Modeling in UPPAAL

Example and solution

dr. Tamás Bartha

BME Department of Control for
Transportation and Vehicle Systems

Contents

- This lecture presents a task *(that is harder than a typical homework)* and explains how to solve it
- Furthermore some useful modeling practices of UPPAAL are also presented:
  - Generating and using random values
  - Modeling atomic operations
  - Modeling synchronous communication
    - Using a global shared variable
    - Using dedicated arrays of channels
  - Reducing state space by removing temporary variables
  - Using data structures and functions
  - Writing and checking temporal logic expressions

Warmup

Solving a simple exercise

Warmup exercise

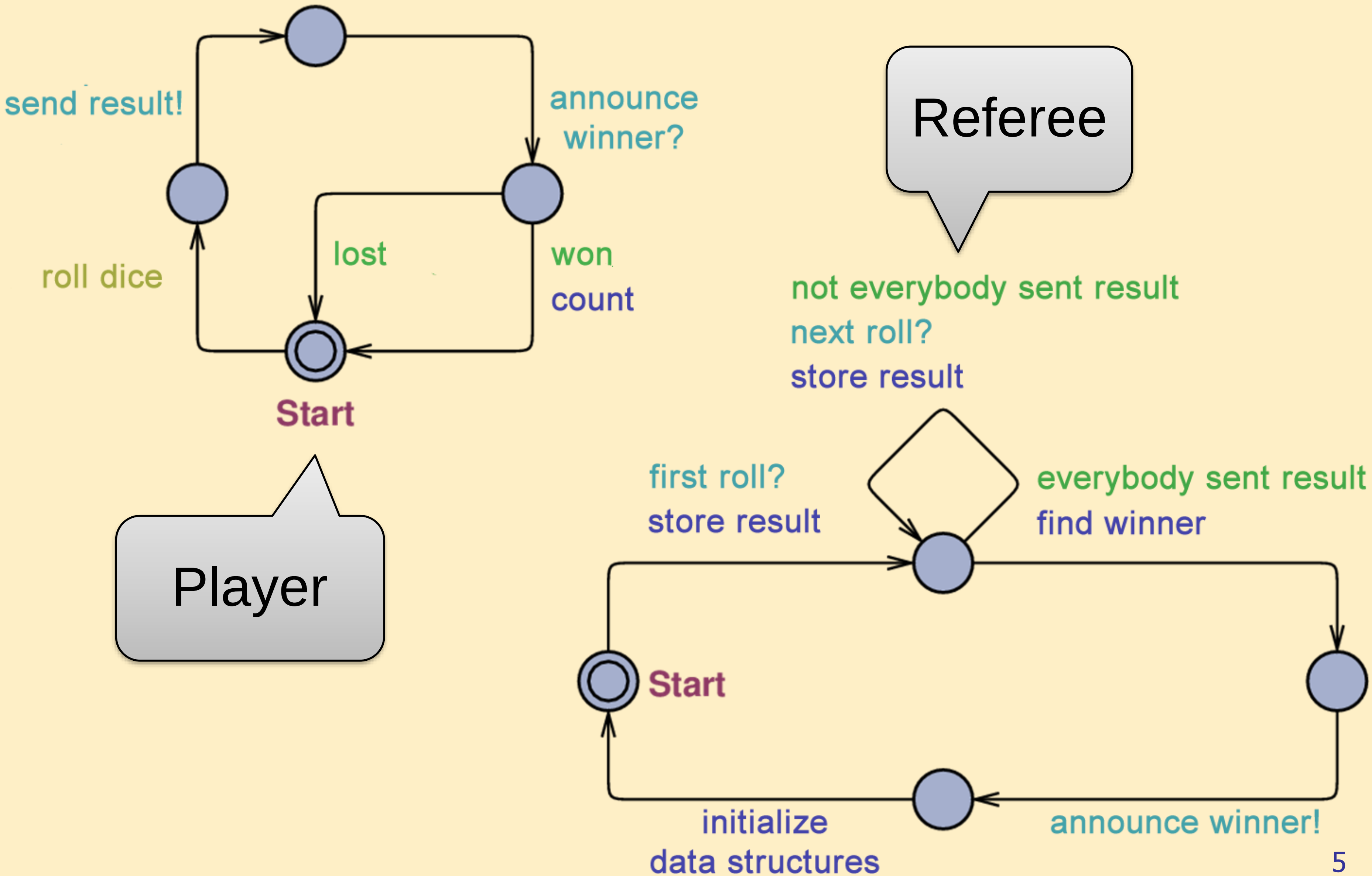
The exercise

- Rolling a dice
  - *n* players, 1 referee
  - Each player rolls a dice once
  - They tell the result to the referee
  - The referee
    - Stores the results
    - Finds the largest result(s)
    - Announces the winner(s)
  - Players count the number of their winning results

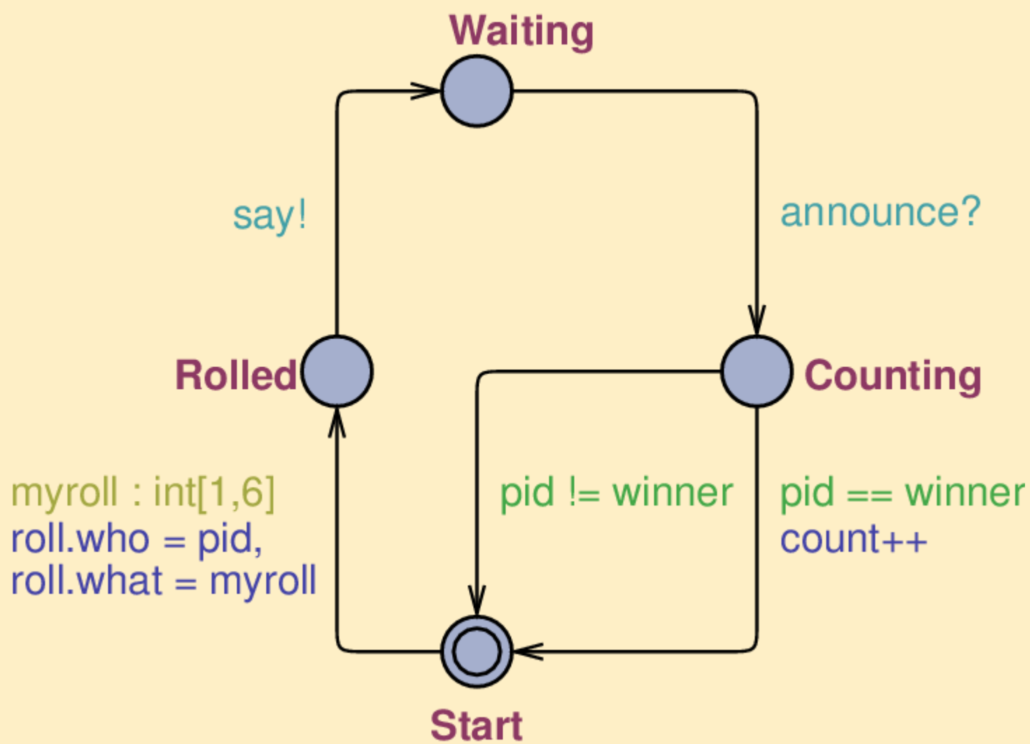
What do we have to solve?

- Generate random value
- Communication
  - "Pass" values
  - Broadcast communication
  - Handling channel arrays
  - Ordering of update sections
- Data structures
- Functions
- Concurrency and timing
- Model checking

Basic idea for the solution



Solution: System and the player



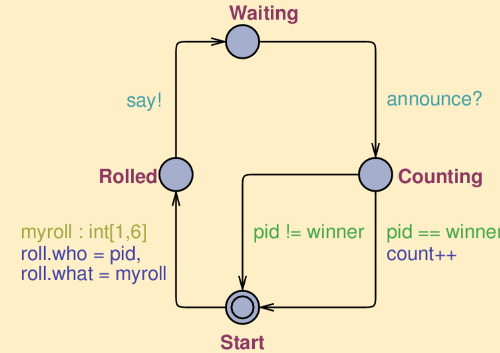
Player:

```
Player(id_t pid)

int[0,wins] count = 0;
clock x;
```

System:

```
system Player, Referee;

const int players = 3;
const int wins = 10;

typedef int[0,players-1] id_t;
typedef int[0,6] dice_t;
struct {
  id_t who;
  dice_t what;
} roll;

id_t winner;
chan say;
broadcast chan announce;
```

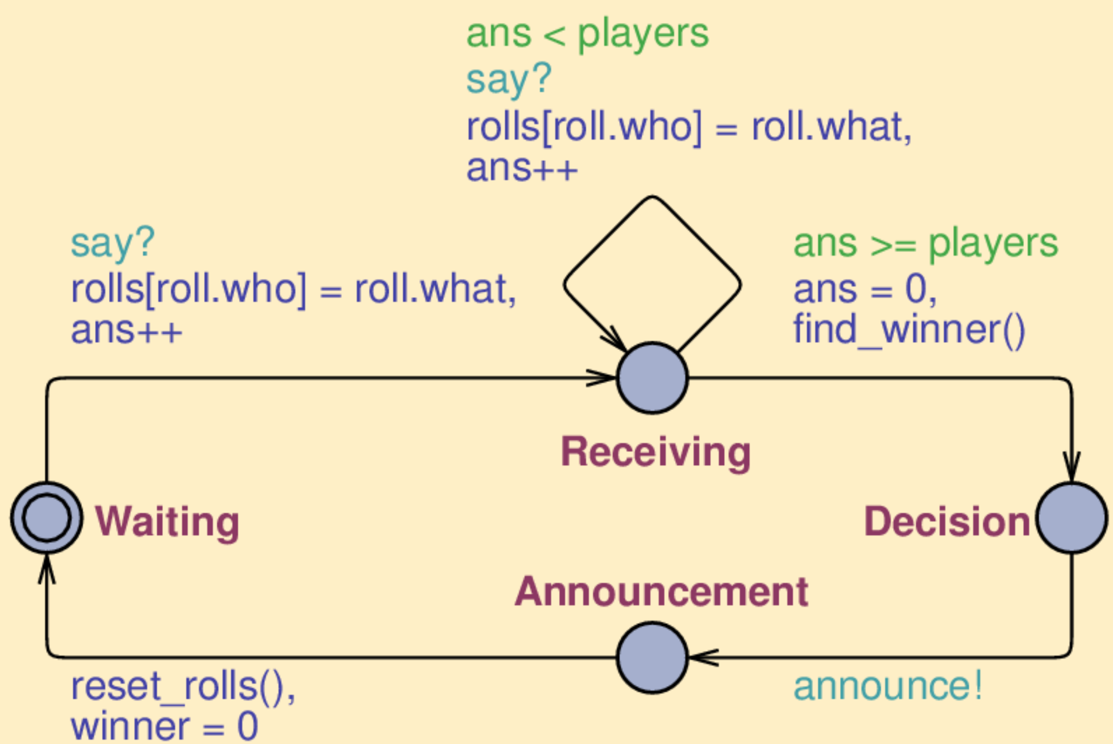
Solution: Referee



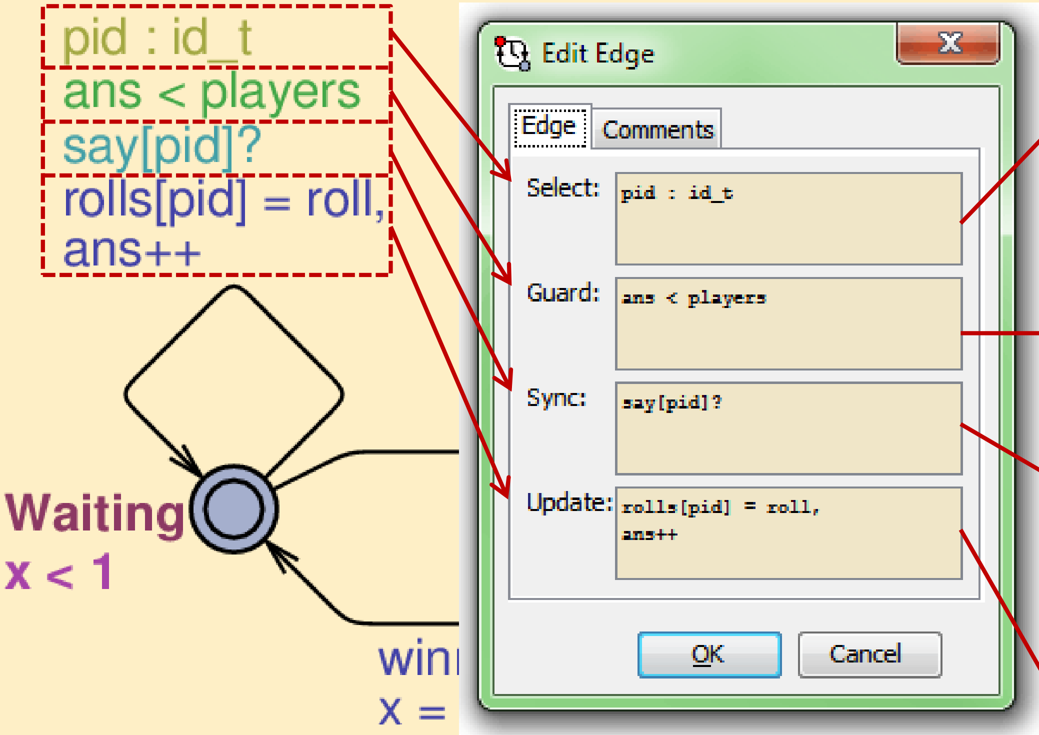
Referee:

```
int [0,players] ans = 0;
dice_t rolls[id_t];
dice_t best = 0;

clock x;

void find_winner() {
  int[0,players] i;

  for (i = 0; i < players; i++) {
    if (rolls[i] > best) {
      best = rolls[i];
      winner = i;
    }
  }
  best = 0;
}
```

```
void reset_rolls() {
  int[0,players] i;

  for (i = 0; i < players; i++) rolls[i] = 0;
}
```



Outlook: Arc expressions



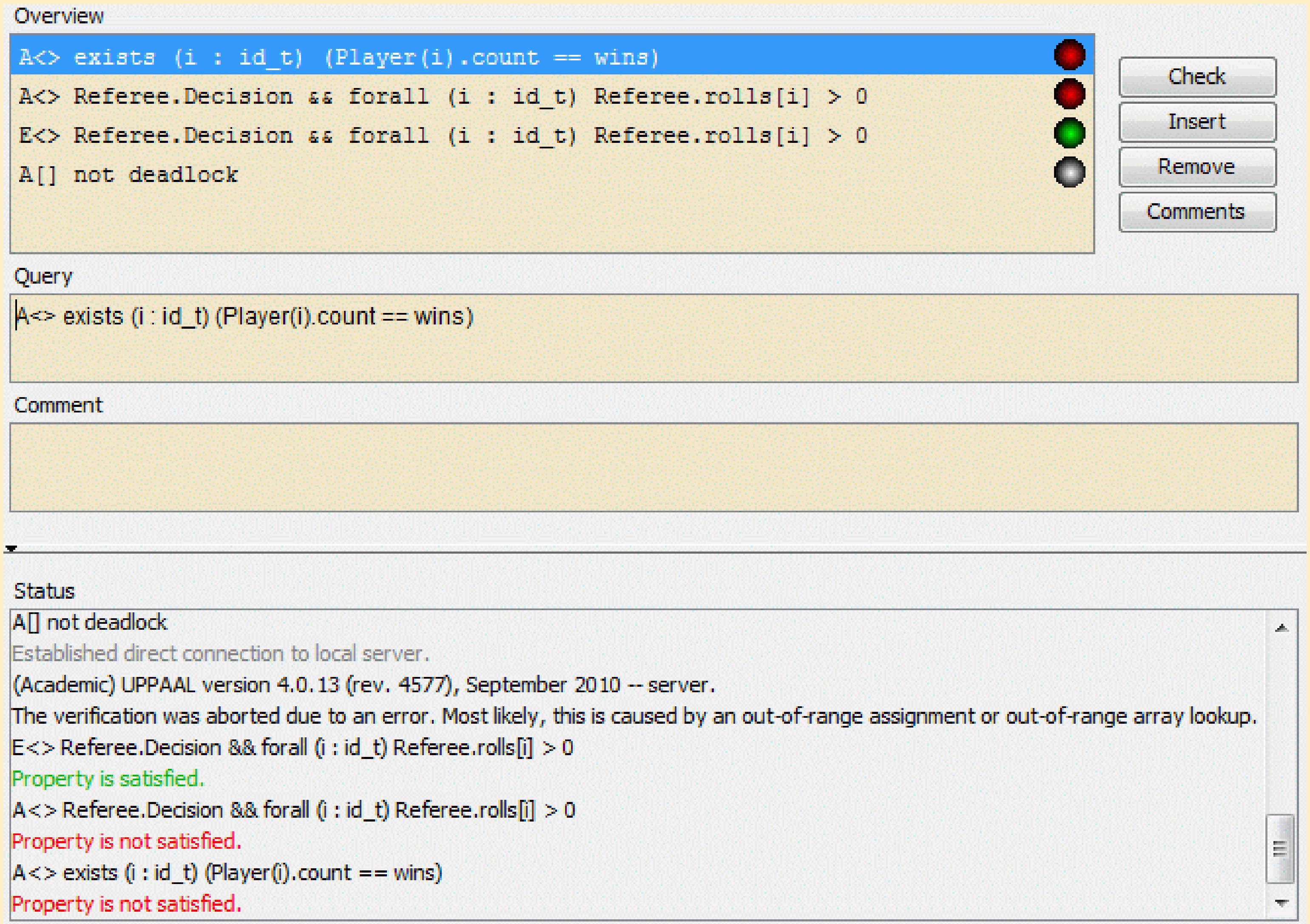
- Selection
  - Non-deterministic choice from the domain of a variable
- Guard
  - Enabling condition (logical expression)
- Synchronization
  - Synchronization on a channel between process "pairs"
- Update
  - Expression evaluated during the transition (may have side effect)

- Evaluation order of expressions: Select » Sync » Guard » Update

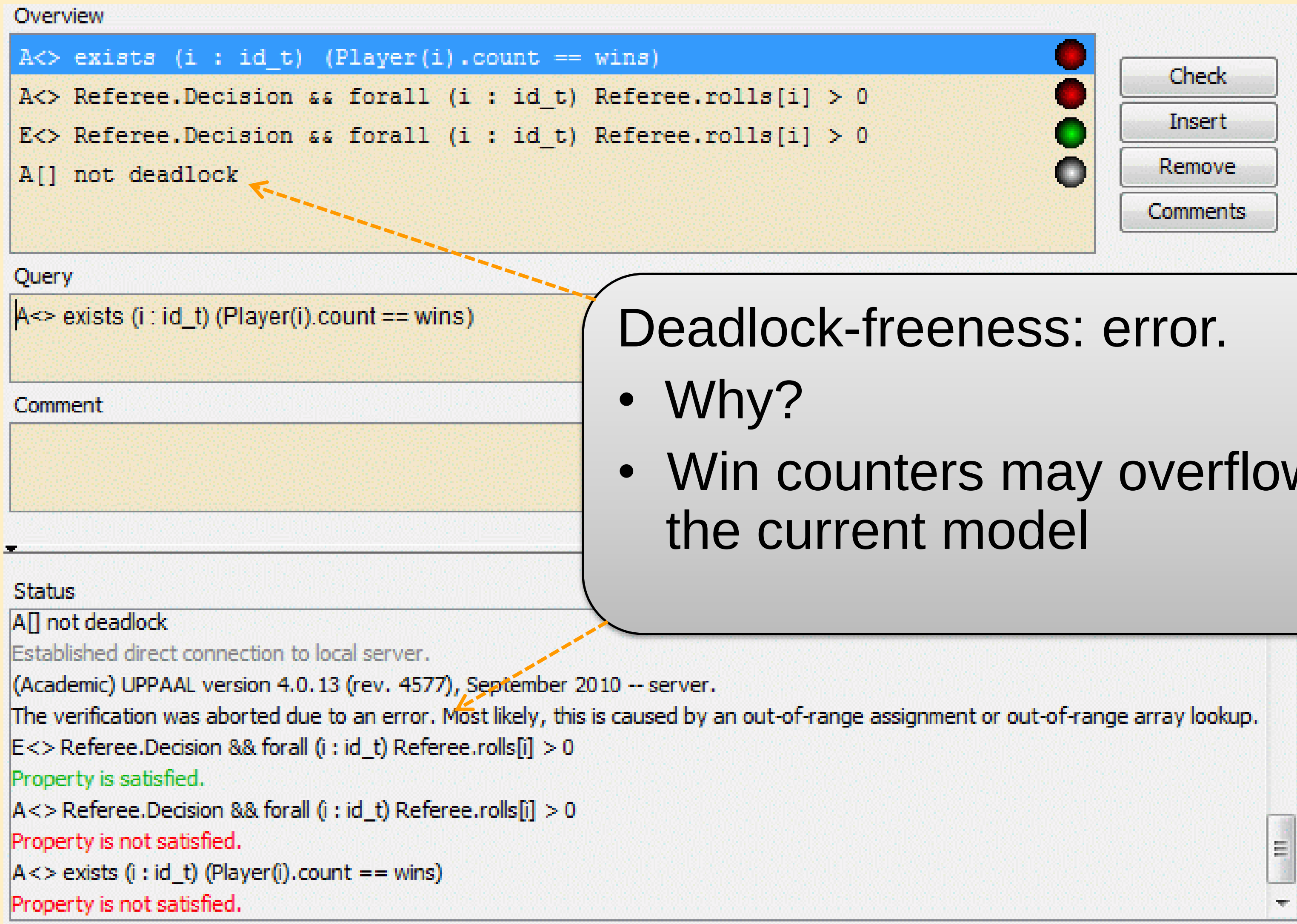
Let’s check the behavior!

- On each path, there is a player who wins all games
  - There is always an “absolute winner”
  - A<> exists (i : id_t) (Player(i).count == wins)
- Referee only decides if all players rolled
  - This happens at least once:
    - E<> Referee.Decision && forall (i : id_t) (Referee.rolls[i] > 0)
  - This happens at least once on all paths:
    - A<> Referee.Decision && forall (i : id_t) (Referee.rolls[i] > 0)
- The system has no deadlock
  - There is no such state, which has no enabled (!) transition to another state
  - A[] not deadlock

Let’s check the behavior!



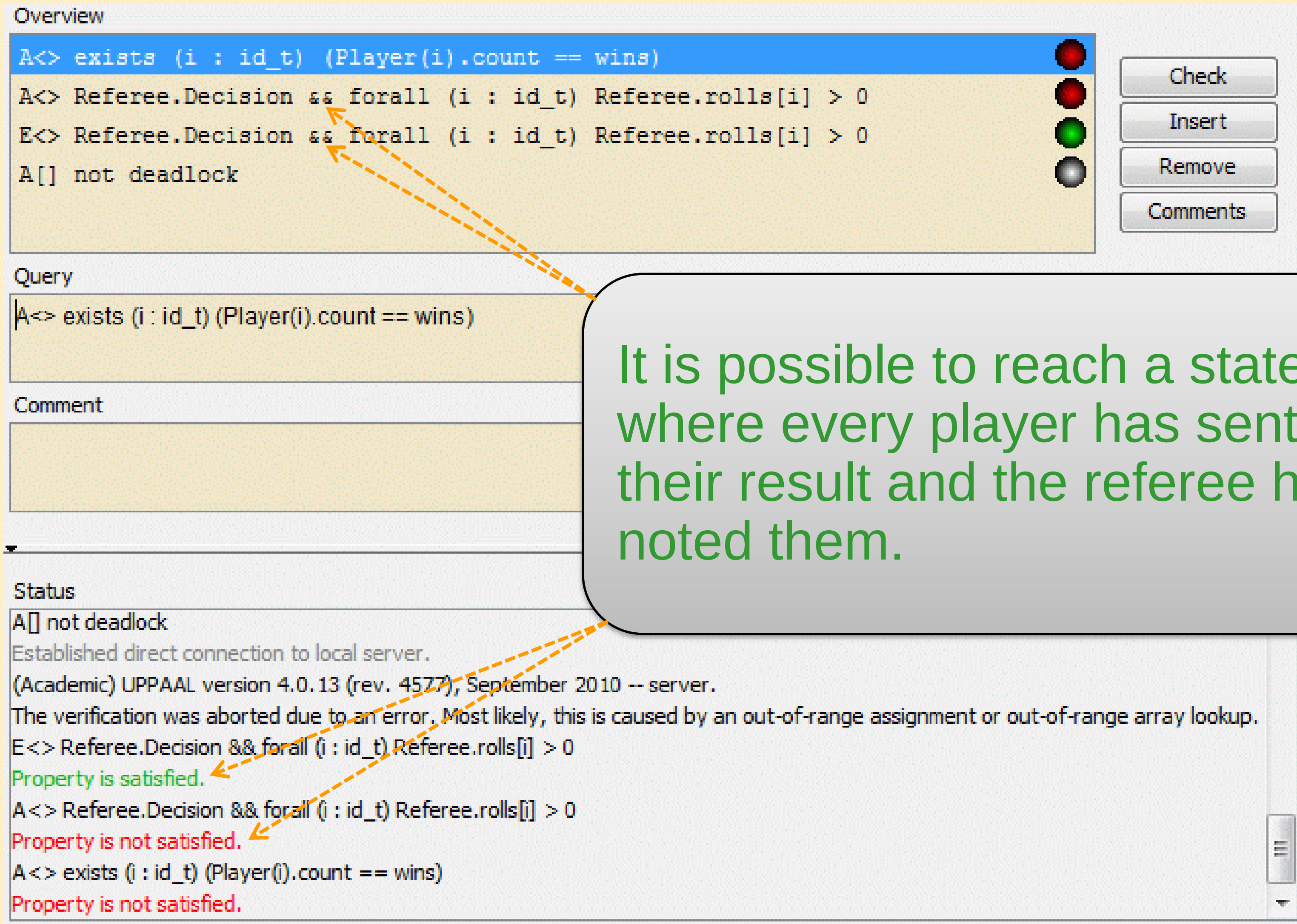
Let’s check the behavior!



Deadlock-freeness: error.

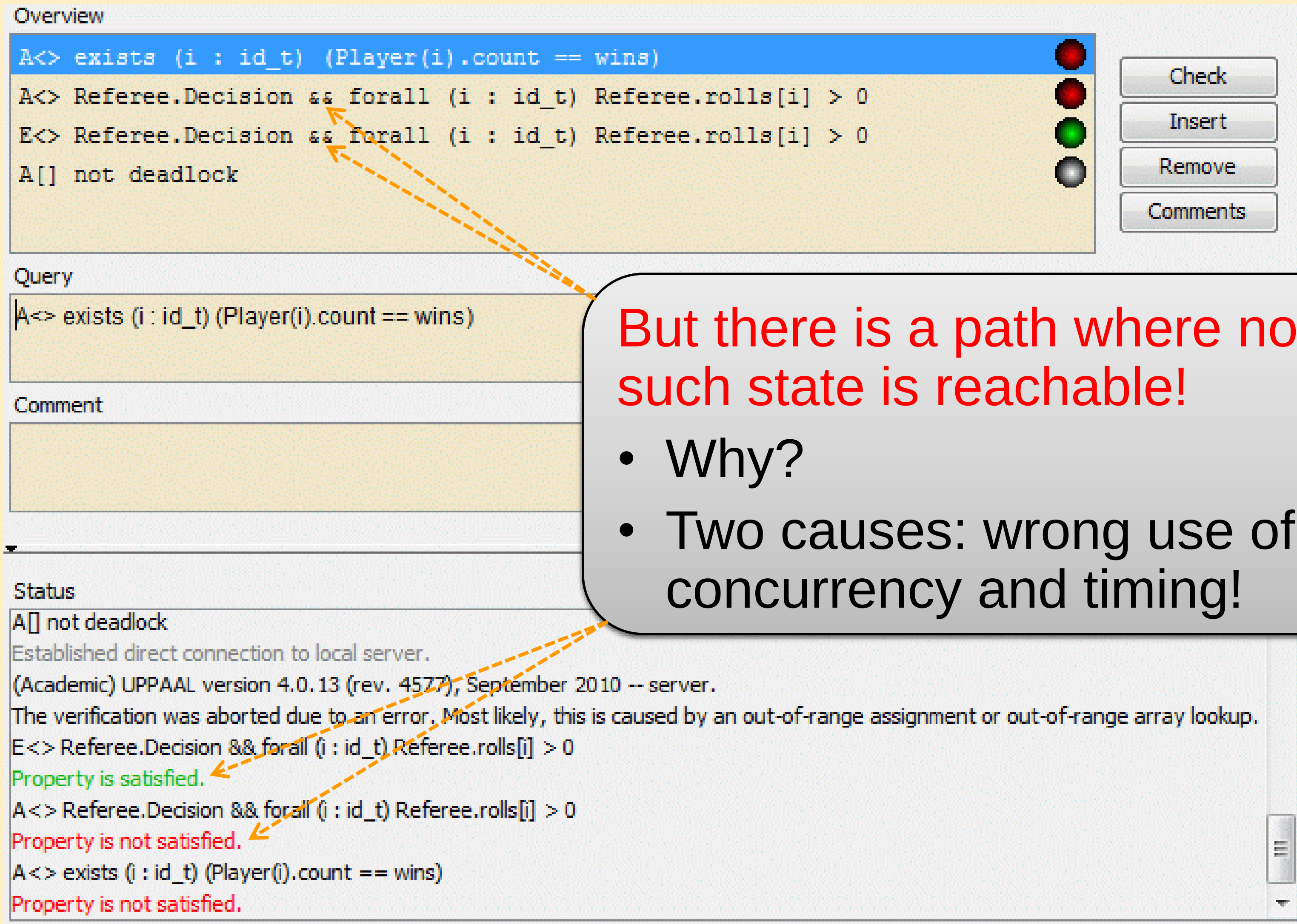
- Why?
- Win counters may overflow in the current model

Let's check the behavior!



It is possible to reach a state where every player has sent their result and the referee has noted them.

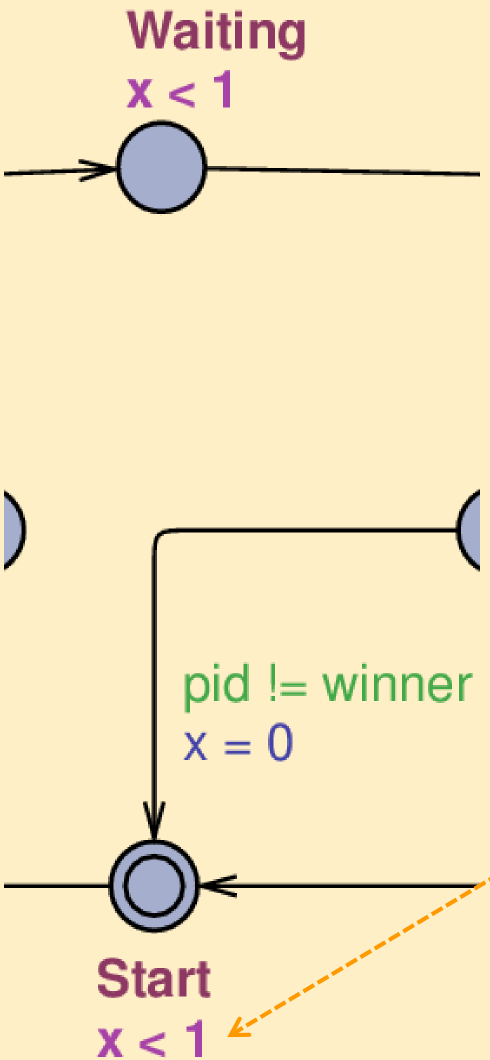
Let's check the behavior!



But there is a path where no such state is reachable!

- Why?
- Two causes: wrong use of concurrency and timing!

Wrong timing? Why?

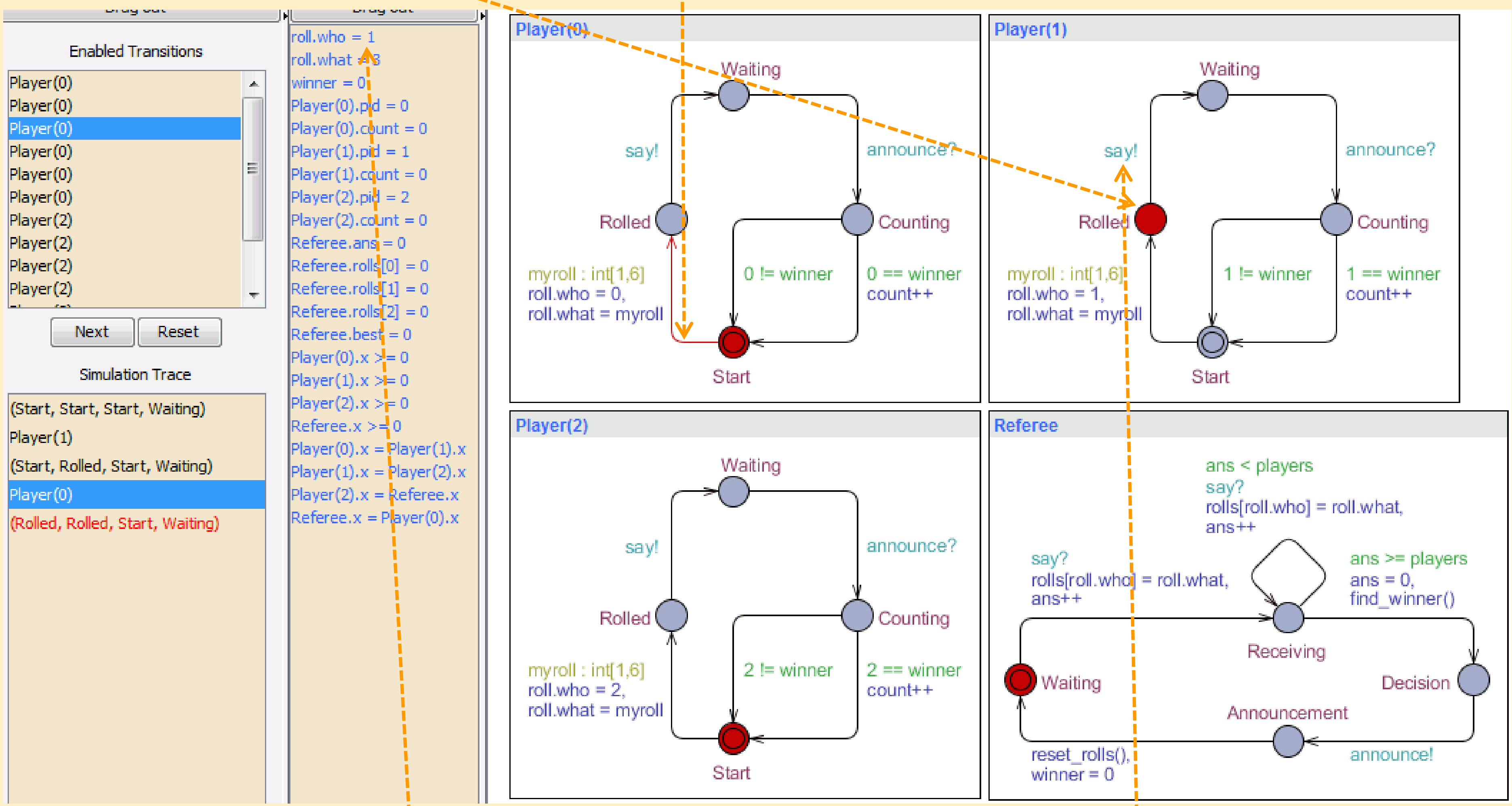


- If we examine all possible paths (e.g. A<>) then UPPAAL also checks the possibility of not leaving a state
- Solution:
  - Introduce a clock variable
    - Add invariant to state
    - We can only stay in a state for at most 1 time units
    - Don't forget to initialize the clock variable!

Wrong concurrency? Why?

2nd player rolls

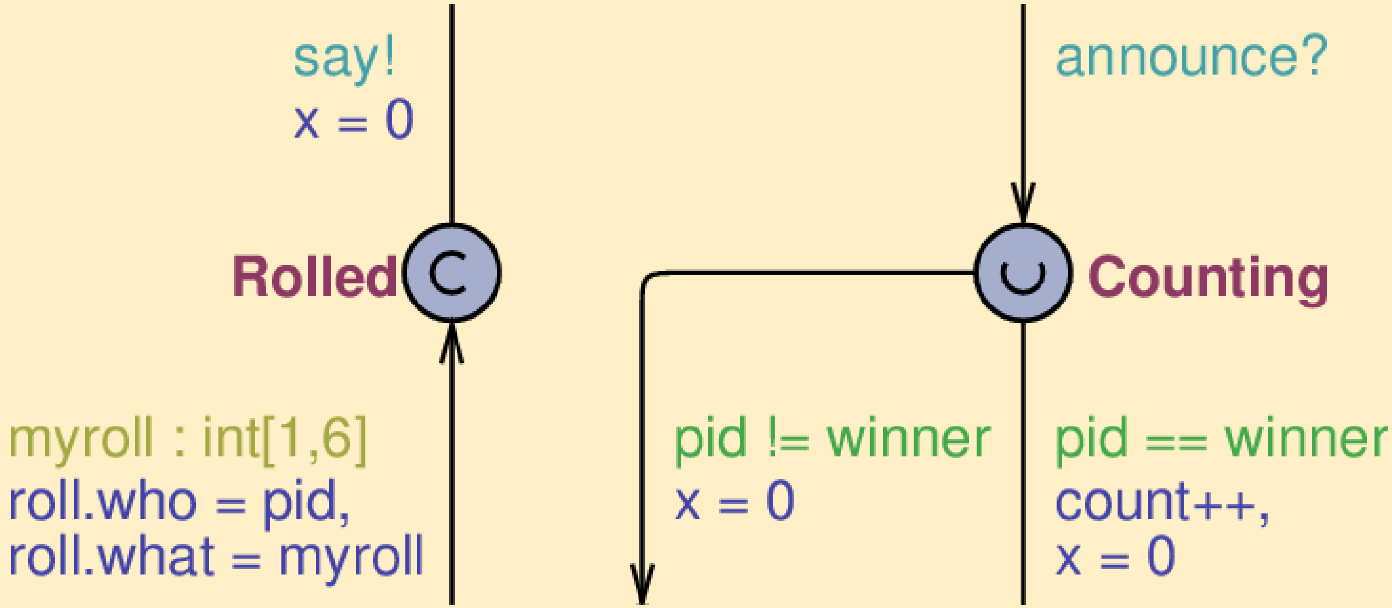
1st player rolls



1st player overwrites the shared variable

2nd player “sends” wrong one

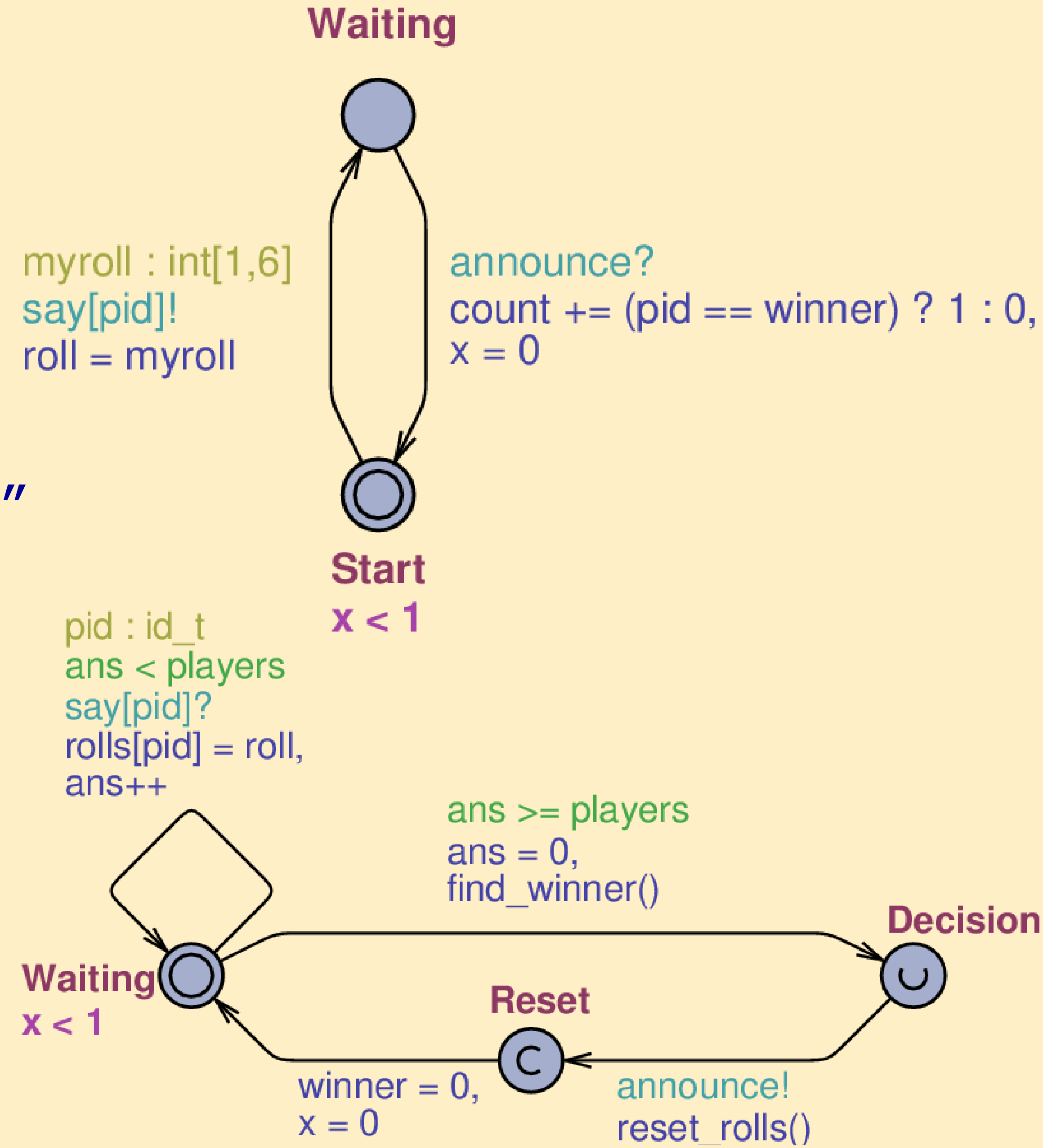
Avoiding wrong concurrency



- The problem is that states Waiting and Rolled are concurrent and firings are non-deterministic
- Solution:
  - Avoiding concurrency: introduce "committed" state
    - We must leave a "committed" state instantly

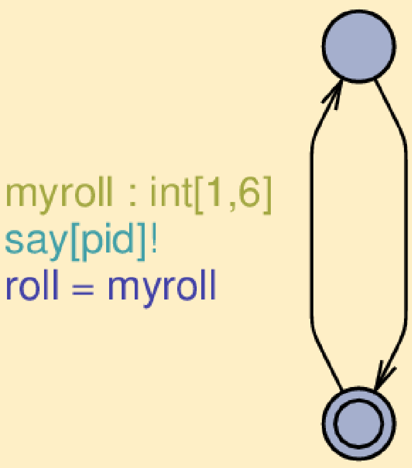
Other constructs for simplification

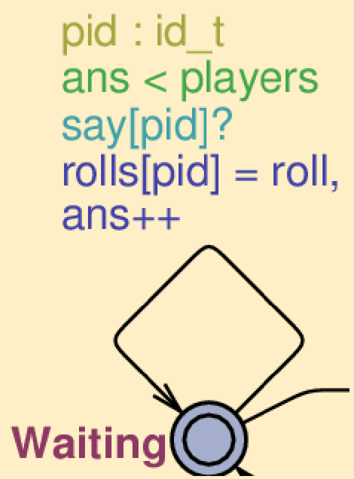
- Using arrays of channels
- Applying operator "? :"
- Collecting results in a single state
- Using iterators
- Omitting reset state



Special constructs

- Using arrays of channels
  - Receiving process monitors all channels "at once" using a Select construct
  - Channel id can be used in the Update section!





- Using iterators

```
void reset_rolls() {
  for (i : id_t) rolls[i] = 0;
}

void find_winner() {
  for (i : id_t) {
    if (rolls[i] > best) {
      best = rolls[i];
      winner = i;
    }
  }
  best = 0;
}
```

Other modeling advices, best practices

- Order of evaluating arc expressions:
  Select » Sync » Guard » Update
  - On a synchronized arc, Update of the sender is evaluated before the Update of the receiver!
  - Cannot test a global variable that was set by synchronized arc!
  - Cannot “test” a variable in Sync with a Guard!
- Checking the behavior of functions is difficult. Debugging is not possible. Try to develop the model in small steps and check its behavior often with simulation and verification!

Other modeling advices, best practices

- When verifying properties such as A<> q, clock variables must be used to avoid the trivial counterexample.
  - Do not forget the semantics of "leads to" p --> q: A[] (p imply A<> q)
- Do not forget to initialize clock variables!
- The model checker of UPPAAL cannot handle deadlocks when using channel or automata level priorities. Such modeling constructs should be avoided.

Solving an exercise

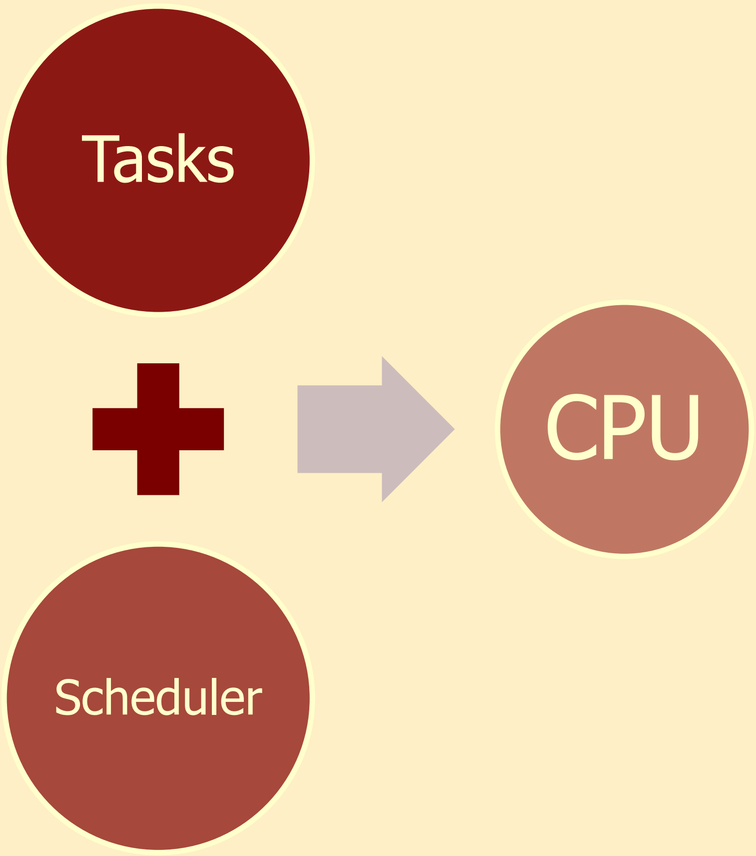
Using our knowledge so far

The exercise

- Modeling tasks and threads in a simple operating systems
  - Tasks are executed in fixed length periods
  - At the beginning of each period, tasks decide (non-deterministically) if they "apply" for running or if they decline running in that period
  - Each task requires a given percentage of CPU
  - Finite number of threads, one task per thread
  - At the end of a period, tasks are stopped and the operating system returns to its initial state
  - The process above is repeated

The system contains three main components

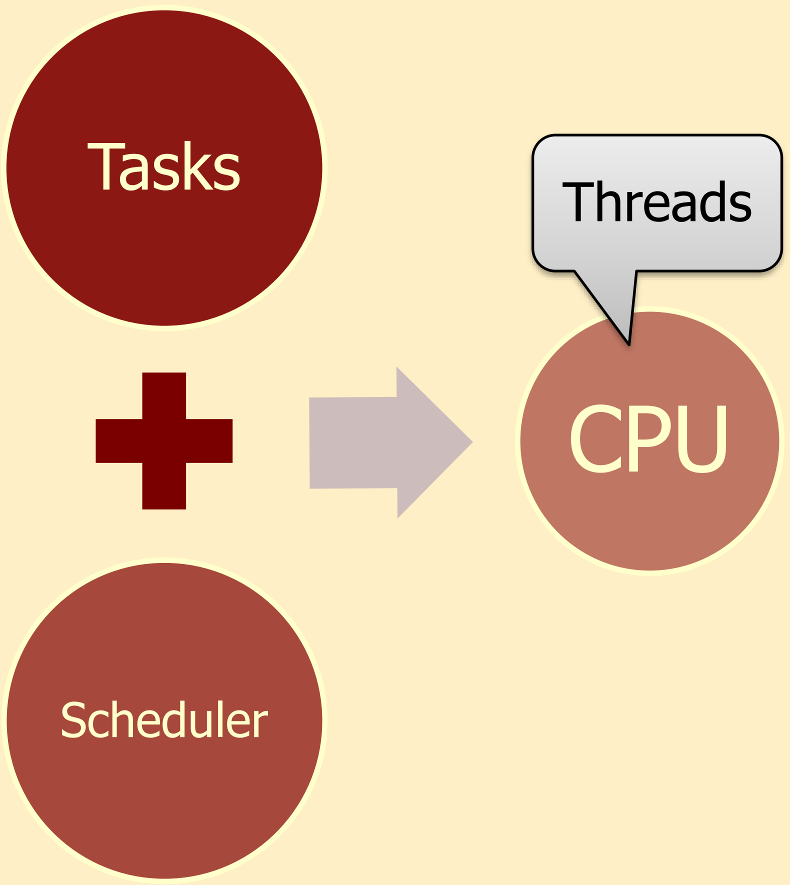
- Tasks
  - Affinity: probability of the task "applying to run"
  - Demand: the task requires (10*Demand) percent of CPU
  - Priority: priority level of the task



- The total CPU requirement must be at most 100% for the tasks that are selected for running
- Within this limit, tasks have to be selected based on their priority

The system contains three main components

- Scheduler
  - Selects running tasks from those that "applied to run"
  - There is a limited number of threads that can run tasks
  - Each task is allocated to a separate thread
  - No more tasks can be running than the number of threads
- CPU
  - Resource needed to run tasks
  - Two states: active, inactive
  - Tasks can run in active state
  - A preemptive interrupt can occur in active state

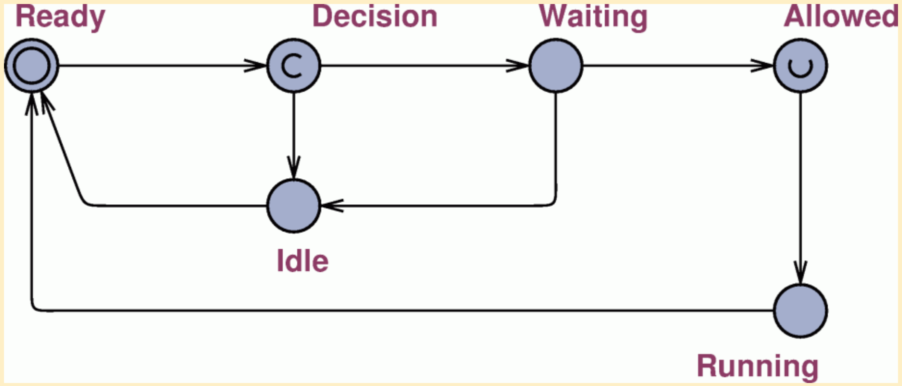


Basic operation of the system

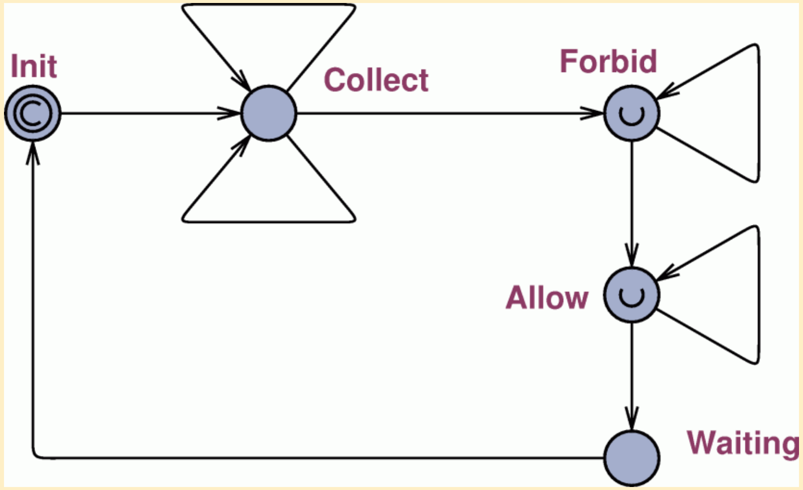
- The tasks
  - Generate a random number *p* between 0 and 10 when leaving their initial state
  - This is compared to their Affinity parameter: if $p \geq$ Affinity, then they apply for running, otherwise they decline to run and become inactive
- The scheduler
  - Stores applications and declines
  - Processes applications: orders the tasks descending by priority and CPU requirement, while observing the limits
  - Assigns the selected tasks to threads and stores this assignment in a global data structure

Let's start modeling!

- Task
  - Ready: initial state
  - Decision: decides on running
  - Idle: declined, inactive
  - Allowed: selected for running
  - Running: runs

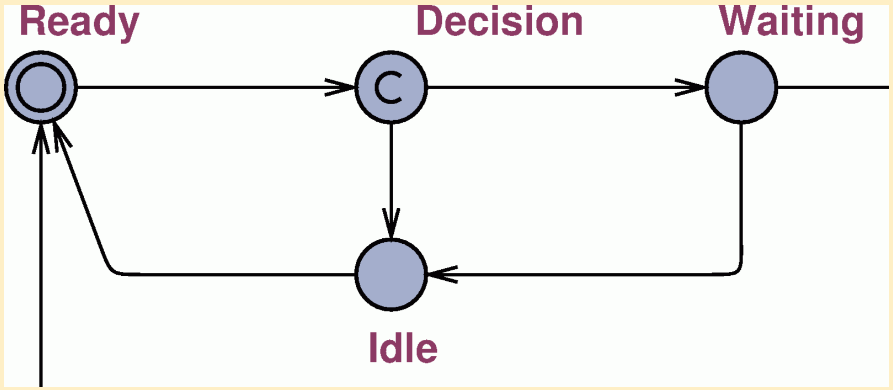


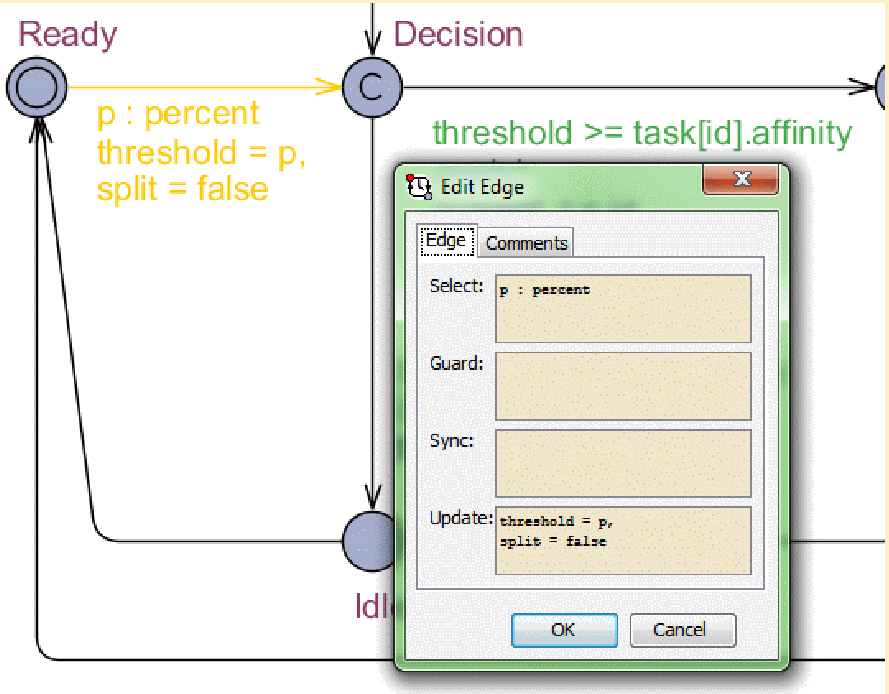
- Scheduler
  - Init: initial state
  - Collect: collecting applications and declines
  - Forbid: notifies rejected tasks
  - Allow: notifies selected tasks
  - Waiting: waiting to end period



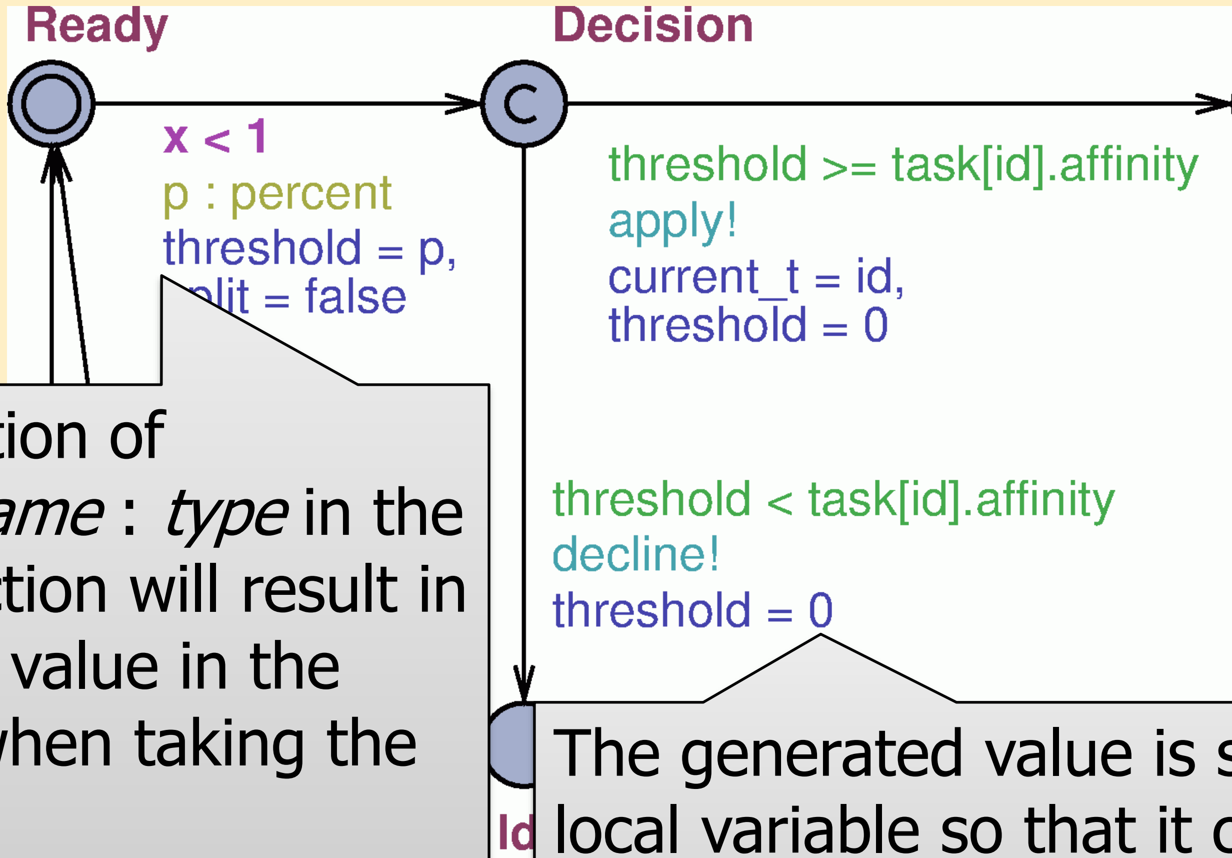
First problem: Modeling random choice

- Simple solution
  - Does it work? Yes, because UPPAAL chooses randomly from enabled transitions
  - Is it what we want? No, because probabilities should be proportional to the affinities
- Correct solution
  - Generate random value using Select construct of UPPAAL





Modeling random choice



A declaration of *variablename* : *type* in the Select section will result in a random value in the variable when taking the transition.

This variable can only be used in other expressions of the same transition!

The generated value is stored in a local variable so that it can be used in the proceeding steps.

The purpose of the Committed state is that the two operations should not be interrupted.

Declarations

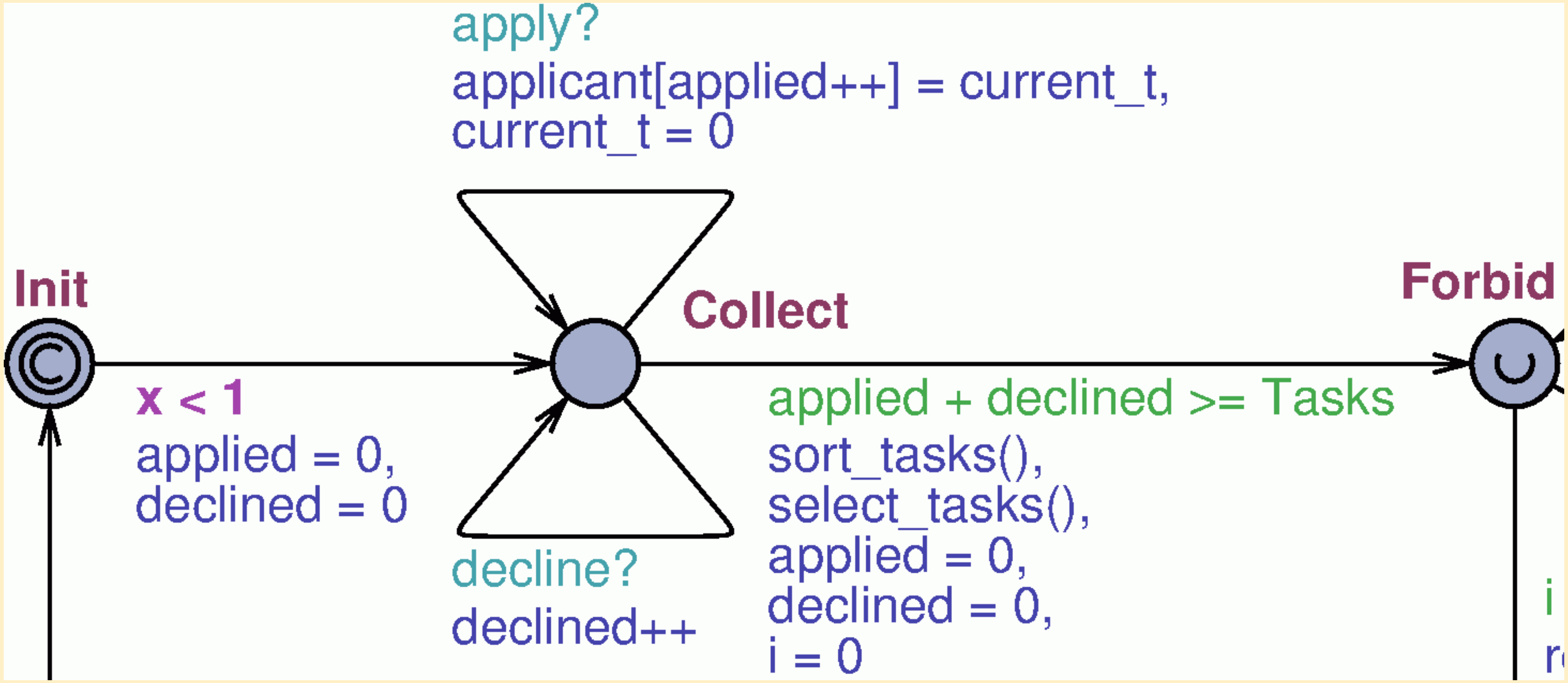
Global

```
typedef int[0,10] percent;

const int Levels = 3;
typedef int[0,Levels-1] p_level;

const int Tasks = 5;
typedef int[0,Tasks-1] t_id;
t_id current_t;

typedef struct {
  percent affinity;
  percent demand;
  p_level pri;
} task_t;

// affinity, demand, priority
const task_t task[Tasks] = {
…,
};
```

Local (Task)


```
clock x;
meta bool split = false;
percent threshold;
```

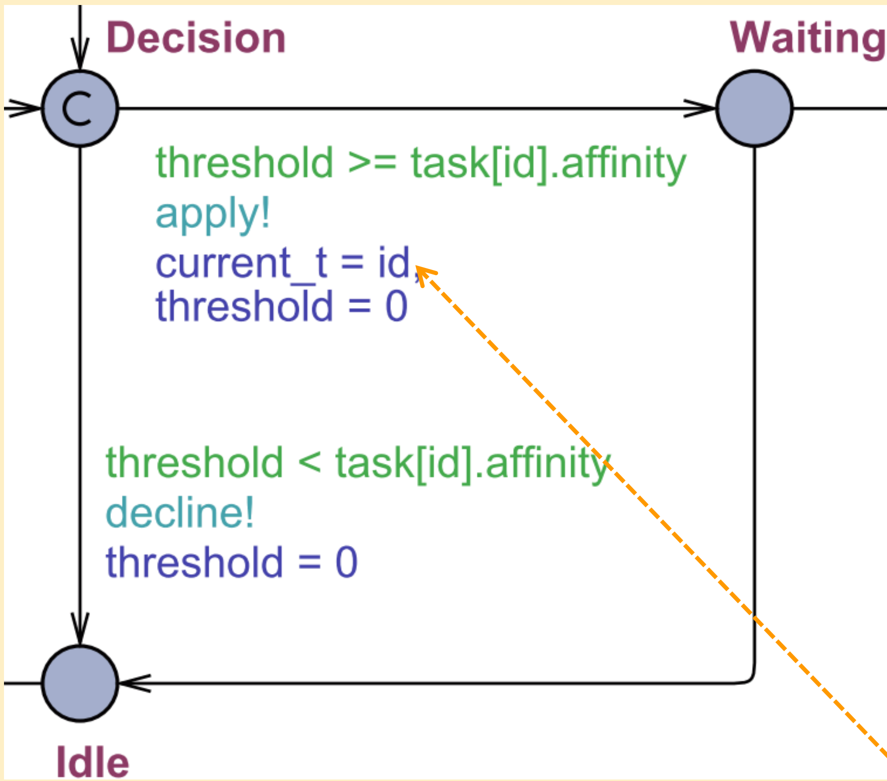
How does counting applications work?



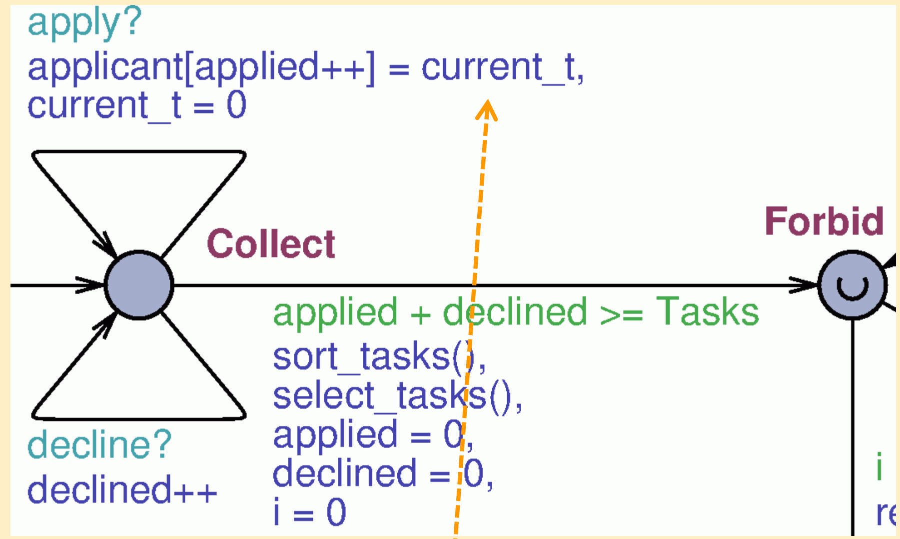
- We are staying in state Collect until each task either applied or declined
- Applied tasks are stored in a local array
- Functions sort_tasks(), select_tasks() are selecting tasks when entering state Forbid

Collecting applications and declines

Task



Scheduler

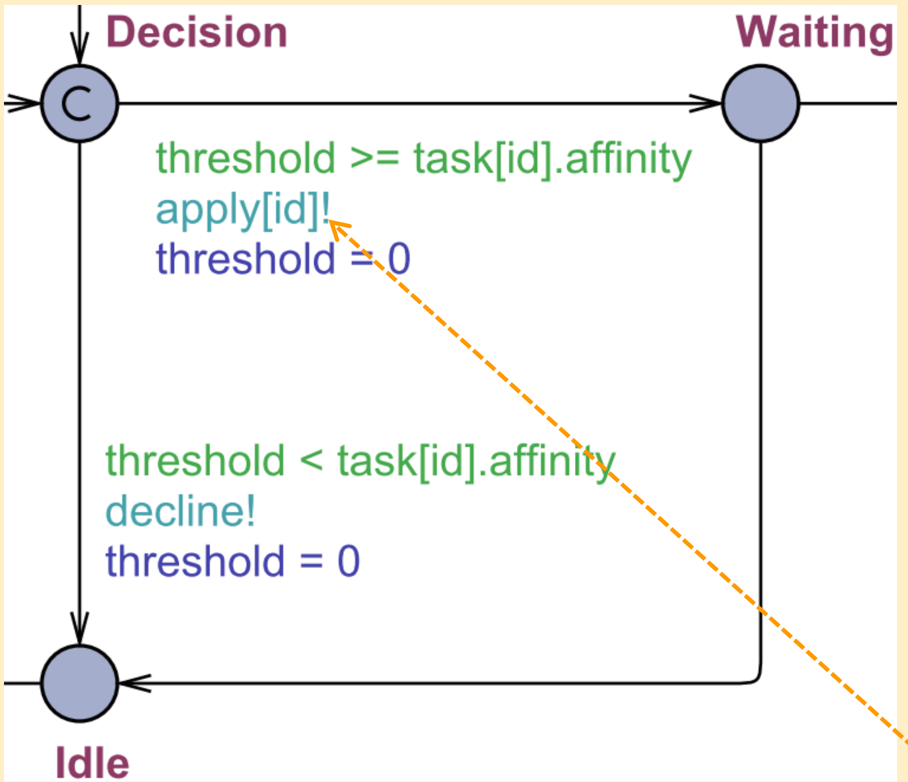


Modeling synchronous communication with global variable.

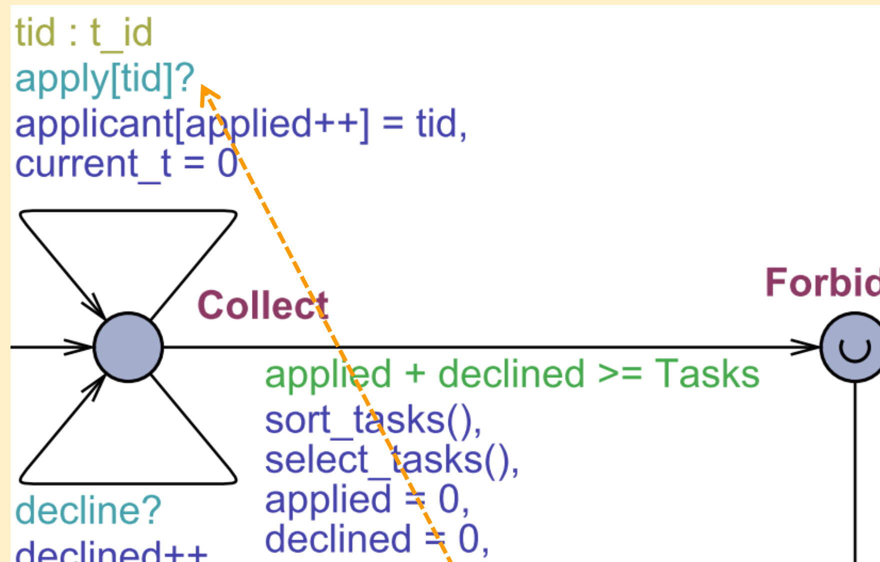
It is guaranteed that the Update of the sender is executed first!

Modeling synchronous communication / 2

Task



Scheduler

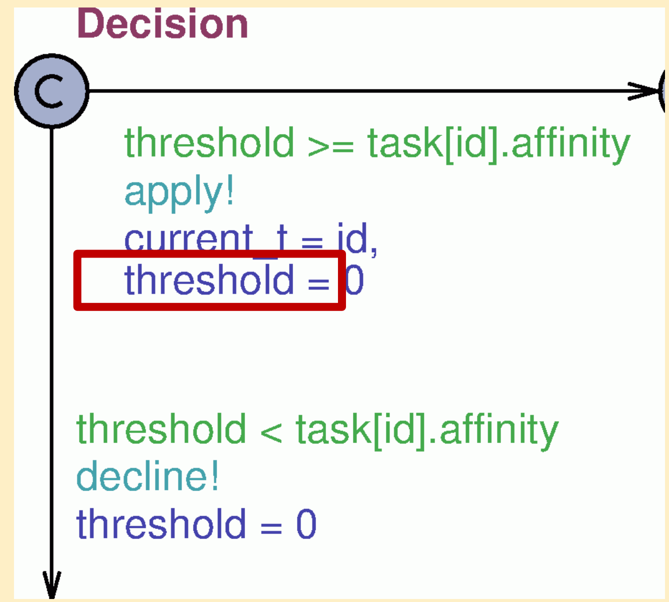


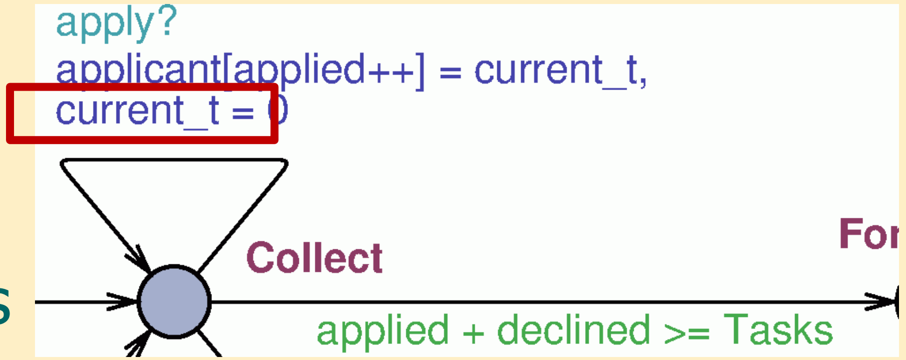
Modeling synchronous communication using channel arrays.

Can only be used for variables with small domain!

Why should we reset temporary variables?

- Temporary variable
  - A set of trajectories for each value
  - Multiplies the size of the state space
  - Can be reduced by resetting the variable
- Interleaving
  - Transitions of asynchronous automata in different orders
  - Same result on different paths
  - Can be reduced by synchronization and committed states



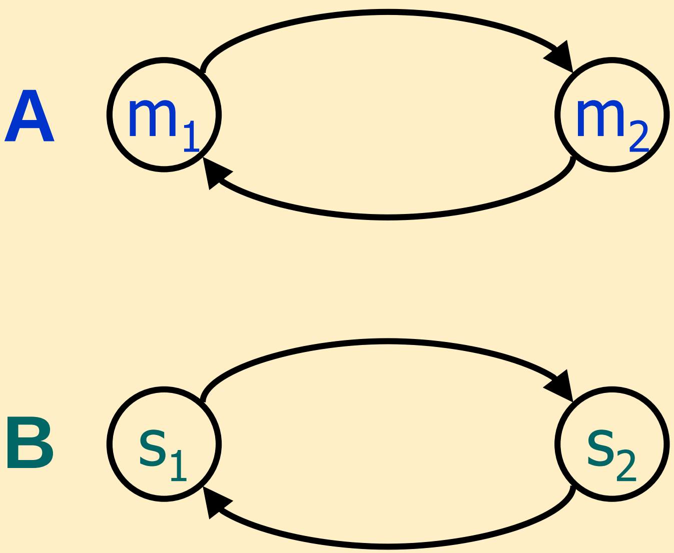


Outlook: Behavior of two automata

Direct product, interleaving, synchronization

Behavior of asynchronous automata: Interleaving

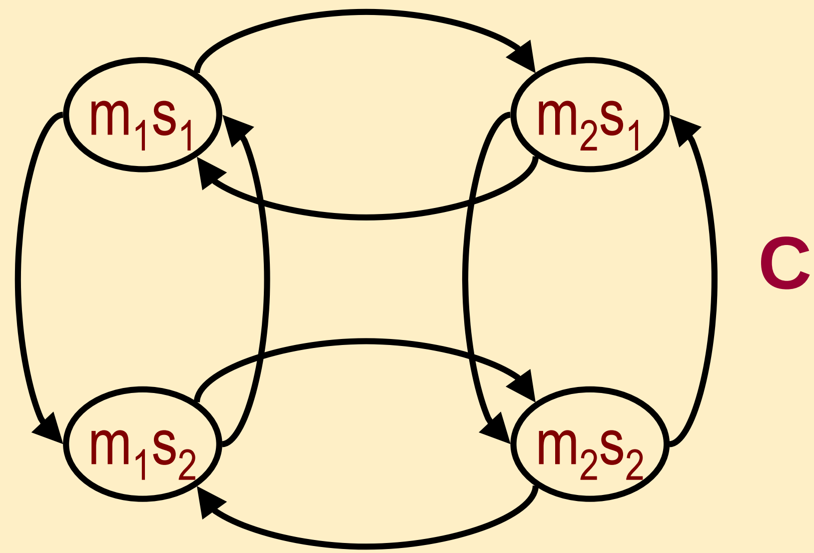
- System of two (independent) automata



- States of the automata:

  $A = \{m_1, m_2\}$, $B = \{s_1, s_2\}$

- (Direct) product: state space of the system

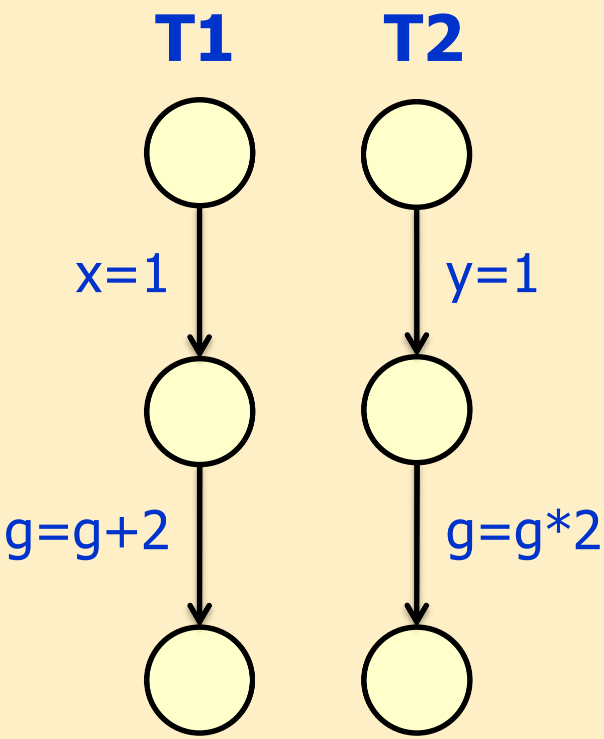


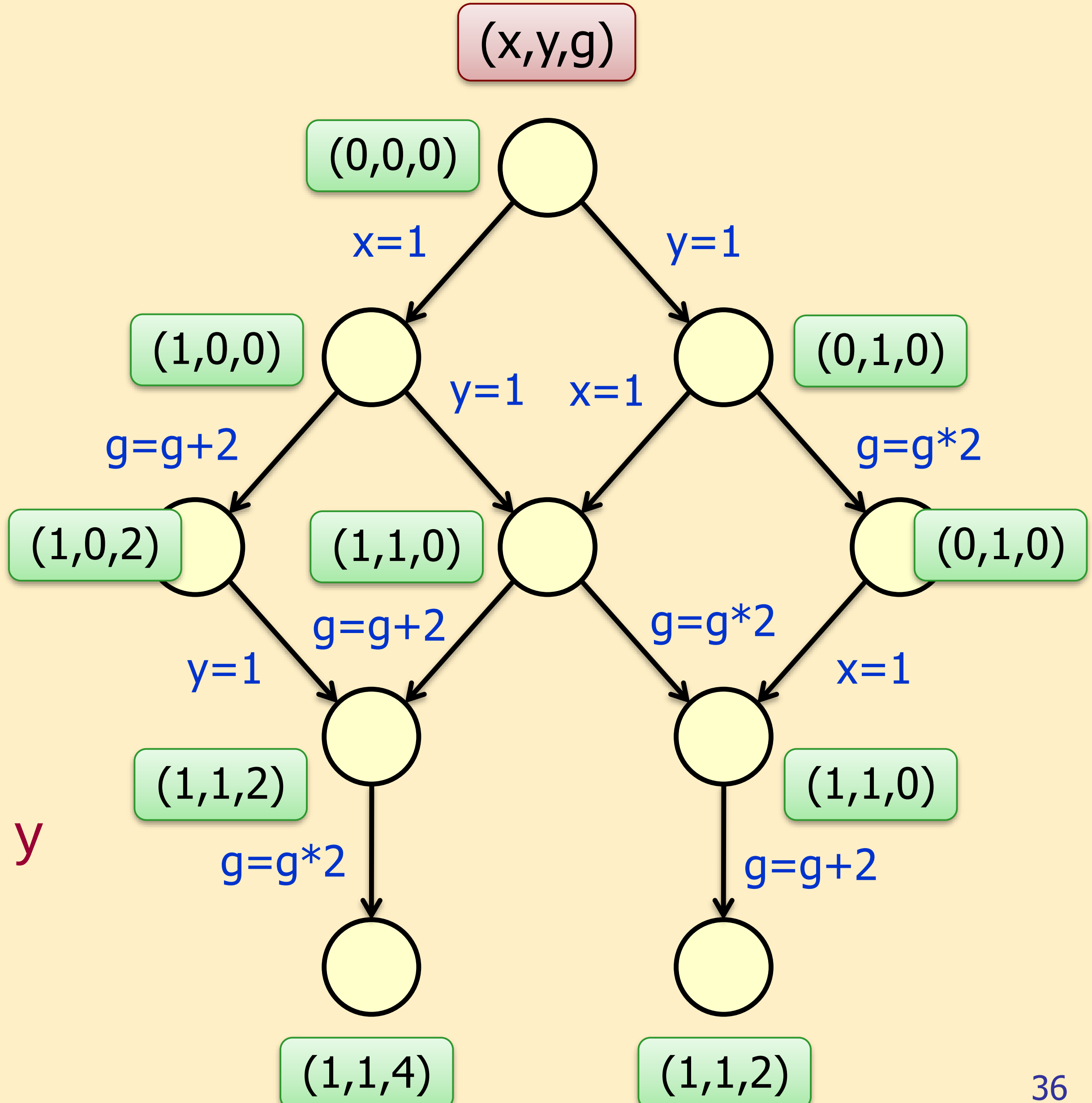
- Set of states:

  $C = A \times B$

  $C = \{m_1s_1, m_1s_2, m_2s_1, m_2s_2\}$

Example: Alternative paths



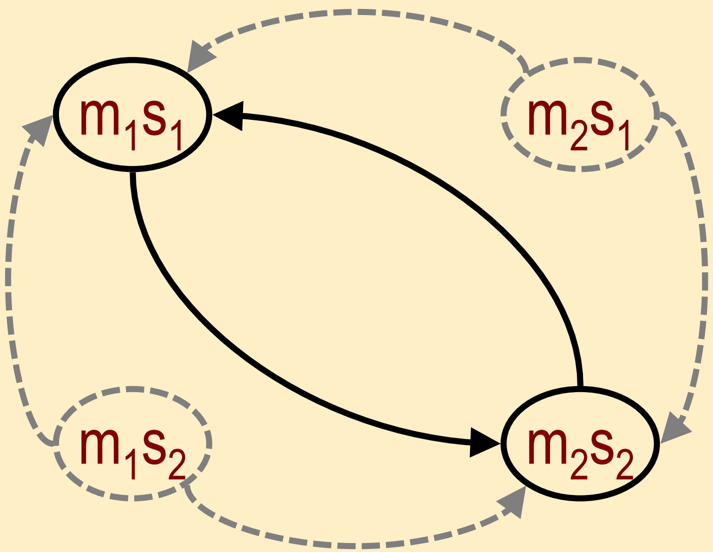


Local variables: x and y

Global variable: g

Synchronizations and guards simplify the model

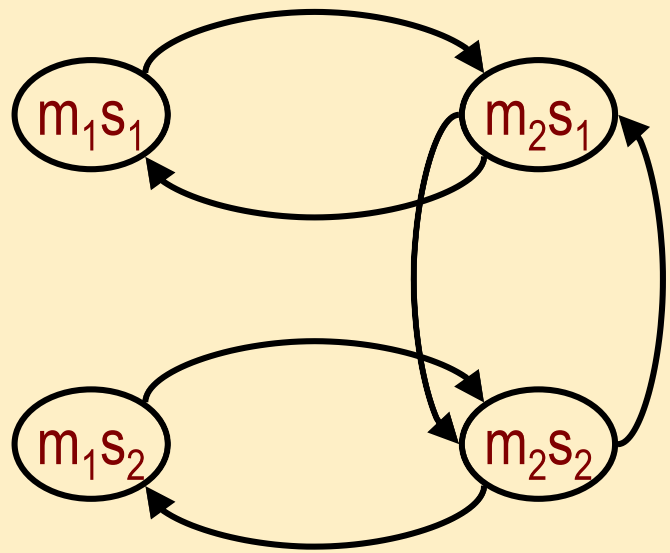
- Synchronization: taking the transitions at the same time



C'

- E.g. "A and B takes the transition at the same time if their state index is the same"

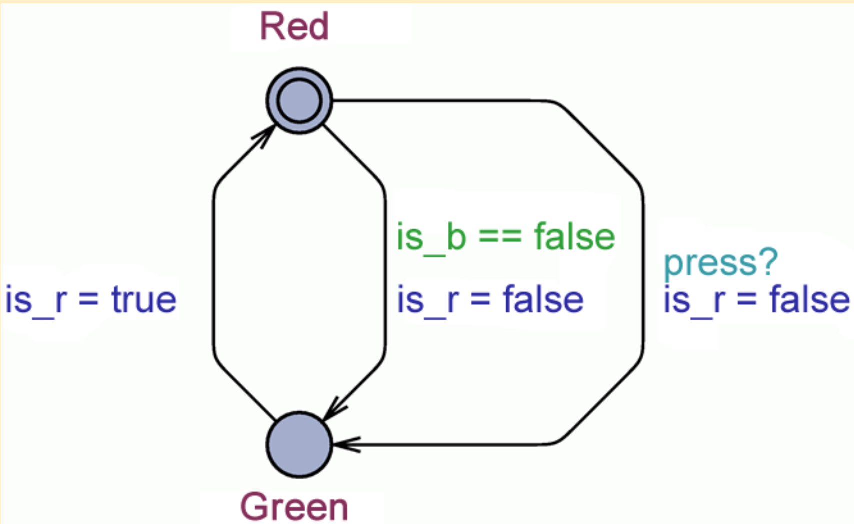
- Guards: disable certain transitions

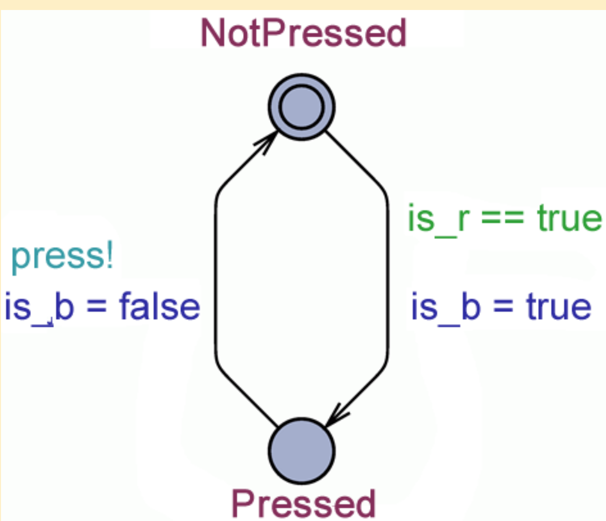


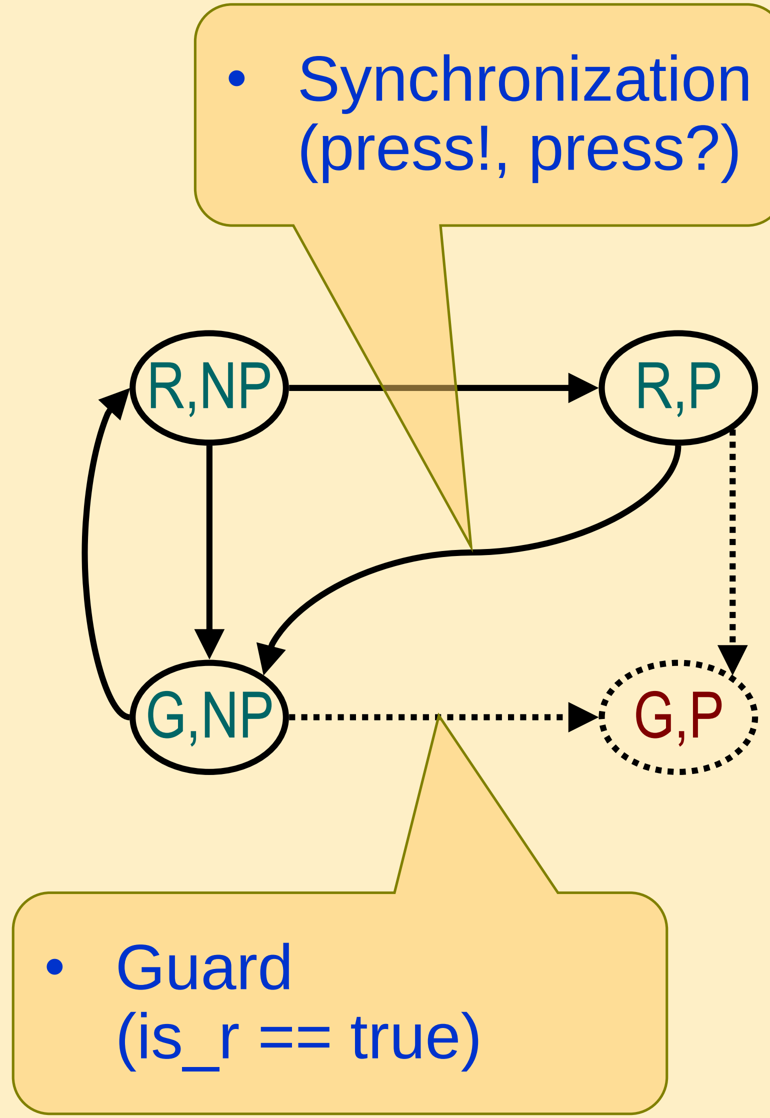
C''

- E.g. "B can only take the transition if A is in state m_2"

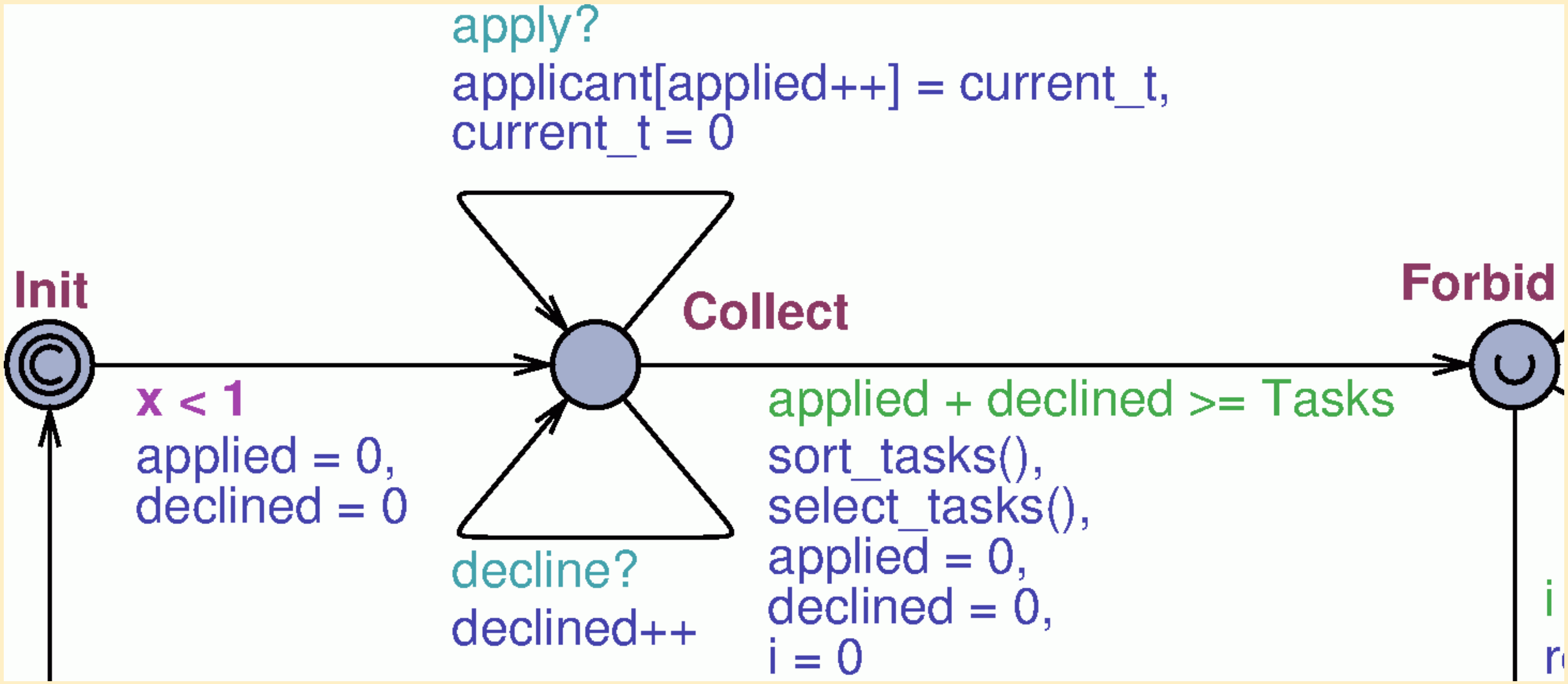
Example: Pedestrian light with button







Let’s get back to our exercise



- Applied tasks are stored in a local array
- Functions sort_tasks(), select_tasks() are selecting tasks when entering state Forbid
  - Ordering tasks decreasing by their priority and CPU requirement, while observing the limits

Selecting and rejecting tasks

- sort_tasks()
  - Uses a 2D array for ordering:

```
typedef struct {
  int[0,Tasks] length;
  t_id task[Tasks];
} buffer_t;
buffer_t buffer[Levels];
```

- select_tasks()
  - Collects selected tasks decreasing by priority until a limit is reached

- Let the parameters be:

```
// affinity, demand, priority
const task_t task[Tasks] = {
{0, 2, 0},
{3, 3, 1},
{3, 4, 1},
{3, 1, 1},
{3, 5, 2}
};
```

- Example applicants: 0, 2, 3, 4
- Example order:

```
buffer[0] = [0]
buffer[1] = [2, 3]
buffer[2] = [4]
```

- Selected: 0, 2, 3
- Rejected: 4

Ordering tasks based on CPU requirement

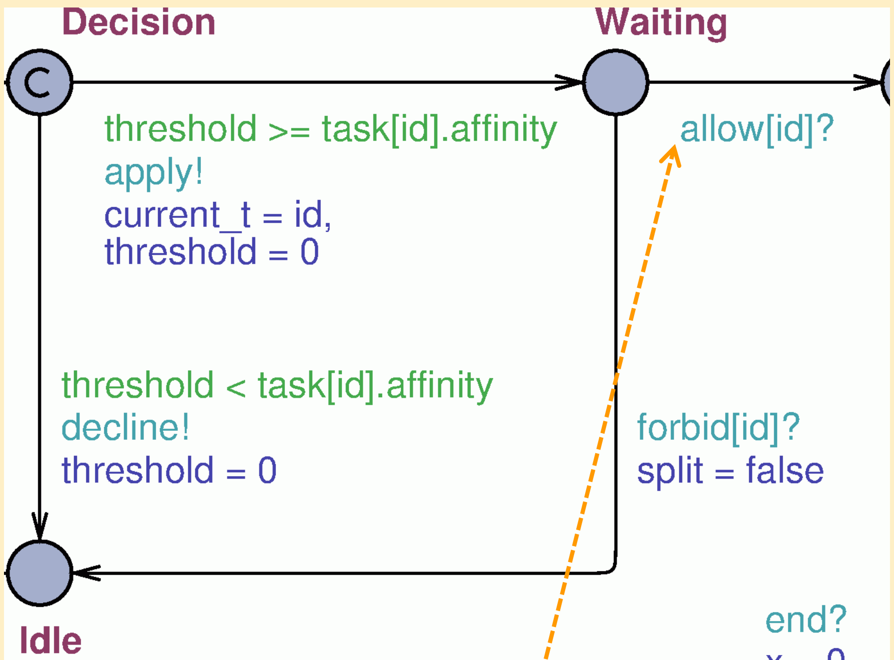
```
void insert_at(int[0,Tasks] pos, t_id tid) {
  int i;
  for (i = buffer.length; i > pos; i--) {
    buffer.task[i] = buffer.task[i - 1];
  }
  buffer.task[pos] = tid;
  buffer.length++;
}

void sort_tasks() {
  int i, j, pri, pos;
  for (i = 0; i < applied; i++) {
    pri = task[applicant[i]].pri;
    for (j = 0, pos = -1; j < buffer[pri].length && pos < 0; j++) {
      if (task[applicant[i]].demand > task[buffer[pri].task[j]].demand)
        pos = j;
    }
    insert_at(pri, pos < 0 ? buffer[pri].length : pos, applicant[i]);
    applicant[i] = 0;
  }
}
```

Selecting tasks while observing limits

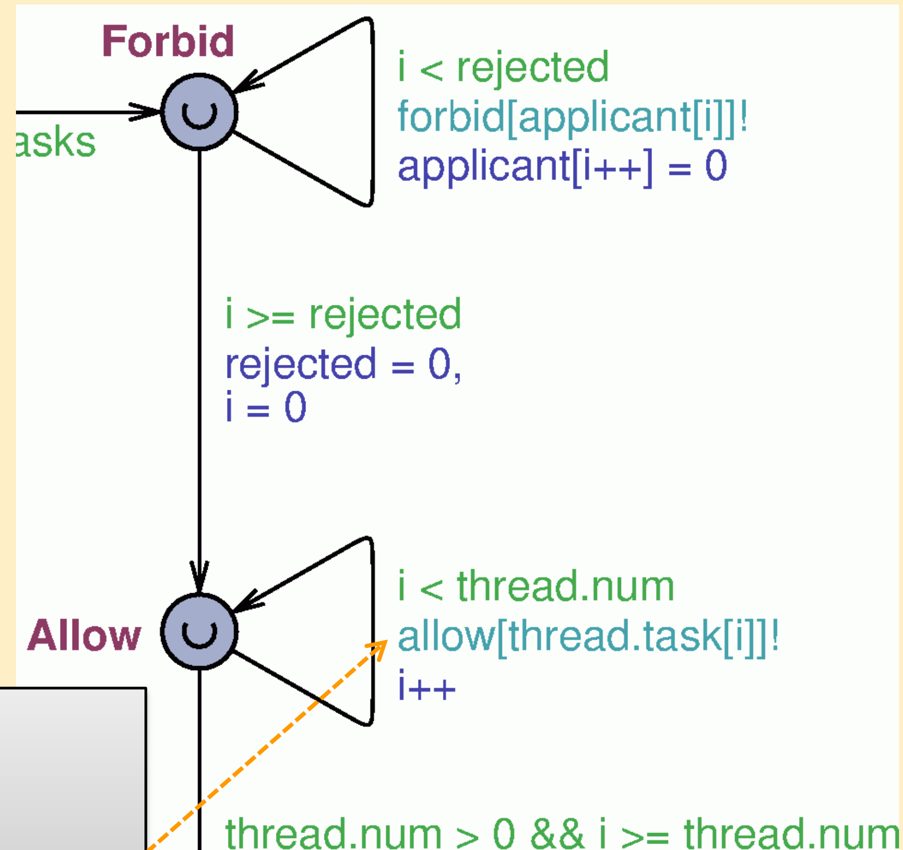
```
void select_tasks() {
  int i, pri;
  percent p = 0;
  rejected = 0;
  thread.num = 0;
  for (pri = 0; pri < Levels; pri++) {
    for (i = 0; i < buffer[pri].length; i++) {
      if (p + task[buffer[pri].task[i]].demand <= 10 &&
          thread.num < Threads) {
        thread.task[thread.num++] = buffer[pri].task[i];
        p = p + task[buffer[pri].task[i]].demand;
      }
      else applicant[rejected++] = buffer[pri].task[i];
      buffer[pri].task[i] = 0;
    }
    buffer[pri].length = 0;
  }
}
```

Notification about selection and rejection

Task



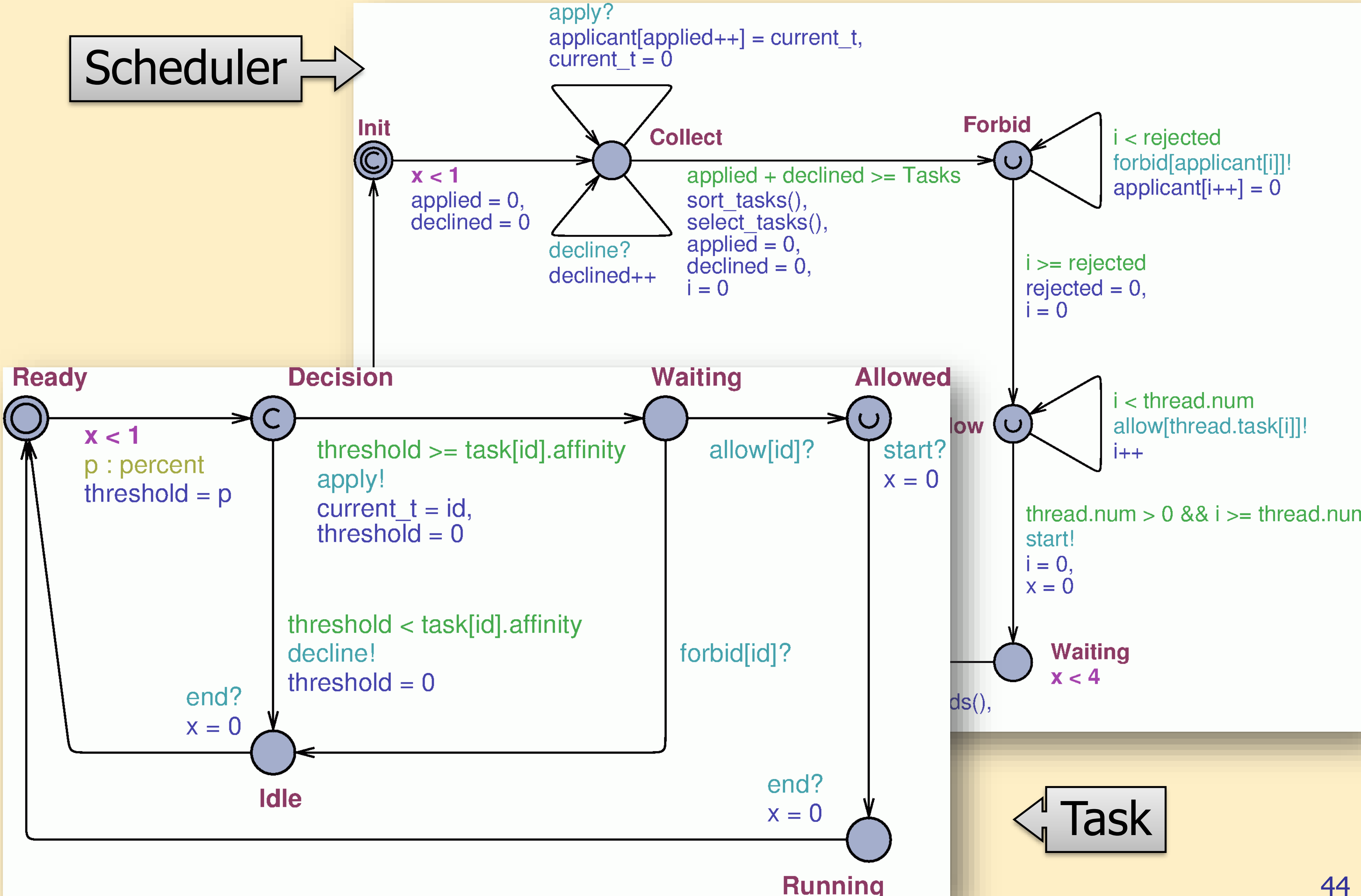
Scheduler



Selected and rejected tasks are notified individually on separate channels.

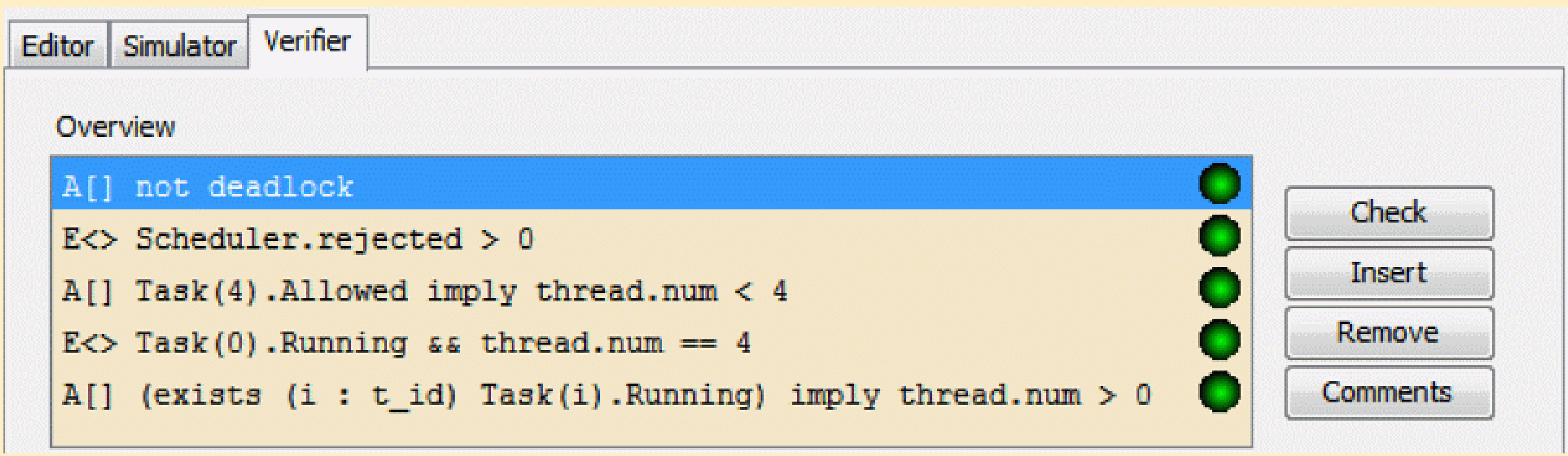
Temporary variables are reseted.

The model already works (without a CPU)



Intermediate checking

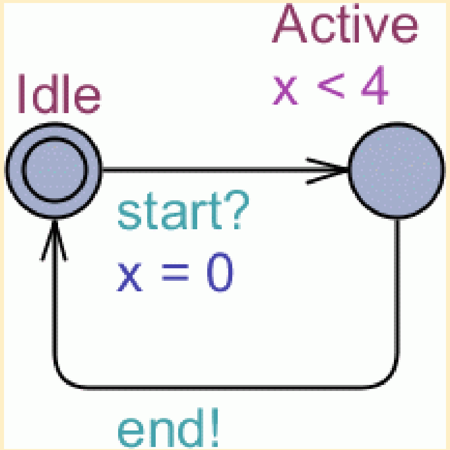
- We already have a functional system
  - It is recommended to check this intermediate system
- Some requirements:
  1. The system contains no deadlocks.
  2. It is possible that a task is rejected by the scheduler.
  3. When selecting task 4, not all threads can be occupied.
  4. It is possible that all threads are occupied.
  5. It is not possible that a task is running but no thread is occupied.

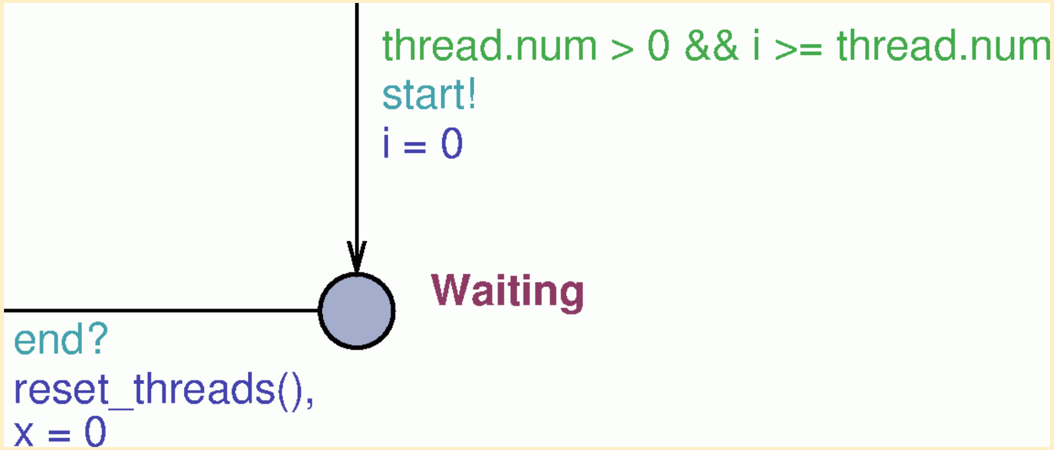


Extending the model with a CPU

- Starting signal is sent by the scheduler on a broadcast channel. After this:
  - Tasks selected to run change to running state,
  - The scheduler changes to idle state until the end signal,
  - The CPU changes to active state, threads and tasks running are stored in a global data structure.
- The CPU sends an end signal when leaving active state. After this:
  - The CPU changes to inactive state,
  - The scheduler changes to initial state, the list of running threads and tasks is cleared,
  - Tasks also change to their initial state.

Starting and stopping with CPU





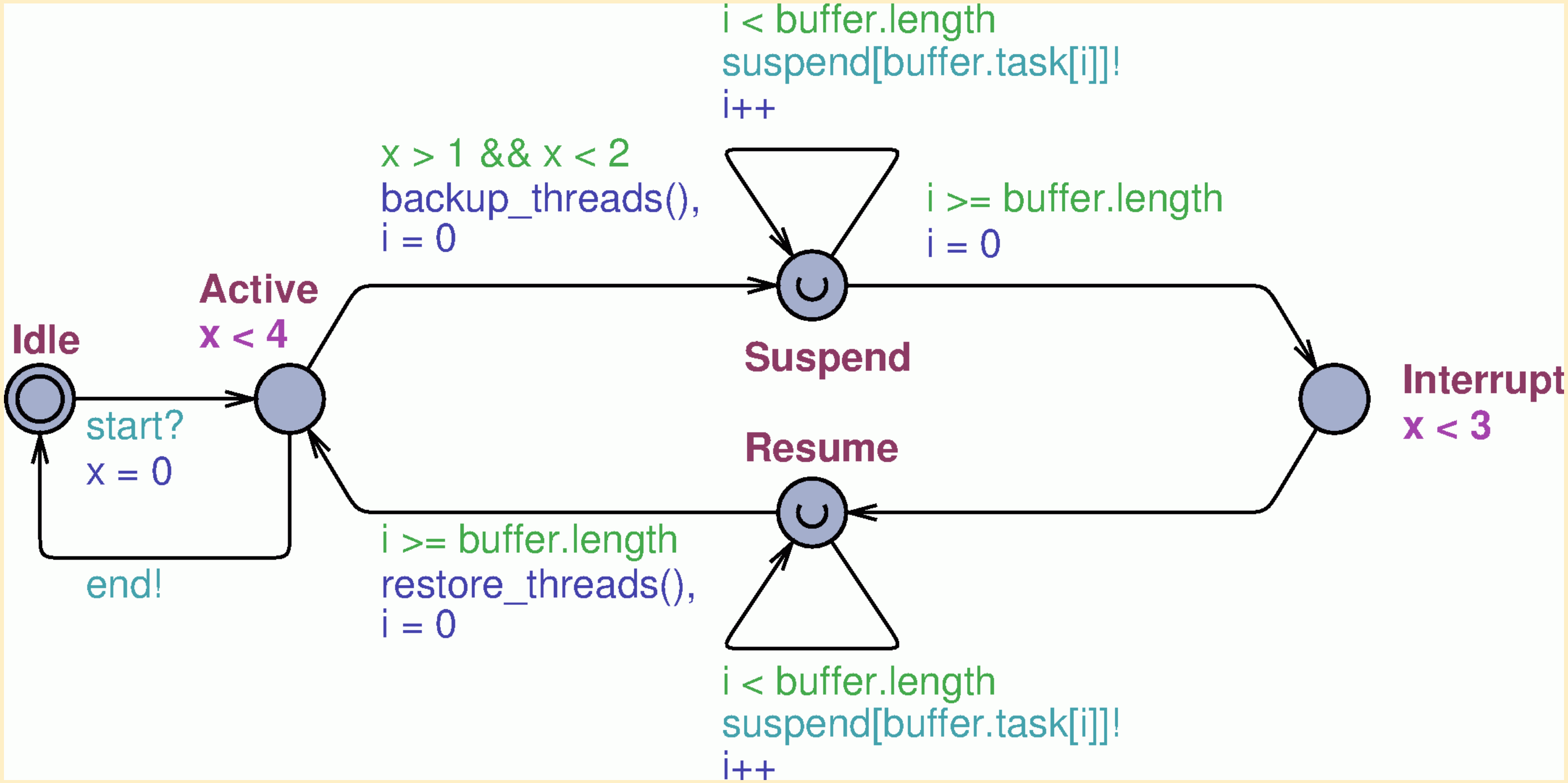

```
const int Threads = 4;

typedef struct {
  int[0,Threads] num;
  t_id task[Threads];
} thread_t;

thread_t thread;
```

```
chan apply, decline;
urgent chan allow[Tasks], forbid[Tasks];
chan suspend[Tasks];
broadcast chan start, end;

void reset_threads() {
  while (thread.num > 0)
    thread.task[thread.num-- - 1] = 0;
}
```

Let’s make the model more advanced: Interrupts!

- An interrupt can occur in the active state of the CPU
  - Certain tasks can be interrupted (preemptive)
  - CPU requirement of the interrupt determines which tasks will be interrupted
  - At least as many tasks must be suspended (starting with the lowest priorities), such that enough CPU capacity will be available (the CPU requirements of the interrupt and the remaining tasks must be at most 100%)
    - The CPU selects the tasks to be suspended
    - It also notifies the suspended tasks
    - These tasks change to suspended state
  - After the interrupt
    - The CPU notifies the previously interrupted tasks
    - These tasks change to running state
    - The CPU also changes to running state

Modeling interrupt



- Tasks are suspended by function backup_threads(), and restored by function restore_threads()
- Tasks are notified individually on separate channels about suspending and restoring

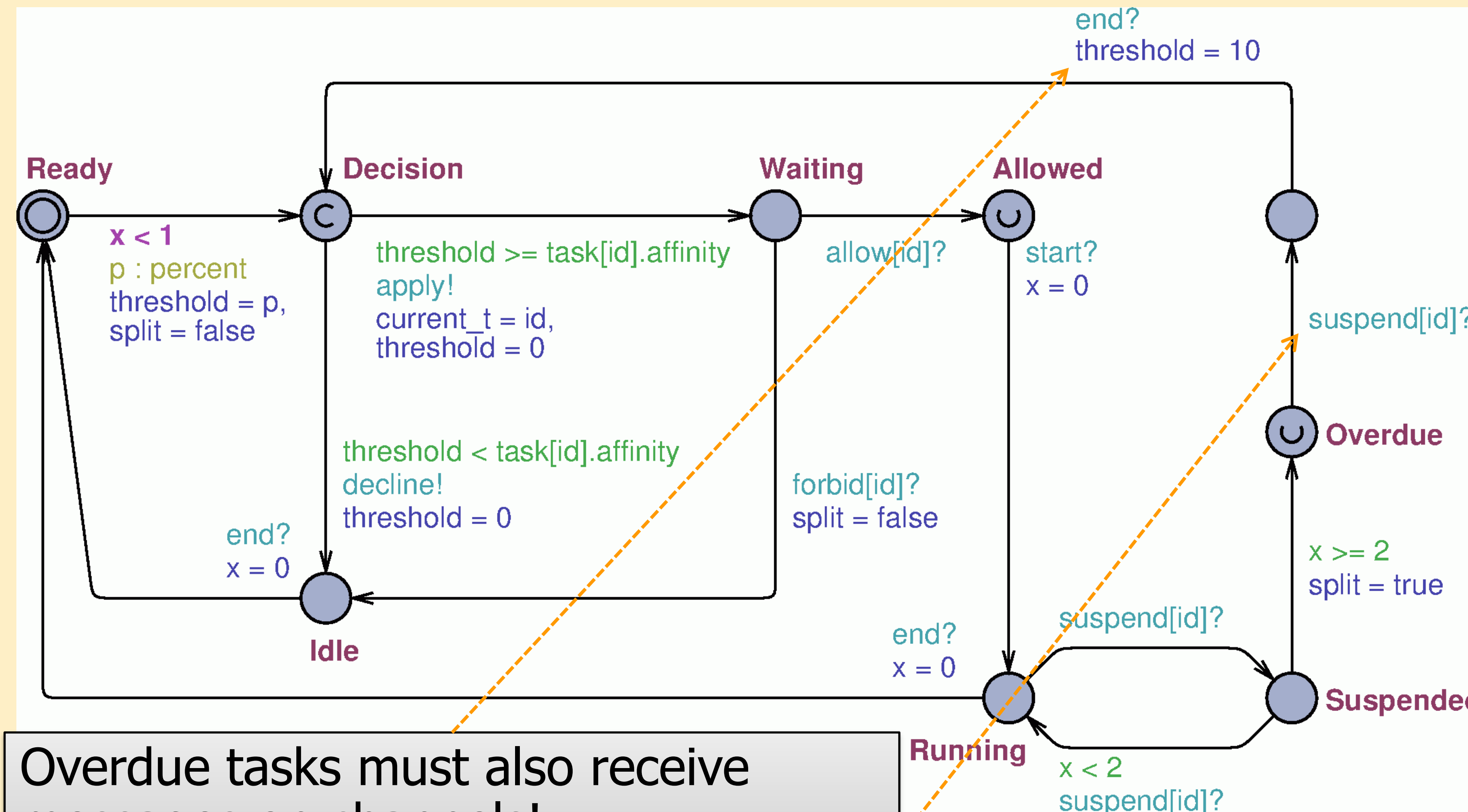
Selecting tasks for suspending

```
void backup_threads() {
  int i, p;
  t_id tid;
  for (i = 0, p = 0; i < thread.num; i++)
    p += task[thread.task[i]].demand;
  buffer.length = 0;
  for (i = 0; i < thread.num; i++) {
    if (p + i_demand > 10) {
      tid = thread.task[thread.num - i - 1];
      buffer.task[buffer.length++] = tid;
      thread.task[thread.num - i - 1] = 0;
      p -= task[tid].demand;
    }
  }
  thread.num -= buffer.length;
}
```

Even more advanced: Overdue tasks

- When tasks are suspended for too long, they will be overdue and they cannot be completed in the current period
- Such overdue tasks will try to continue running in the next period
- This is modeled by changing to the state where they apply for running (after the end signal)
  - (i.e., they skip the random choice of applying or declining)

Extending the model of a task with overdue

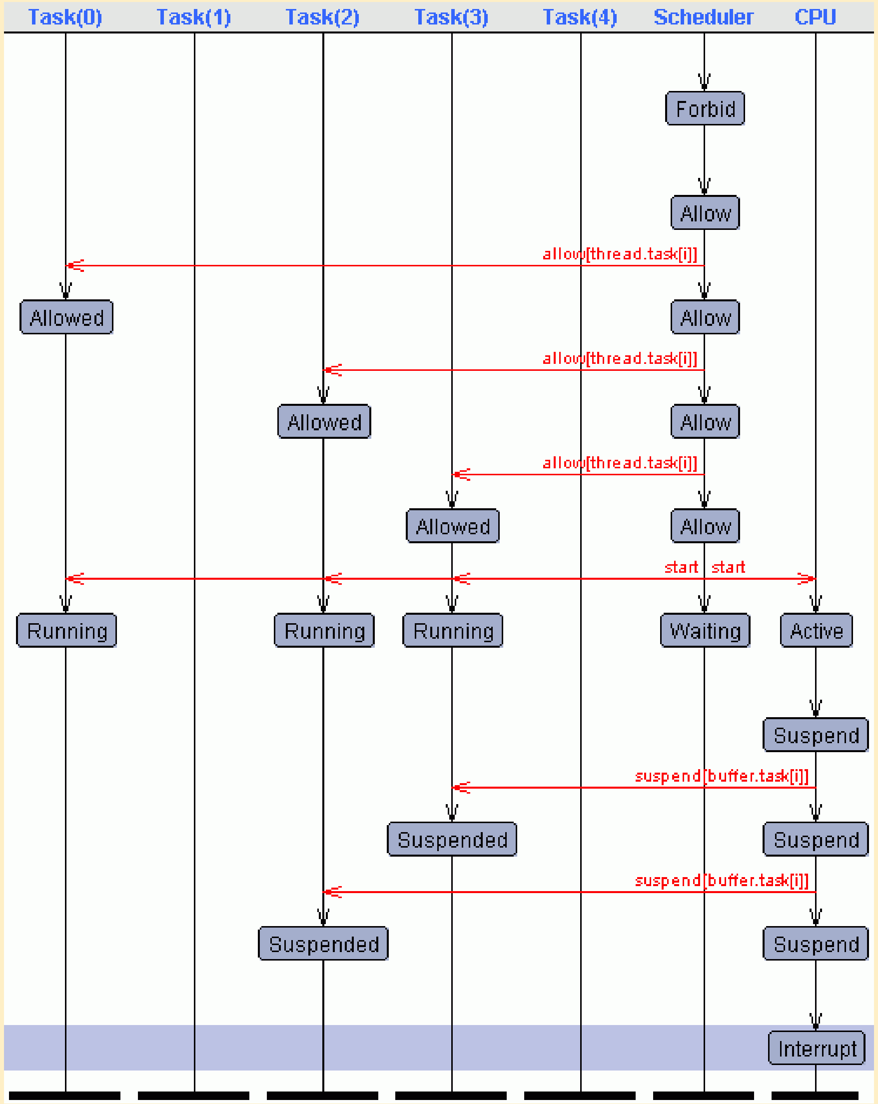


Overdue tasks must also receive messages on channels!

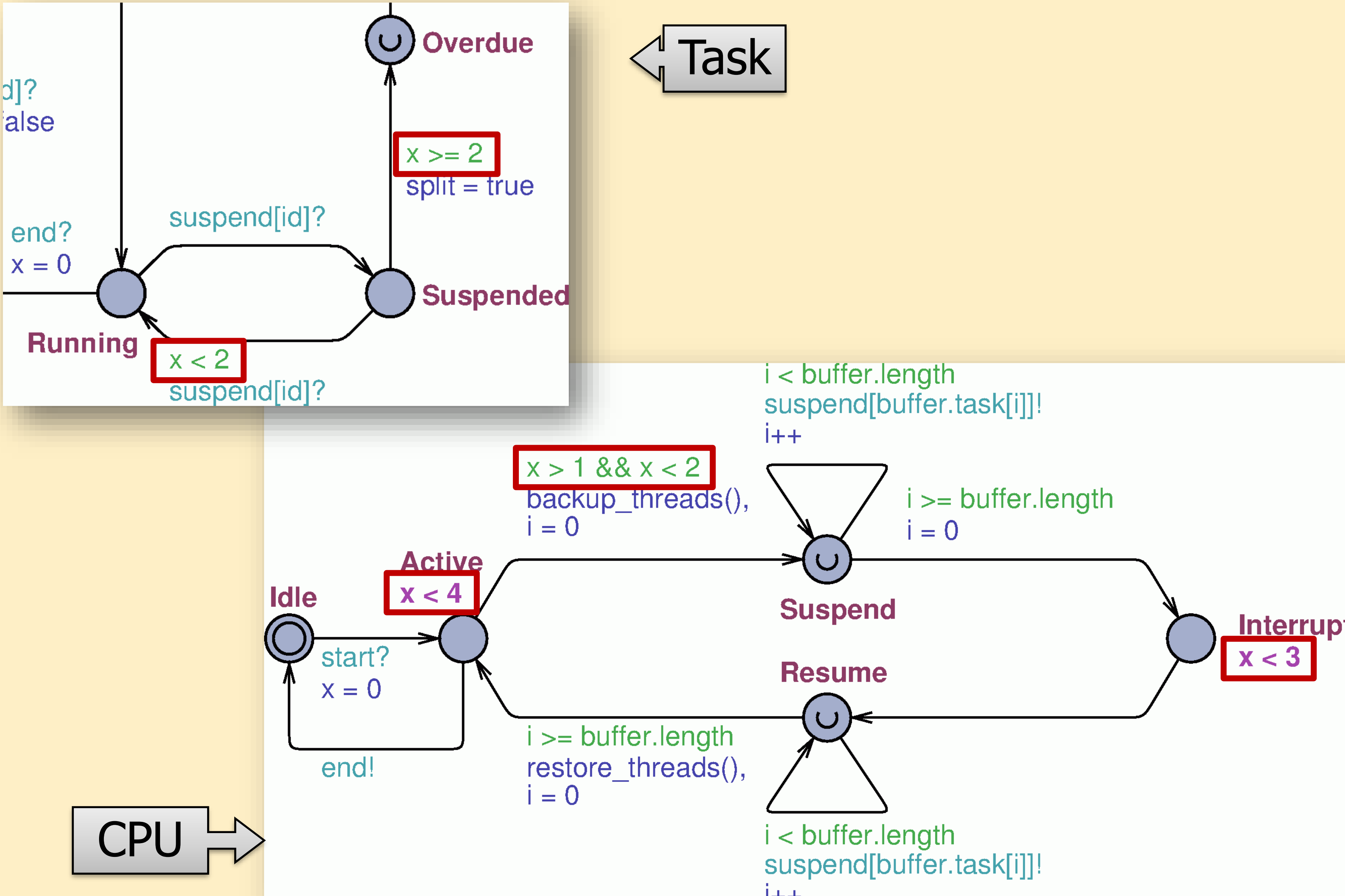
A value of 10 ensures application.

We must introduce time limits

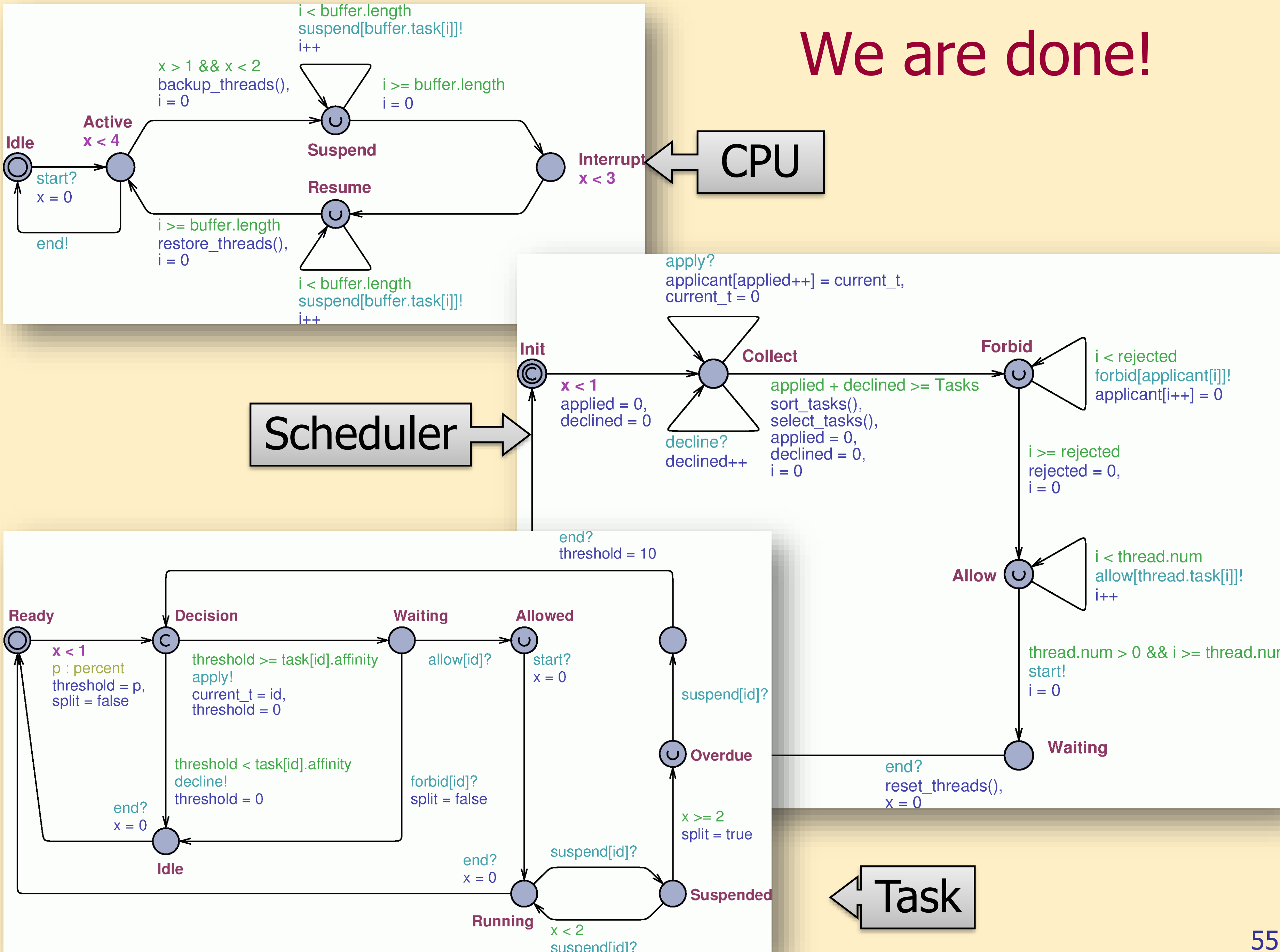
- A task has the following time limits:
  - Clocks of the selected tasks, the scheduler and the CPU start at the same time
  - The CPU can be active for at most 4 time units
  - An interrupt can occur between the 1st and 2nd time units
  - The interrupt must last at most until the 3rd time unit
  - Suspended tasks become overdue after the 2nd time unit



Time limits in the model



We are done!



CPU
Idle
Active
x < 4
start?
x = 0
end!
x > 1 && x < 2
backup_threads(),
i = 0
i < buffer.length
suspend[buffer.task[i]]!
i++
Suspend
i >= buffer.length
i = 0
Interrupt
x < 3
Resume
i >= buffer.length
restore_threads(),
i = 0
i < buffer.length
suspend[buffer.task[i]]!
i++
Scheduler
Init
x < 1
applied = 0,
declined = 0
apply?
applicant[applied++] = current_t,
current_t = 0
Collect
decline?
declined++
applied + declined >= Tasks
sort_tasks(),
select_tasks(),
applied = 0,
declined = 0,
i = 0
Forbid
i < rejected
forbid[applicant[i]]!
applicant[i++] = 0
i >= rejected
rejected = 0,
i = 0
Allow
i < thread.num
allow[thread.task[i]]!
i++
thread.num > 0 && i >= thread.num
start!
i = 0
Waiting
end?
reset_threads(),
x = 0
Task
Ready
x < 1
p : percent
threshold = p,
split = false
Decision
threshold >= task[id].affinity
apply!
current_t = id,
threshold = 0
threshold < task[id].affinity
decline!
threshold = 0
Waiting
allow[id]?
forbid[id]?
split = false
Allowed
start?
x = 0
end?
threshold = 10
suspend[id]?
Overdue
x >= 2
split = true
end?
x = 0
Idle
end?
x = 0
Running
suspend[id]?
Suspended
x < 2
suspend[id]?

Requirements to be verified

1. The model is deadlock free.
2. It is possible that an applied task has to be rejected.
3. It is possible that all threads are busy, i.e., maximal number of tasks are running.
4. If a task is running, the number of busy threads in the global data structure is greater than 0.
5. It is possible that the CPU suspends more than 2 threads due to an interrupt.
6. There is a path where no task is suspended in all of the periods, but it is not possible for all paths, i.e., there is at least one path where at least one task is suspended at least once.
7. It is not possible that a task is in suspended state after the 3rd time unit.
8. If a task is suspended, it may be completed eventually.

Temporal expressions to be verified

Overview

```
1. A[] not deadlock
2. E<> Scheduler.rejected > 0
   A[] Task(4).Allowed imply thread.num < 4
3. E<> CPU.Active && thread.num == 4
   E<> CPU.Interrupt && CPU.buffer.length > 1
4. A[] (exists (i : t_id) Task(i).Running) imply thread.num > 0
5. E<> CPU.Interrupt && CPU.buffer.length > 2
6. E[] (forall (i : t_id) Task(i).split == false)
   A[] (forall (i : t_id) Task(i).split == false)
7. E<> (exists (i : t_id) Task(i).Overdue && Task(i).x > 3)
8. Task(1).Overdue --> Task(1).split == true
   E<> (exists (i : t_id) Task(i).Overdue && Task(i).split == true)
```