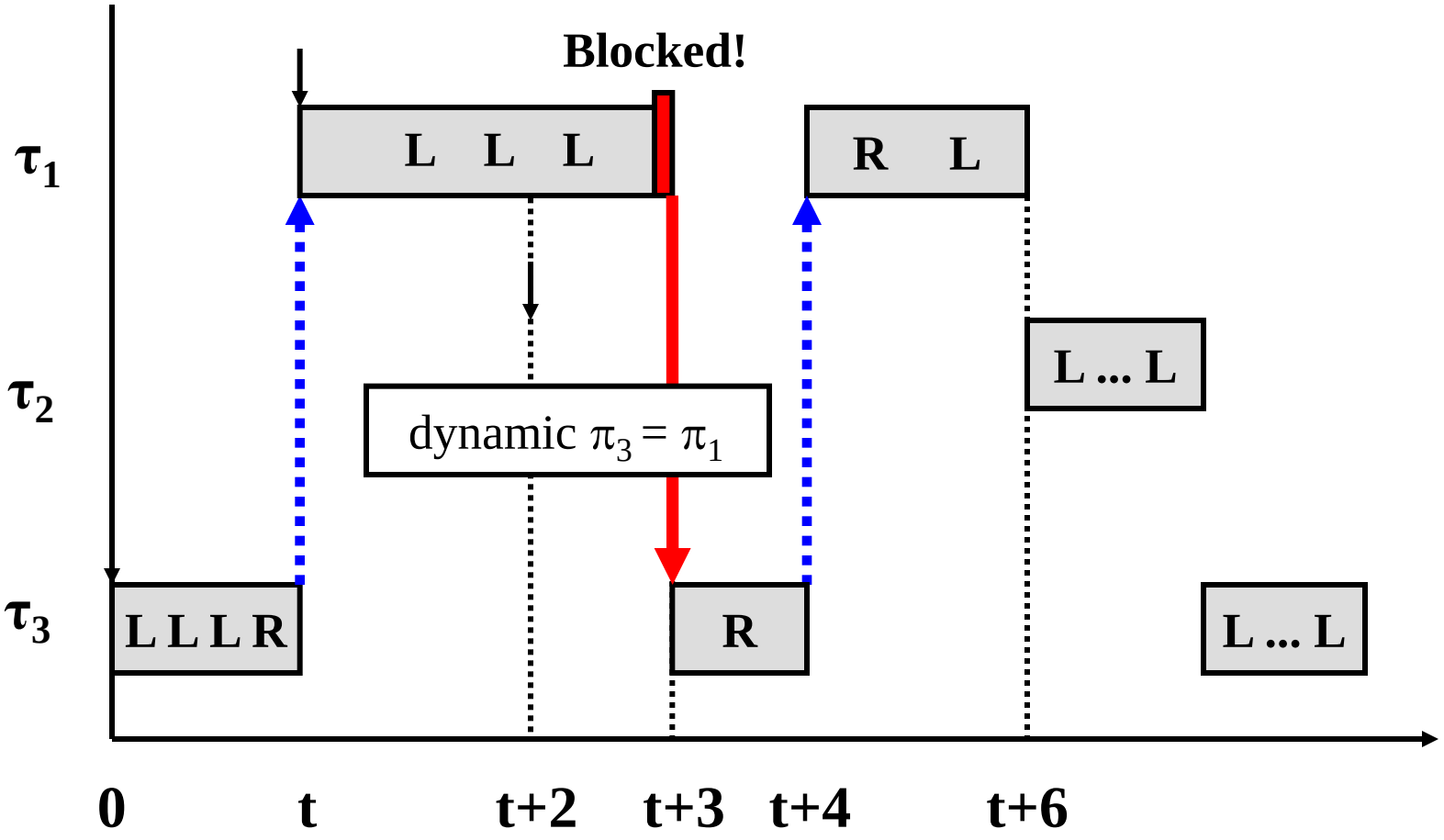
Priority Inheritance Protocols

- Basic Priority Inheritance Protocol
- NonPreemptive Critical Sections (NPCS)
- Basic (Original) Priority Ceiling Protocol
- Stack-Based Priority Ceiling Protocol
- **Analysis**
  - Utilization-Based Test
  - Response Time Analysis

Basic Priority Inheritance Protocol

- For each resource (semaphore), a list of blocked tasks must be stored in a priority queue.
- A task τ_i uses its assigned priority, unless it is in its critical section and blocks some higher priority tasks, in which case, task τ_i uses (**inherits**) the highest dynamic priority of all the tasks it blocks.
- Priority inheritance is **transitive**; that is, if task τ_i blocks τ_j and τ_j blocks τ_k , then τ_i can inherit the priority of τ_k (π_k).

Priority Inheritance

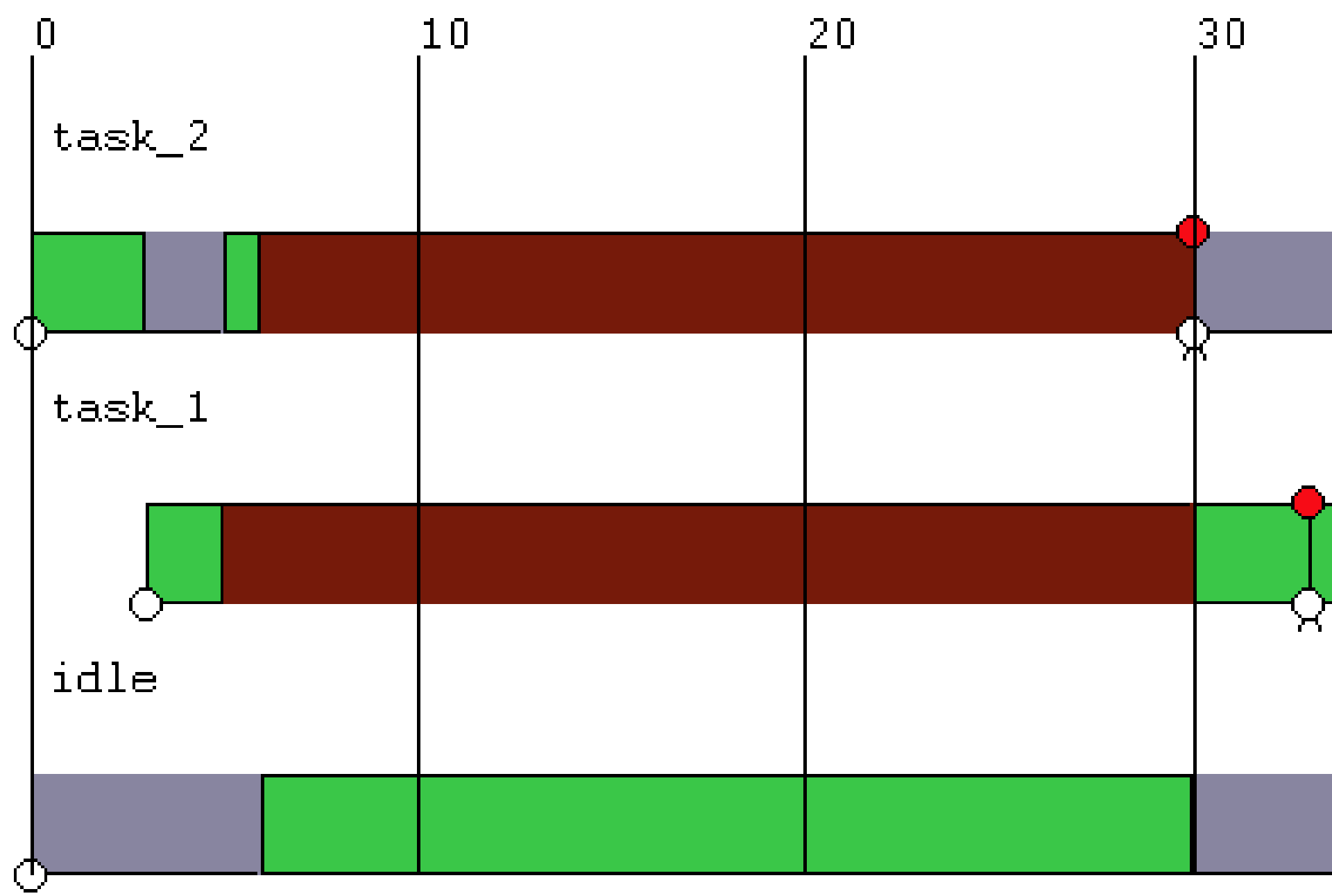


Properties of Priority Inheritance

- Under the basic priority inheritance protocol, if there are *m* semaphores that can block a job J in a task, then J can be blocked at most *m* times; e.g., on each semaphore at most once.
- The Basic **Priority Inheritance Protocol** has two problems:
  - **Deadlock** - two tasks need to access a pair of shared resources simultaneously. If the resources, say A and B, are accessed in opposite orders by each task, then deadlock may occur.
  - **Blocking Chain** - the blocking duration is bounded (at most sum of critical section times), but may be substantial.

```
periodic task_1
   period        30   deadline 30    offset 3
   priority 1
   inheritance::respop(sem1)
   pop(sem1)
   [2,2]
   inheritance::respop(sem2)
   pop(sem2)
   [2,2]
   inheritance::resvop(sem2)
   vop(sem2)
   [1,1]
   inheritance::resvop(sem1)
   vop(sem1)
   [1,1]
endper

periodic task_2
   period        30   deadline 30    offset 0
   priority 2
   [2,2]
   inheritance::respop(sem2)
   pop(sem2)
   [2,2]
   inheritance::respop(sem1)
   pop(sem1)
   [2,2]
   inheritance::resvop(sem1)
   vop(sem1)
   [1,1]
   inheritance::resvop(sem2)
   vop(sem2)
   [1,1]
endper
```


0
10
20
30
task_2
task_1
idle

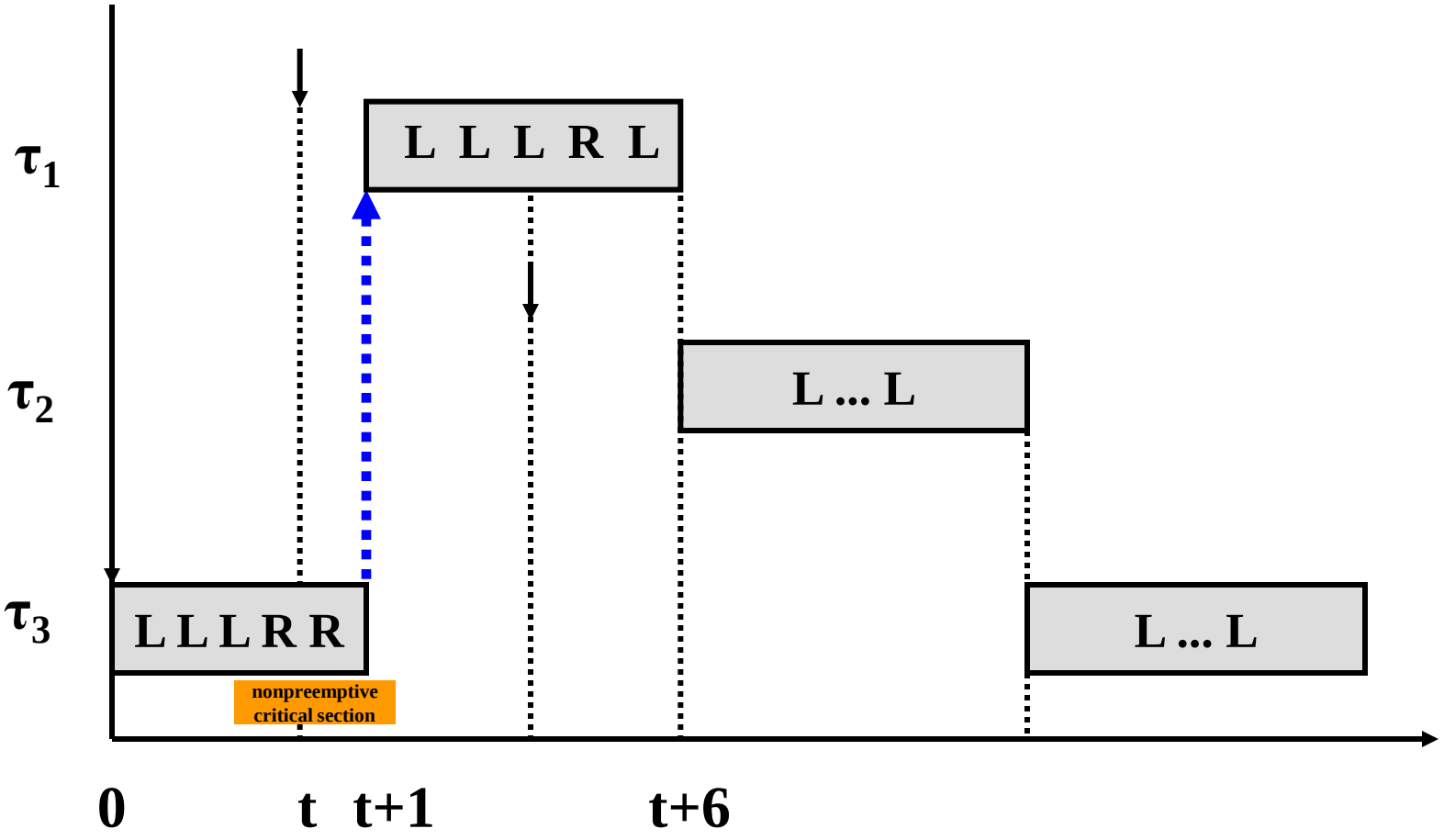
Blocking Chain Example

- Task τ_1 : L R_2 L R_3 L R_4 L ... L R_n L, $\varphi_1 = 2(n-1)$
- Task τ_2 : L R_2 R_2, $\varphi_2 = 2(n-2)$
- Task τ_3 : L R_3 R_3, $\varphi_3 = 2(n-3)$
- Task τ_4 : L R_4 R_4, $\varphi_4 = 2(n-4)$
- ...
- Task τ_{n-1} : L R_{n-1} R_{n-1}, $\varphi_{n-1} = 2$
- Task τ_n : L R_n R_n, $\varphi_n = 0$

NonPreemptive Critical Sections (NPCS)

- All critical sections are executed nonpreemptively.
- When a job requests a resource, it is always allocated the resource.
- When a job holds a resource (in a critical section), it executes at a priority higher than any other task.
- Since no job is ever preempted when it holds a resource, deadlock can never occur.
- The blocking time due to resource conflicts is the maximum critical section time over **all** lower priority tasks.

NonPreemptive Critical Sections



Advantages and Disadvantages

- **Advantages:**
  - Simplicity
  - Use with fixed-priority and dynamic-priority systems
- **Disadvantages:**
  - Every task can be blocked by every lower priority task, even when there is no resource sharing between the tasks.
- Idea behind Priority Ceiling Protocols: Only allow blocking when tasks share resources.

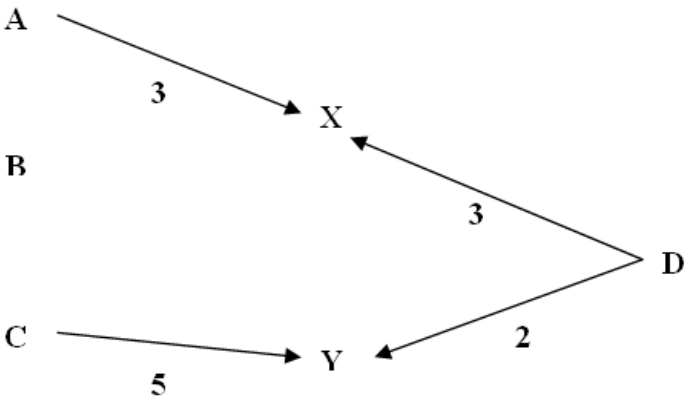
Sample Problem

1. Consider the periodic task set:

   {(A,10,4[X;3]}), (B,20,6), (C,30,5[Y;5]), (D,40,10[X;3[Y;2]])}

   That is, Task A has a period of 10, and a run-time of 4 in which resource X is locked for a total of 3 time units. Further, suppose that the tasks are assigned priorities using a rate monotonic assignment.

   a. Draw the corresponding Task/Resource Graph.



$B_A = 5, B_B = 5, B_C = 3, B_D = 0$

   b. If the resources are accessed in NonPreemptable Critical Sections (**NPCS**), what is the **maximum blocking time** for each task?

Priority Ceiling Protocols (PCP)

- A higher priority task can be blocked **at most once** during each job by a lower priority task.
- Deadlocks are prevented.
- Transitive blocking is prevented.
- Mutually exclusive access to shared resources is supported.

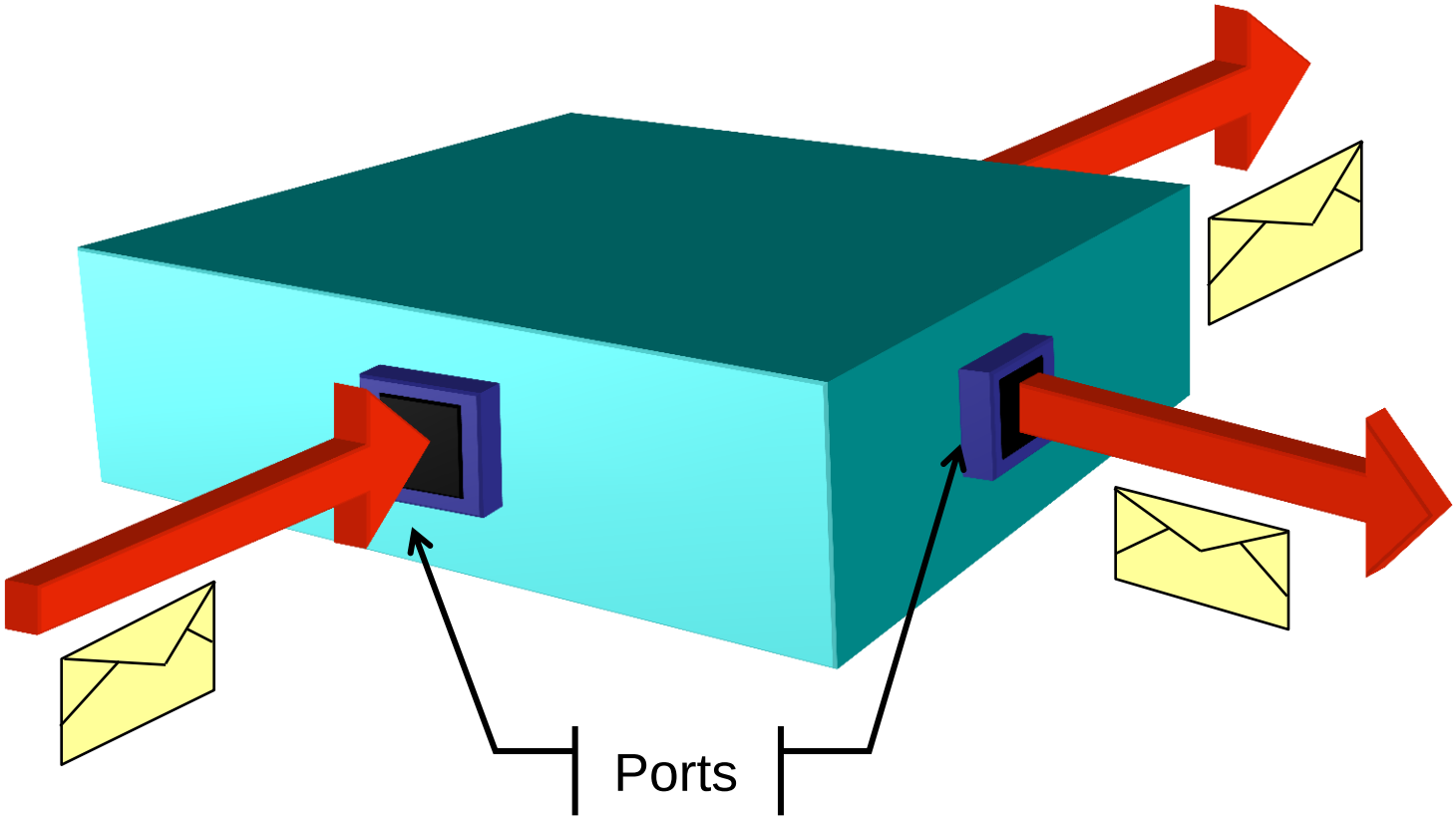
Priority Ceiling Protocols

- Each task has a static default priority.
- Each resource has a static ceiling priority equal to the maximum priority of all tasks that use it.
- A task has a dynamic priority equal to the maximum of its own default priority and any priority it inherits due to **blocking** a higher priority task. The difference is **when does** the dynamic priority change:
  - immediately (before execution) (IPCP), or
  - when blocking occurs (OPCP).

Real-Time UML Constructs

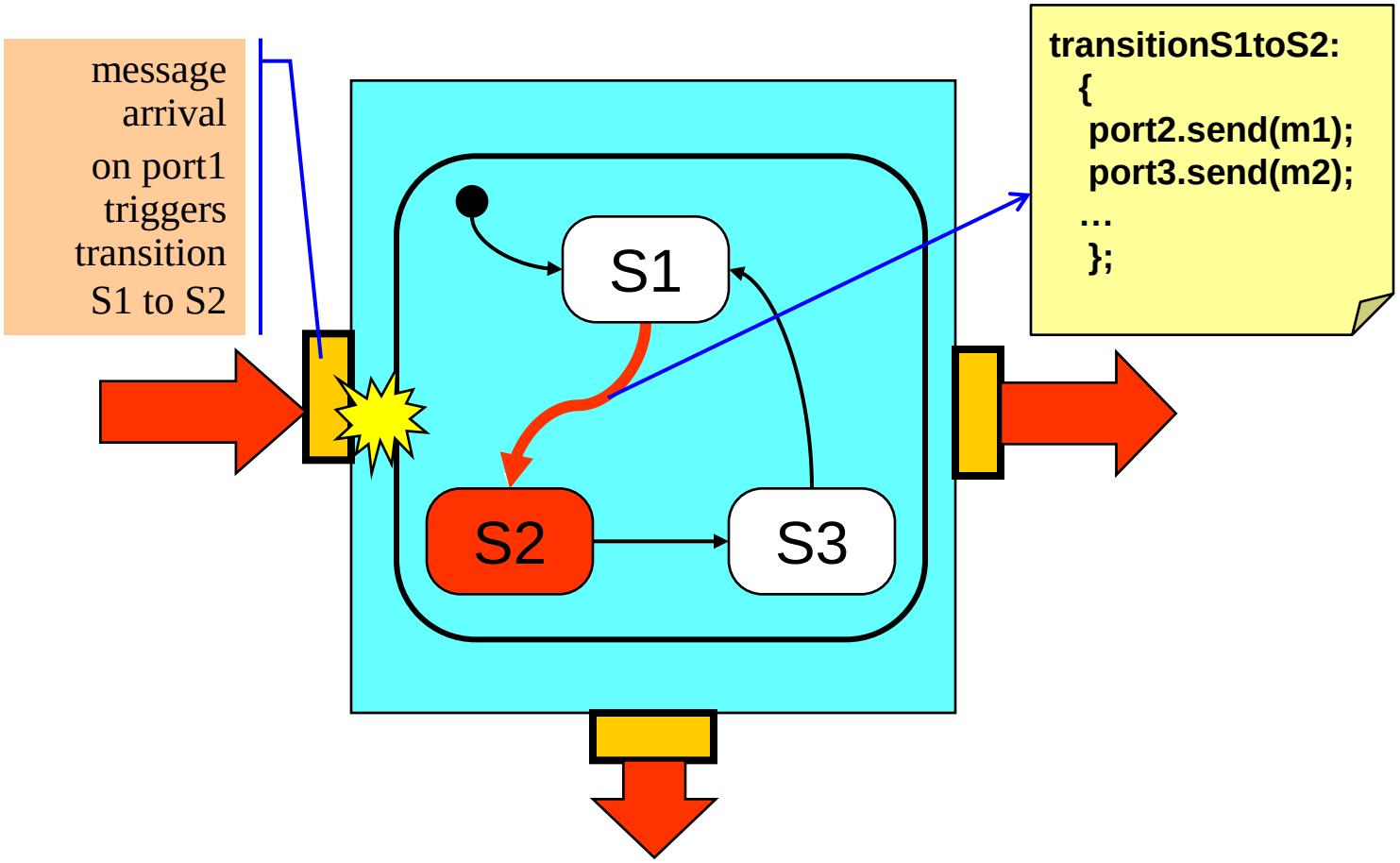
- For Modeling **Structure**
  - capsules (capsule classes)
  - ports
  - connectors
- For Modeling **Behavior**
  - protocols
  - state machines
  - time service

Capsules: Active Objects



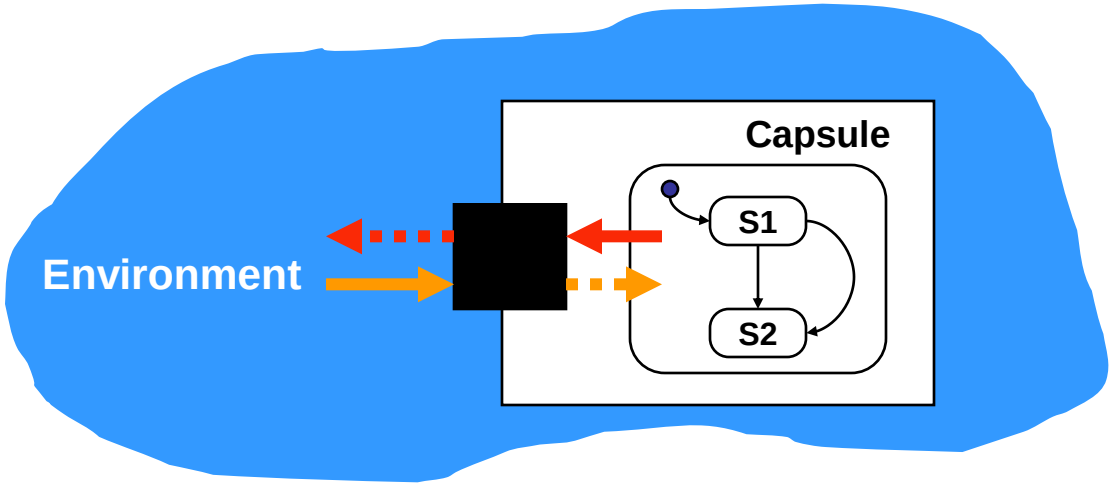
Capsules: Behavior

- Optional hierarchical state machine



Ports: Boundary Objects

- Fully isolate a capsule's implementation from its environment (in both directions)



Ports are created and destroyed along with their capsule

System Ports

- There are four kinds of **system ports** that are used for accessing features of the run-time system:
  - **Frame**: used for dynamic creation, destruction, import and export of capsules at run-time
  - **Timing**: used to access the timing service (setting different timers)
  - **Log**: used to access the log service (printing logging messages)
  - **Exception**: used to access the exception service (ability to define custom policies to recover from exceptions)
- How to access different run-time system features through system ports:
  - create a non-wired port of the system port type required
  - invoke required operation on the port:

portName.functionName (args)

Model Checking

- **Model checking** is an automated technique that, given a finite-state model (M) of a system and a logical property (φ) systematically checks whether this property holds for (a given initial state in) that model [Clarke & Emerson, 1981].
- **Problem:** Although the model is finite-state, the state space of a system typically grows exponentially.
- **SPIN** is one of the most popular model checkers [Holzmann 1991].

Properties to Check

- Deadlock
- Livelock, starvation
- Underspecification – Unexpected reception of messages
- Overspecification – Dead code
- Violations of constraints
  - Buffer overruns
  - Array bounds violations
- Assumptions about speed – Logical correctness versus real-time behavior

Promela

- Promela – **Pro**cess/Protocol **Me**ta **La**nguage
  - Provides a language similar to the C programming language
  - Provides a guarded command language to model finite-state systems
  - Supports dynamic creation of concurrent processes
  - Supports messages channels between processes

SPIN

- **SPIN** – **S**imple **P**romela **In**terpreter
  - A state-of-the-art model checking tool used to check the logical consistency of concurrent systems described using the modelling language Promela.
  - Designed specifically for checking data communication protocols.

Example: Alternating Bit Protocol

```
mtype {MSG, ACK};
chan  toSender = [2] of {mtype, bit};
chan  toReceiver = [2] of {mtype, bit};
proctype Sender(chan in, out)
{
    bit sendbit, recvbit;
    do
        :: out ! MSG, sendbit ->
            in ? ACK, recvbit;
            if
            :: recvbit = = sendbit ->
                sendbit = 1-sendbit
            :: else -> skip
            fi
    od
}
```

Example: Alternating Bit Protocol

```
proctype Receiver(chan in, out)
{
    bit recvbit;
    do
        :: in ? MSG, recvbit ->
                out ! ACK, recvbit;
        :: timeout ->
                out ! ACK, recvbit;
    od
}
init
{
    run  Sender(toSender, toReceiver);
    run  Receiver(toReceiver, toSender);
}
```

Promela Model

- A **Promela Model** consists of the following elements:
  - **type** declarations – message types, typedefs, and constants
  - **channel** declarations – message channels
  - **global variable** declarations
  - **process** declarations
  - [**init** process] – initializes variables and starts processes

Verification versus Testing

- Testing starts with a set of possible test cases, simulates the system on each input, and observes the behavior. In general, testing does not cover all possible executions.
- On the other hand, verification establishes correctness for **all** possible execution sequences.

Model Checking

- **Problem:** The number of states can be very large.
- **Two Phases:**
  1. **Create a model:** some approximation of the system under construction; e.g., use a finite state model you need to model as well as its environment.
  2. **Verify the model:** determine the properties you want to verify, and check whether the model satisfies the properties.
- The verification exercise is only as good as the model.

Mutual Exclusion
(Incorrect Solution)

```
bit x1 = 0; /* used to indicate that process 1 wants in cs */
bit x2 = 0; /* used to indicate that process 2 wants in cs */
int mutex = 0;  /* used to count number of processes in cs */

proctype P1()
{
    x1 = 1;
    x2 == 0;
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x1 = 0;
}

proctype P2()
{
    x2 = 1;
    x1 == 0;
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x2 = 0;
}
```

```
proctype monitor()
{
    assert(mutex!=2);
}

init
{
    run P1();
    run P2();
    run monitor();
}
```

Linear Temporal Logic (LTL)

- LTL formulae are used to specify temporal properties.
- LTL includes propositional logic and temporal operators:
  - **[ ]P =** always P
  - **<>P =** eventually P
  - **P U Q =** P is true until Q becomes true
- **Examples:**
  - Invariance: **[ ] (p)**
  - Response: **[ ] ((p) -> (<> (q)))**
  - Precedence: **[ ] ((p) -> ((q) U (r)))**
  - Objective: **[ ] ((p) -> <>((q) || (r)))**

Temporal Claims in SPIN

- The most powerful method for expressing correctness requirements in Promela models is to use a “never” claim.
- Syntax: **never { … body … }**
- The body of a never claim expresses behavior that is claimed to be impossible.
- A correctness violation occurs if and only if a temporal claim can be completely matched by a system behavior.
- There can be only one never claim in a model.
- Temporal claims do not specify independent system behavior, they only formalize claims about existing system behavior; e.g., the system behavior does not change when a never claim is added.

Rules for Never Claims

- Every “statement” in a temporal claim must model a proposition with no side-effects; e.g., no assignments, receive, or send statements.
- Statements do not define system behavior, they are only used to monitor system behavior.
- To violate a claim, the series of propositions listed in a temporal claim must match the system behavior at every single step of execution.

UPPAAL Components

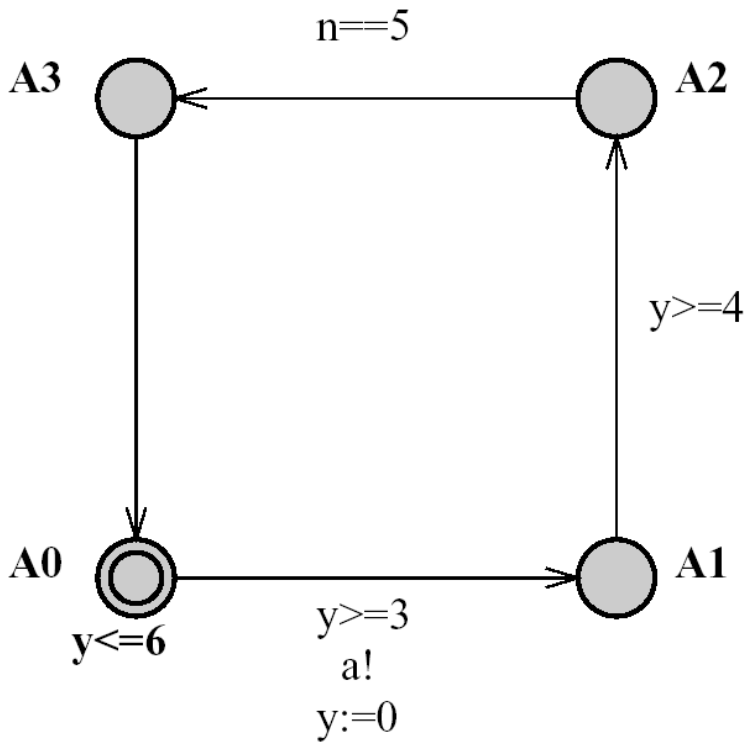
- **UPPAAL** consists of three main parts:
  - a **description language**,
  - a **simulator**, and
  - a **model checker**.
- The **description language** is a non-deterministic guarded command language with data types. It can be used to describe a system as a *network of timed automata* using either a graphical (*.atg, *.xml) or textual (*.xta) format.
- The **simulator** enables examination of *possible* dynamic executions of a system during the early modeling stages.
- The **model checker** exhaustively checks *all* possible states.

Example – .xta file format

(from UPPAAL in a Nutshell)

```
clock x, y;
int n;
chan a;

process A {
  state A0 { y<=6 }, A1, A2, A3;
  init A0;
  trans A0 -> A1 {
    guard y>=3;
    sync a!;
    assign y:=0;
  },
  A1 -> A2 {
    guard y>=4;
  },
  A2 -> A3 {
  guard n==5;
  }, A3 -> A0;
}
```

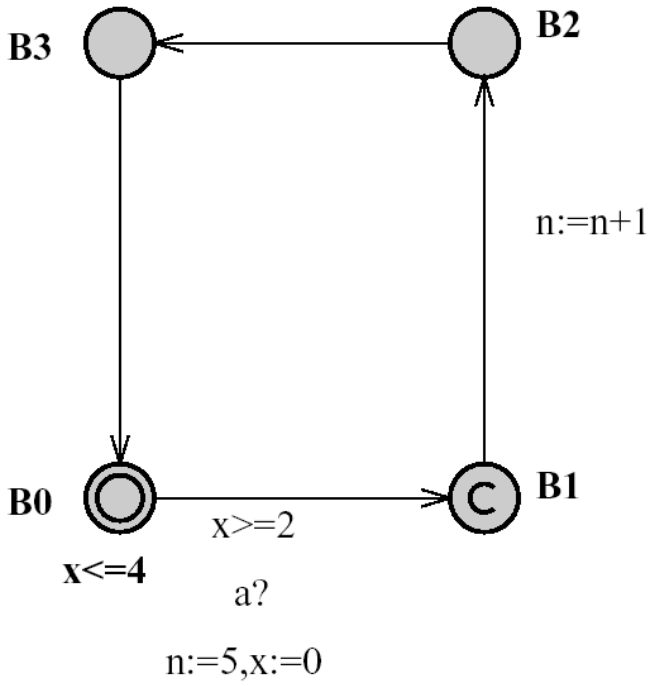


Example (cont.)

(from UPPAAL in a Nutshell)

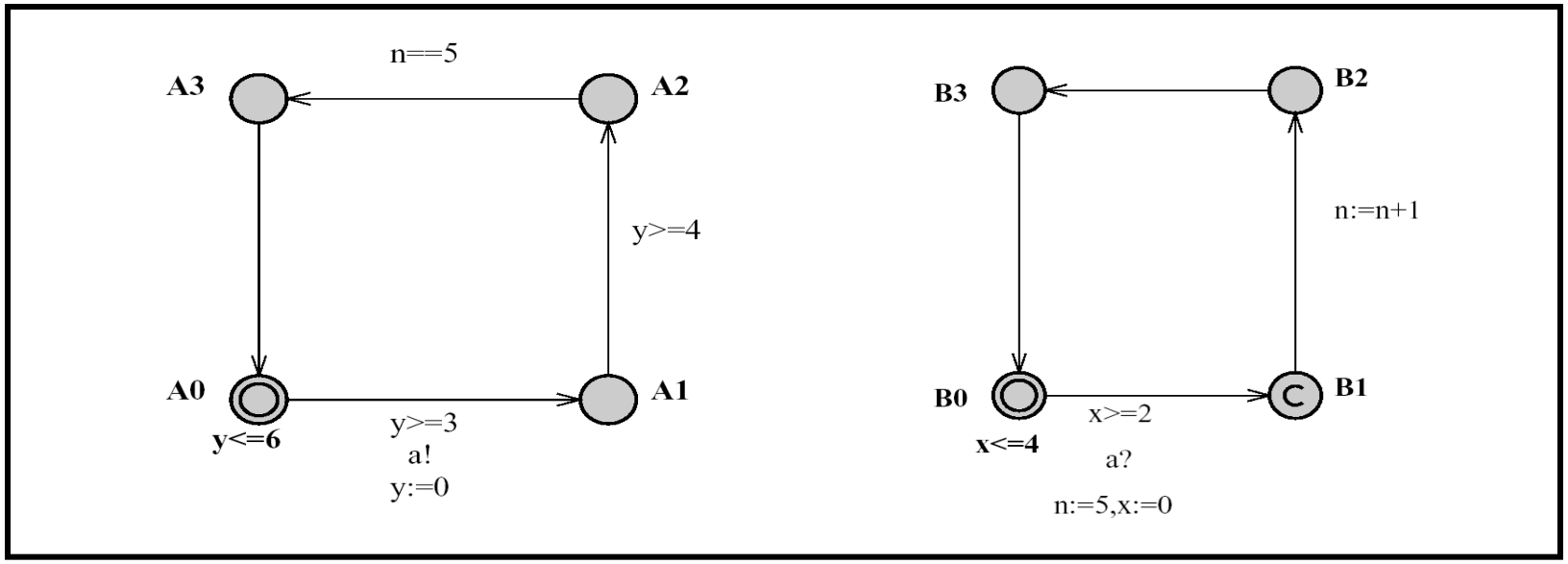
```
process B {
  state B0 { x<=4 }, B1, B2, B3;
  commit B1;
  init B0;
  trans B0 -> B1 {
    guard x>=2;
    sync a?;
    assign n:=5,x:=0;
  },
  B1 -> B2 {
    assign n:=n+1;
  },
  B2 -> B3 {
  }, B3 -> B0;
}

system A, B;
```

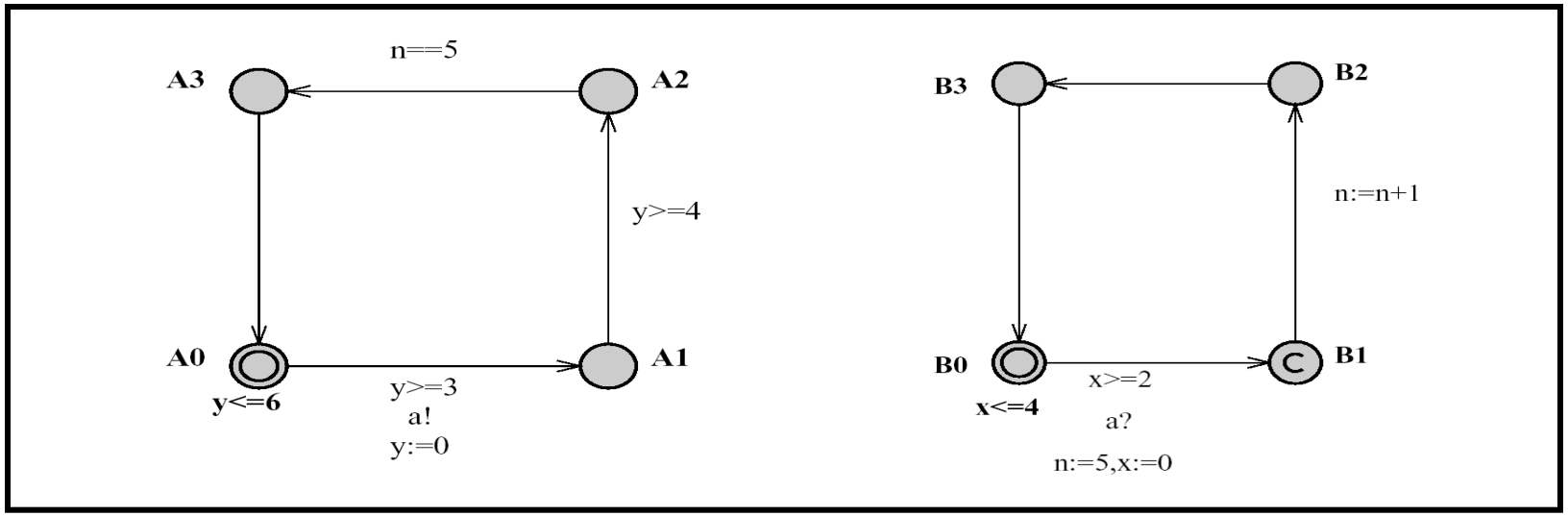


Transitions

- **Delay transitions** – if none of the invariants of the nodes in the current state are violated, time may progress without making a transition; e.g., from $((A_0,B_0),x=0,y=0,n=0)$, time may elapse 3.5 units to $((A_0,B_0),x=3.5,y=3.5,n=0)$, but time cannot elapse 5 time units because that would violate the invariant on B_0.



Transitions (cont.)



- **Action transitions** – if two complementary edges of two different components are enabled in a state, then they can synchronize; also, if a component has an enabled internal edge, the edge can be taken without any synchronizaton; e.g., from $((A_0,B_0),x=0,y=0,n=0)$ the two components can synchronize to $((A_1,B_1),x=0,y=0,n=5)$.

Example

(from UPPAAL2k: Small Tutorial)

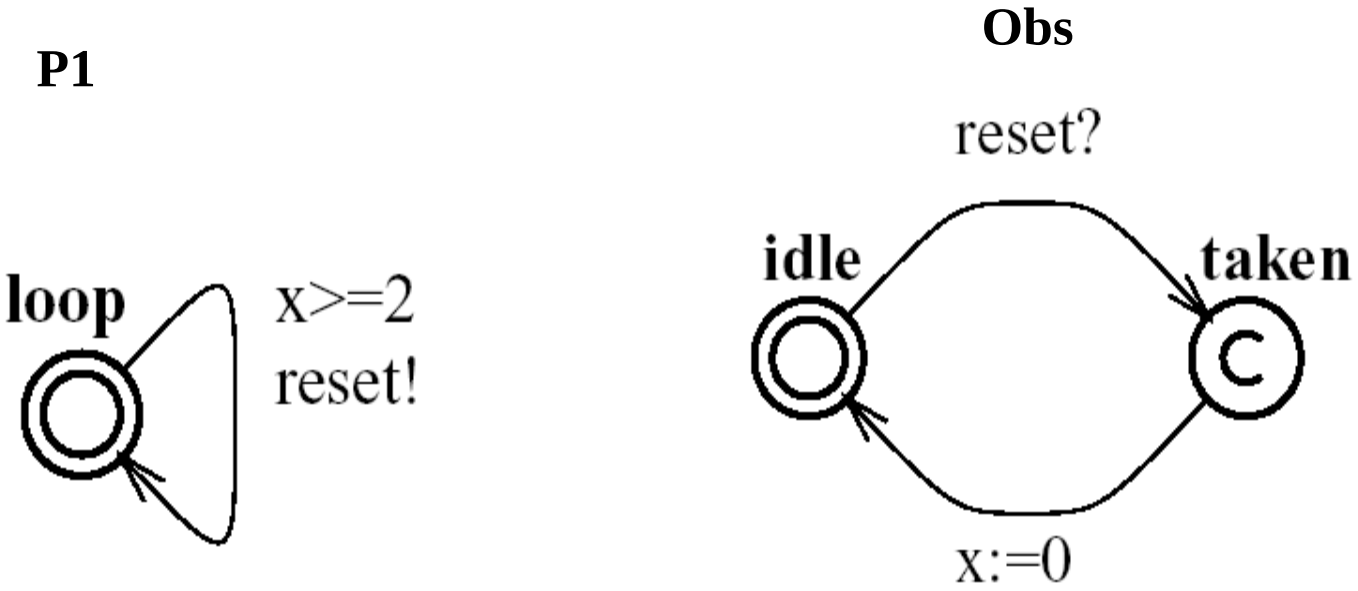


Figure 5: First example with the observer.

Verification (example.xta)

```
int x:=0;
process P{
state S0;
init S0;
trans S0 -> S0{guard x<2000; assign x:=x+1; };
}
process Q{
state S1;
init S1;
trans S1 -> S1{guard x>0; assign x:=x-1; };
}
process R{
state S2;
init S2;
trans S2 -> S2{guard x==0; assign x:=0; };
}
p1:=P();
q1:=Q();
r1:=R();
system p1,q1,r1;
```

Int x

Process P
do
:: x<2000 → x:=x+1
od

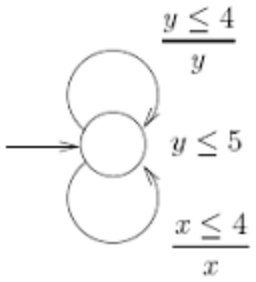
Process Q
do
:: x>0 → x:=x-1
od

Process R
do
:: x=2000 → x:=0
od

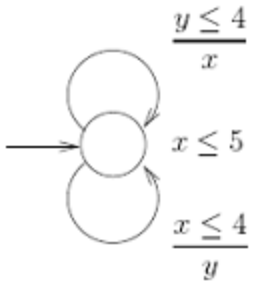
fork P; **fork** Q; **fork** R

Sample Problem

2. Consider the following two timed automata:



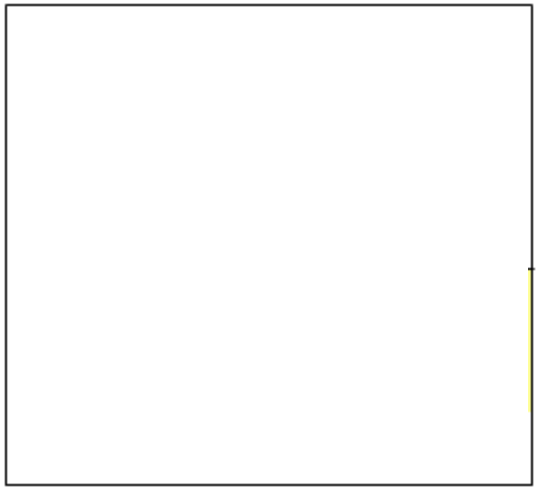
(a)



(b)

As these automata have a single location only, the state of these automata can be viewed as a point in the real plane. A point (d, e), with non-negative d and e, can be used to denote clock x equals d and clock y equals e.

a. Determine the reachable state space for each automata given that the clocks are **local** to each automata.

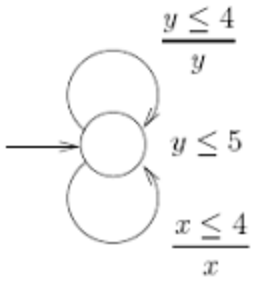


(a)

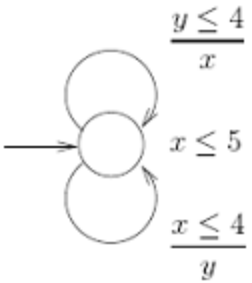
(b)

Sample Problem

2. Consider the following two timed automata:



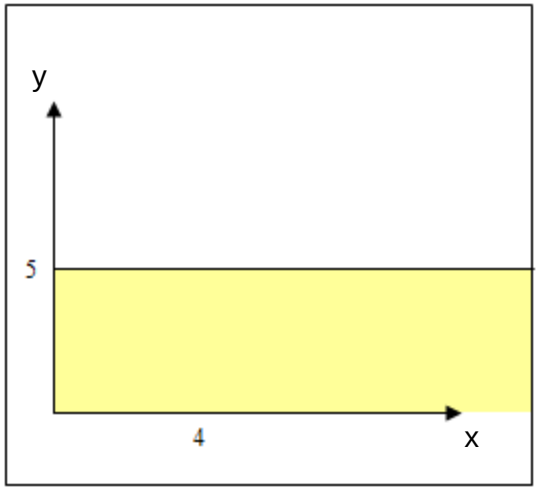
(a)



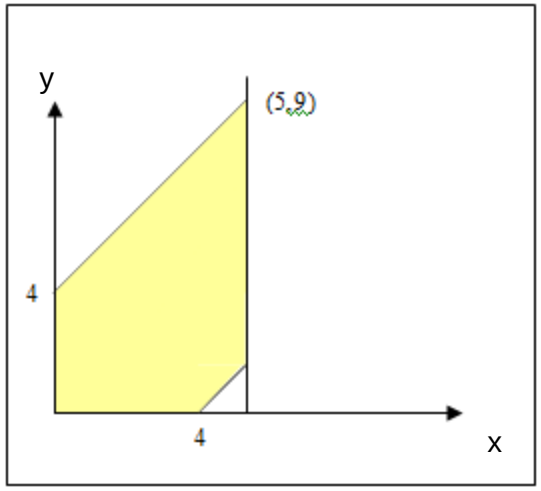
(b)

As these automata have a single location only, the state of these automata can be viewed as a point in the real plane. A point (d, e), with non-negative d and e, can be used to denote clock x equals d and clock y equals e.

a. Determine the reachable state space for each automata given that the clocks are **local** to each automata.



(a)

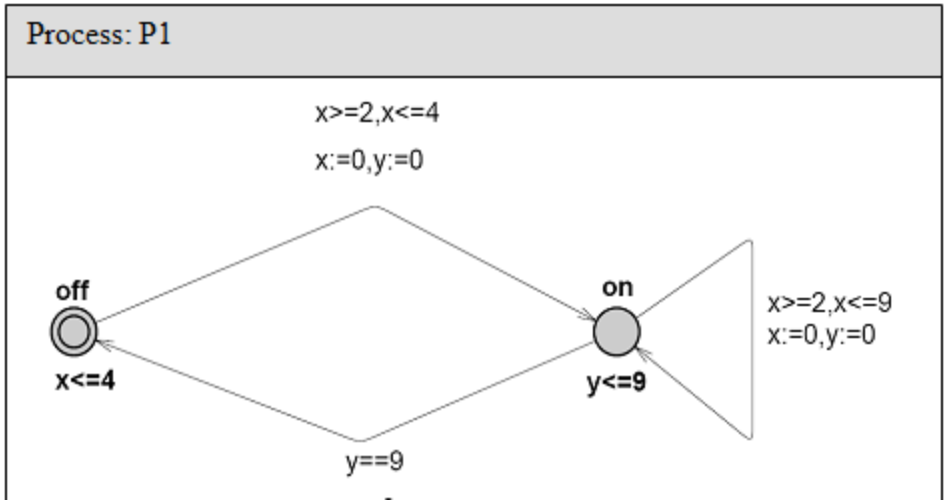


(b)

global.

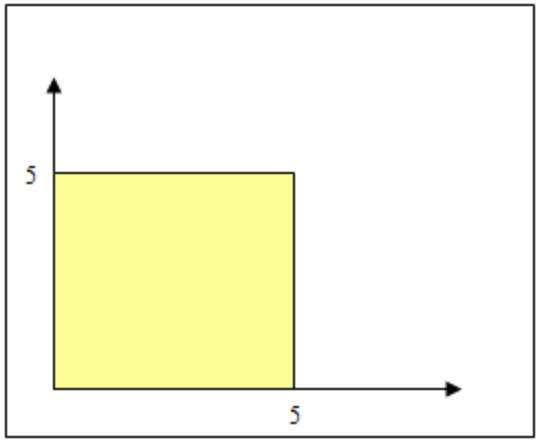
Hint: just create a 2-d graph of the regions denoting the reachable state space.

3. Consider the following UPPAAL model:



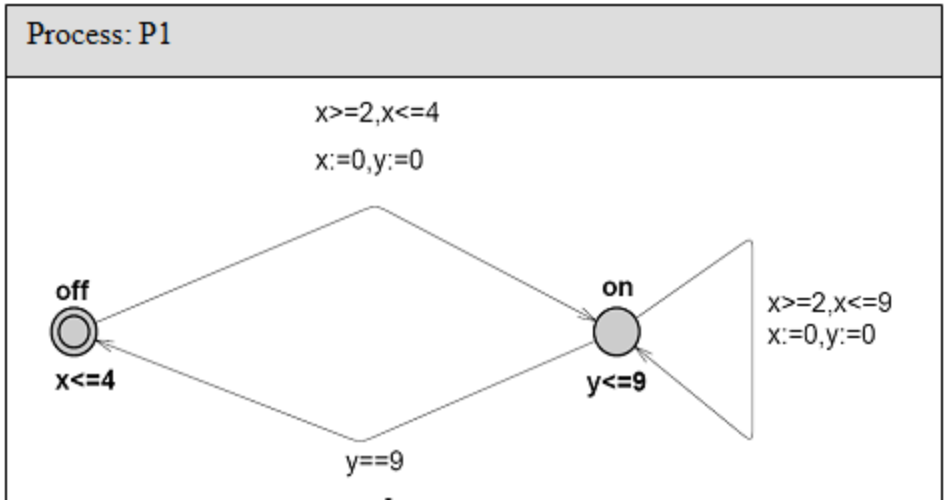
Full Screen
Close Full Screen

global.

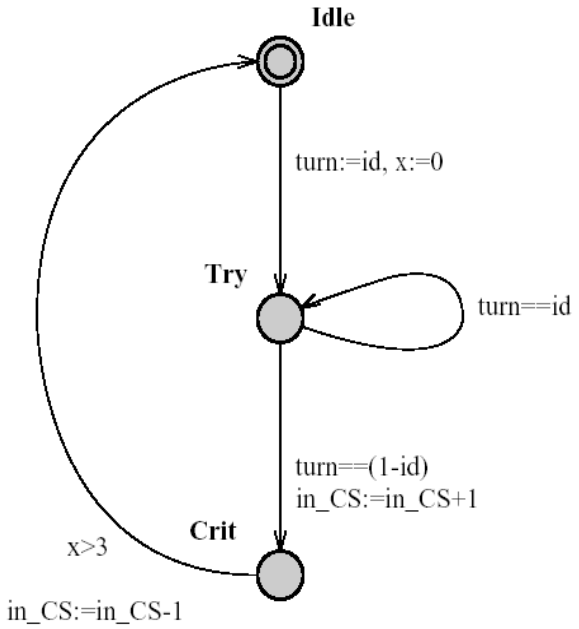


Hint: just create a 2-d graph of the regions denoting the reachable state space.

3. Consider the following UPPAAL model:



Example (mutex2.xta)




```
//Global declarations
int turn;
int in_CS;

//Process template
process P(const id){
clock x;
state Idle, Try, Crit;
init Idle;
trans Idle -> Try{assign turn:=id, x:=0; },
Try -> Crit{guard turn==(1-id); assign in_CS:=in_CS+1; },
Try -> Try{guard turn==id; },
Crit -> Idle{guard x>3; assign in_CS:=in_CS-1; };
}

//Process assignments
P1:=P(1);
P2:=P(0);

//System definition.
system P1, P2;
```

Promela Model Question

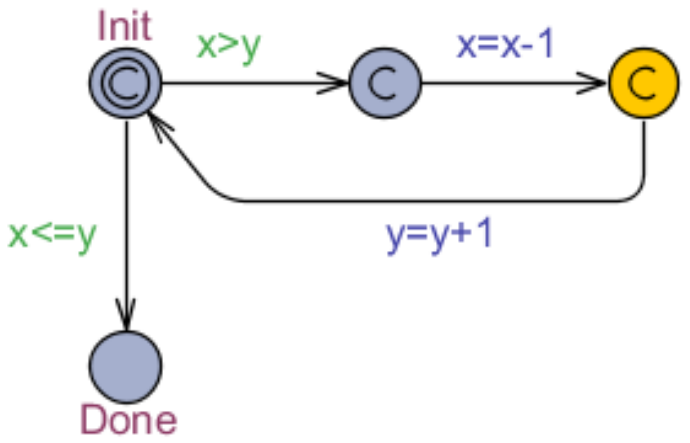
Consider the following Promela model:

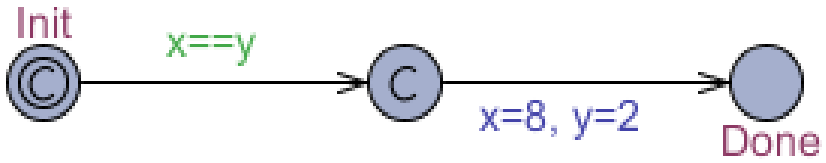
```
int x,y;

proctype p() {
  do
    :: (x > y) -> x=x-1; y=y+1;
    :: else -> break;
  od;
}

proctype q() {
  (x == y) ;
  atomic { x=8; y=2};
}
 init {
  atomic { x=0; y=0; run p(); run q();
}
```

a) Proctype **p()** can be modeled in UPAAL using the following template, draw an equivalent UPAAL template for proctype **q()**, mark the initial state **Init**, and the final state **Done**.





Promela Model Question

(b) Will the processes terminate; e.g., is the property `A<>(p.Done and q.Done)` satisfied? Explain briefly.

(c) What are the possible final values for `x` and `y`?

(d) What if the atomic statement is removed around `{ x=8; y=2; }` ? Does your model for proctype `q()` need to be changed? If so, draw the new model below. Also, in the following list, circle all of the possible final values for `x` and `y`:

- `x=8, y=2`
- `x=8, y=0`
- `x=4, y=5`
- `x=5, y=5`
- `x=4, y=2`
- `x=3, y=3`

Solution

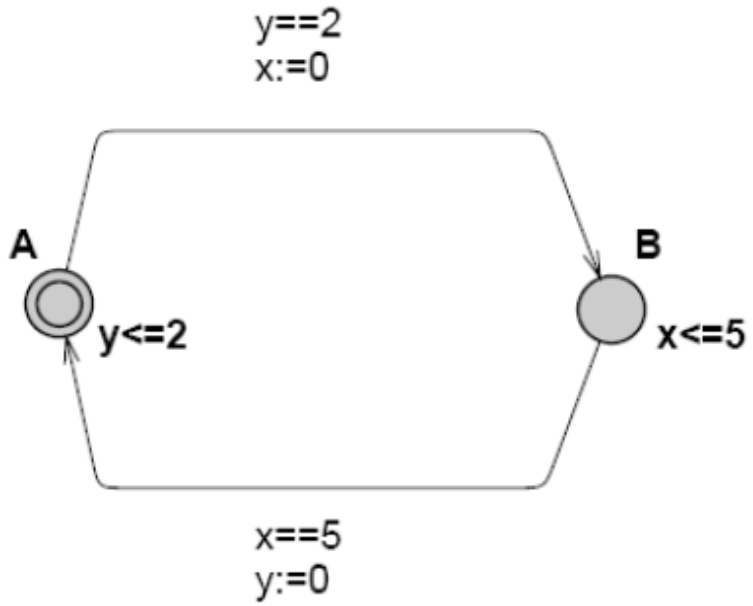
(b) Will the processes terminate; e.g., is the property `A<>(p.Done and q.Done)` satisfied? Explain briefly. Yes

(c) What are the possible final values for `x` and `y`? x=5,y=5, or x=8,y=2

(d) What if the atomic statement is removed around `{ x=8; y=2; }` ? Does your model for proctype `q()` need to be changed? If so, draw the new model below. Also, in the following list, circle all of the possible final values for `x` and `y`: Yes, add another state and transition, the existing transition for x=8, and the new next transition for y=2.

- `x=8, y=2`
- `x=8, y=0`
- `x=4, y=5`
- `x=5, y=5`
- `x=4, y=2`
- `x=3, y=3`

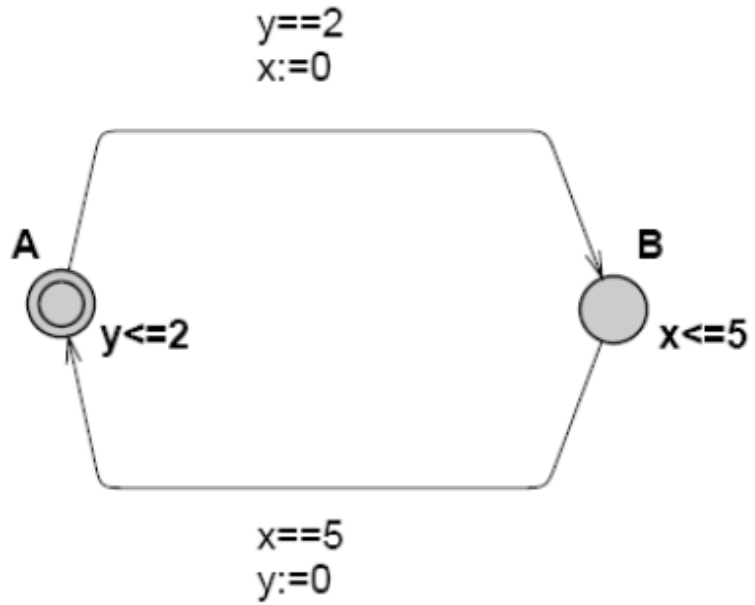
2. Consider the following timed automaton P:



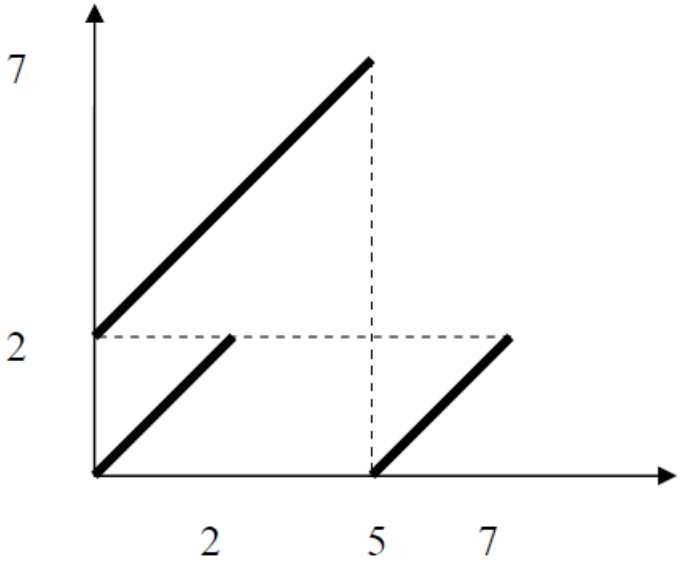
As this automaton has two clocks, x and y, the reachable state space with respect to the clocks can be viewed as a point in a two-dimensional Cartesian plane, one axis for clock x and one axis for clock y. A point (d, e), with non-negative d and e, can be used to denote that clock x equals d and clock y equals e.

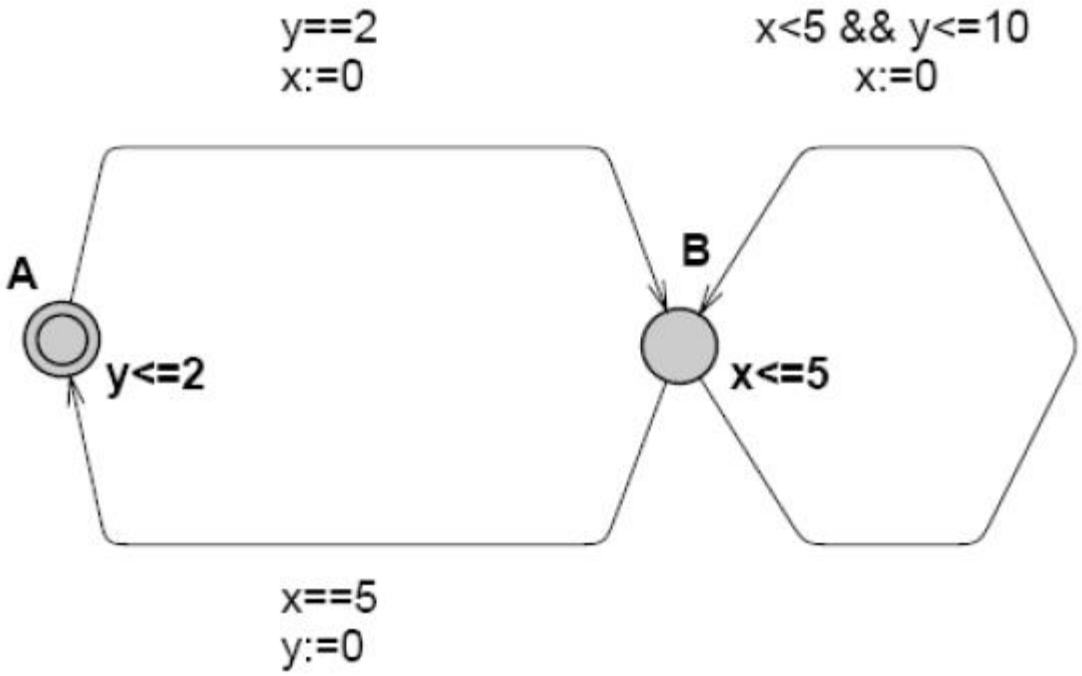
(a) Determine the reachable state space for this automaton; e.g., draw a 2-d graph below:

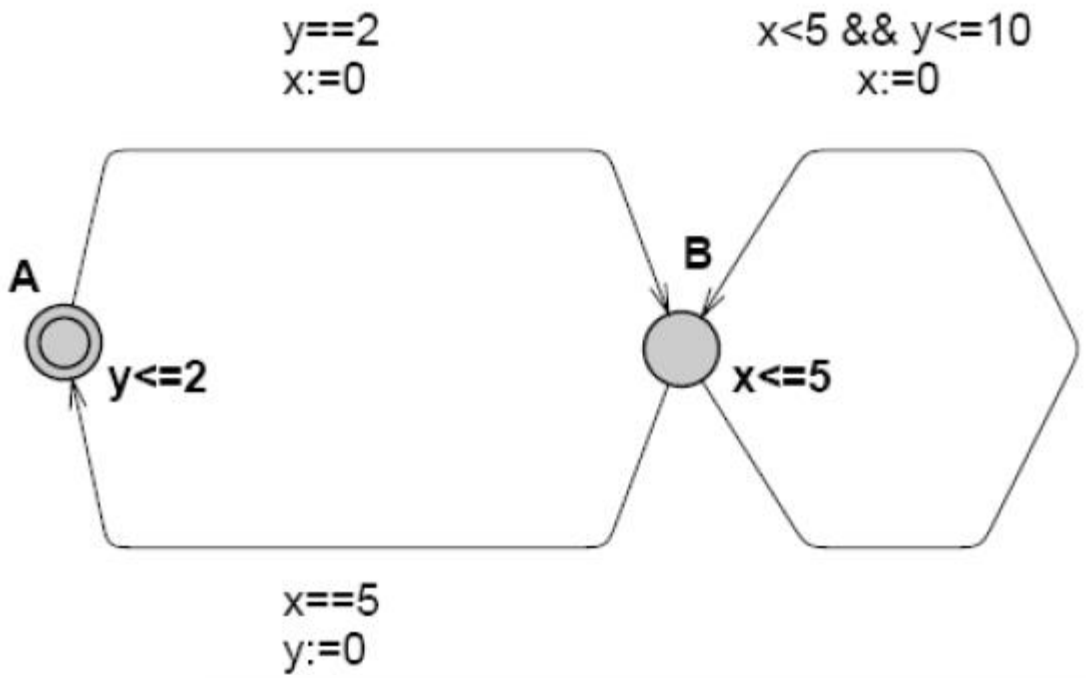
2. Consider the following timed automaton P:

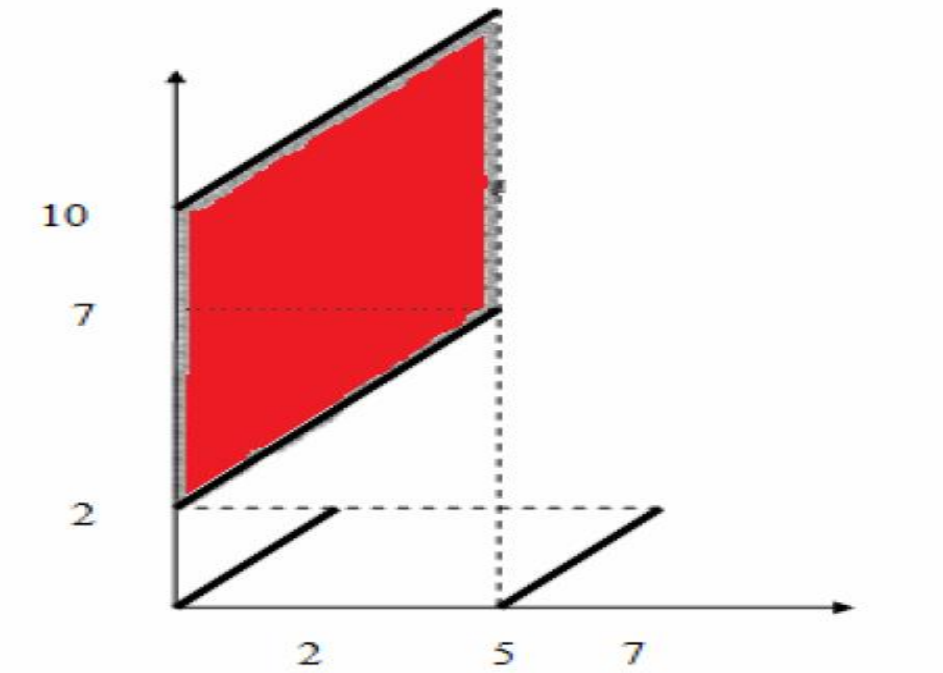


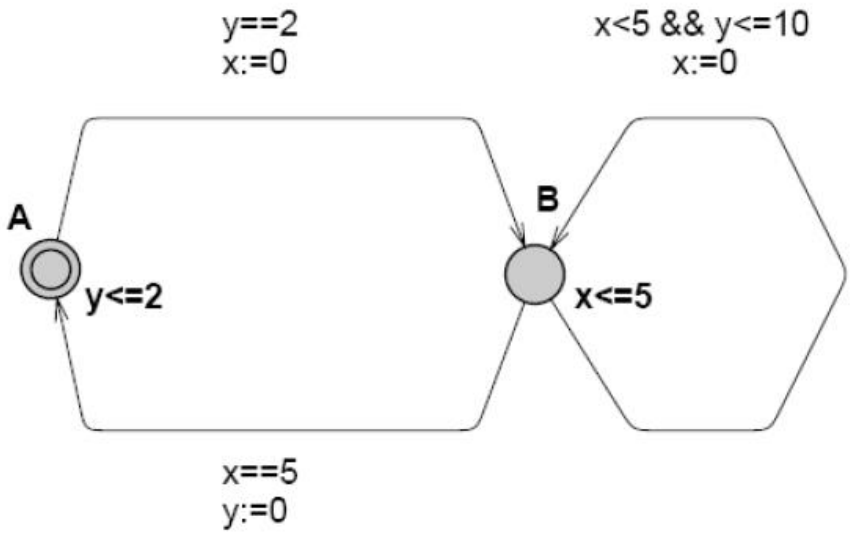
(a) Determine the reachable state space for this automaton; e.g., draw a 2-d graph below:




y==2
x:=0
x<5 && y<=10
x:=0
A
y<=2
B
x<=5
x==5
y:=0


y==2
x:=0
x<5 && y<=10
x:=0
A
y<=2
B
x<=5
x==5
y:=0


10
7
2
2
5
7



(c) Determine which of the following properties are satisfied for the automaton shown above in part (b):

- **A[] not deadlock (RED = satisfied)**
- **P.A --> P.B**
- P.B --> P.A
- **E<> (x==5 and y==7)**
- E<> (x==5 and y==3)
- **E<> (x==2 and y==12)**

The only one that is counter-intuitive is why isn't P.B --> P.A satisfied? Clearly, if an automaton is in state P.B, then once the clock y is over 10, the x clock will advance to 5 and force the transition to P.A. The reason the "leads to" operator is not satisfied here is because we could have an INFINITE number of transitions in the right loop before taking the transition back to P.A.

Summary

- Quiz #2
  - Dec. 2, in class
  - Open Book, Open Notes