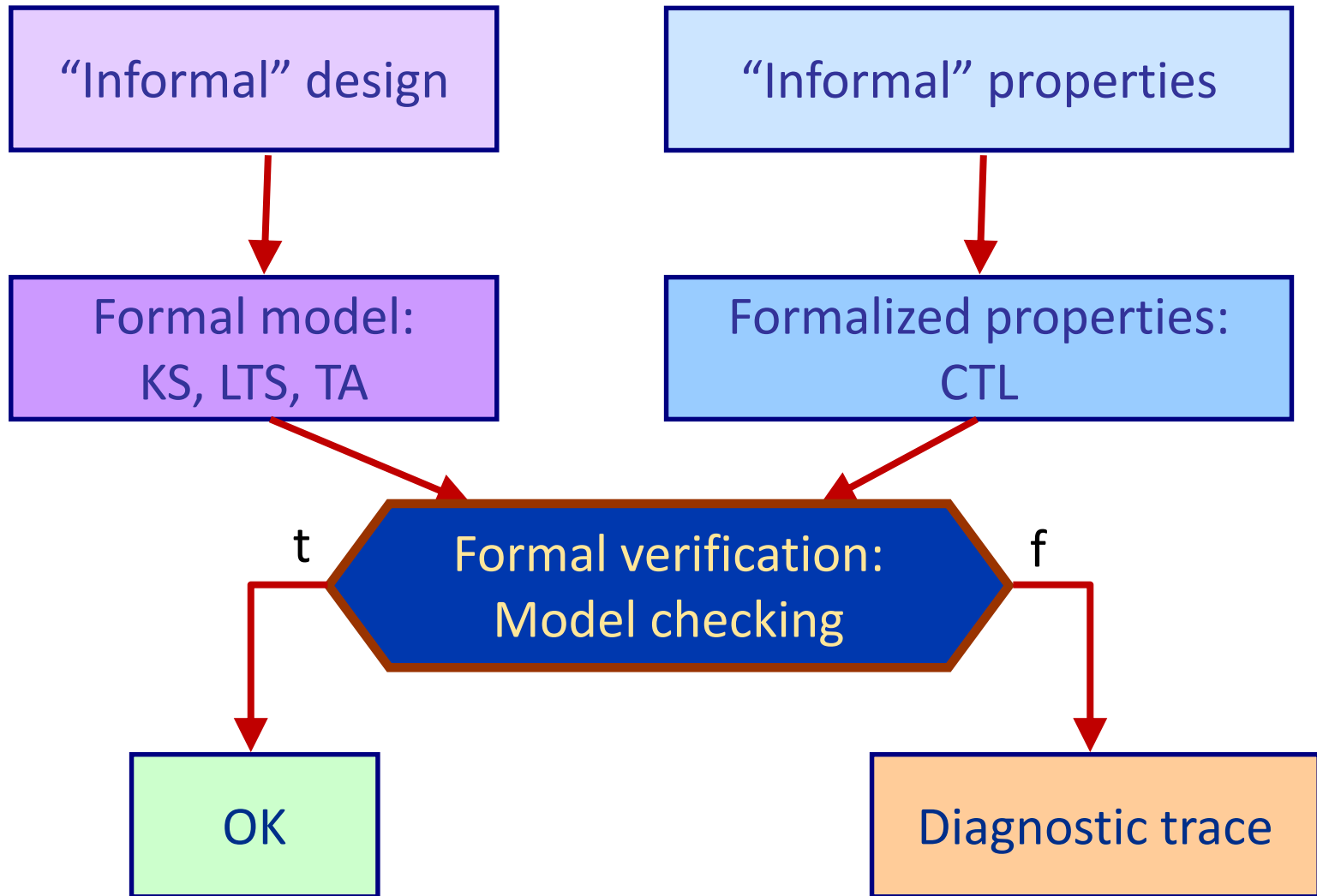


Model checking case study: Scheduling of tasks

Dr. Tamás Bartha

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Formal verification: Goals

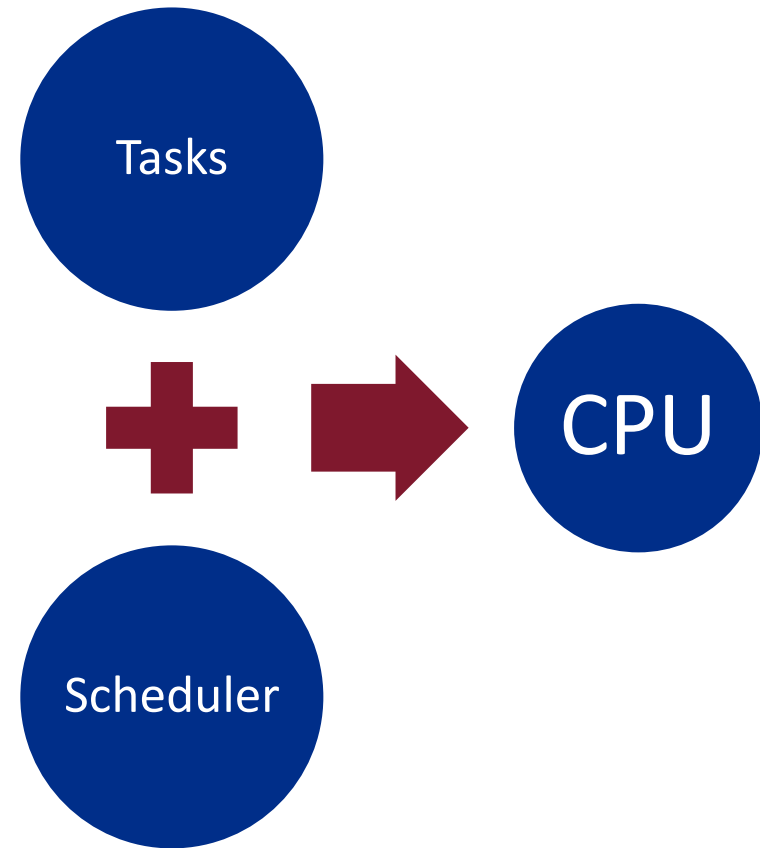


The problem

- Modeling **tasks** and **threads** in a simple operating system (OS)
 - 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 capacity
 - Limited number of OS threads can run tasks, one task may be assigned to a single thread
 - At the end of a period, tasks are stopped
 - The process above is repeated

The main components (1/2)

- Tasks and their parameters
 - **Affinity**: willingness of a task to “apply” for running
 - **Demand**: the task requires $(10 \times \text{Demand})$ percent of CPU processing capacity
 - **Priority**: priority level of the task (higher number means lower priority)



- The total CPU load (i.e., sum of the demands of the tasks that are selected for running) must be at most 100%
- Within this limit, tasks have to be selected based on their priority

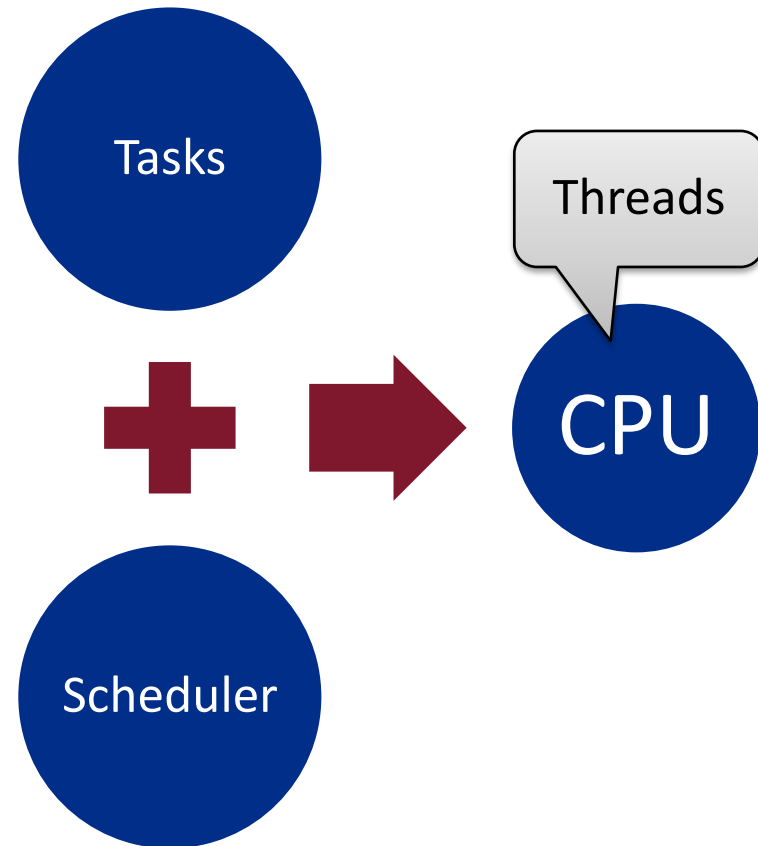
The main components (2/2)

■ Scheduler

- Selects running tasks from those that “applied” for running
- There is a limited number of threads that can run tasks
- Each task is allocated to a separate thread
- I.e., no more tasks can be running than the number of threads

■ CPU

- Capability to run threads
- Two states: active, inactive
- Threads can run in the active state
- **Extension (later): A preemptive interrupt can occur in the 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 $p \geq \text{Affinity}$, then they apply for running
 - otherwise they decline running and become inactive

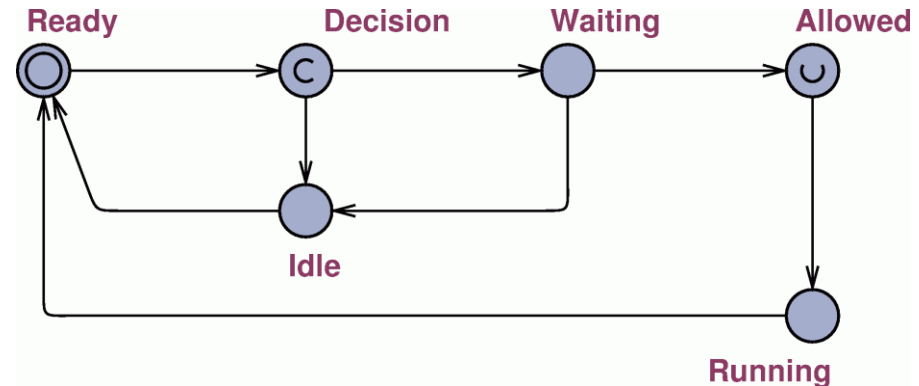
■ The scheduler

- Collects tasks that apply for running
- Processes applied tasks: orders the tasks descending by priority, while observing maximum CPU load
- Assigns the selected tasks to threads and stores this assignment in a global data structure

Let's start modeling!

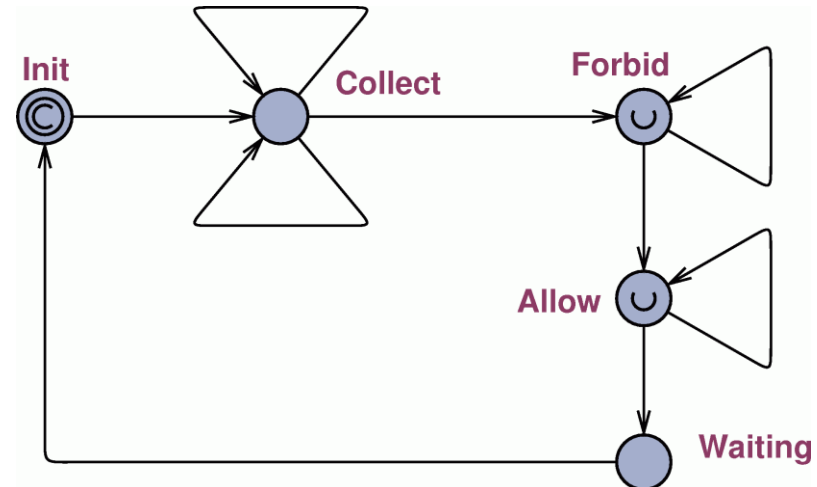
■ States of a Task

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



■ States of the Scheduler

- **Init**: initial state
- **Collect**: collects applications and declined statuses
- **Forbid**: notifies rejected tasks
- **Allow**: notifies selected tasks
- **Waiting**: waiting for period end



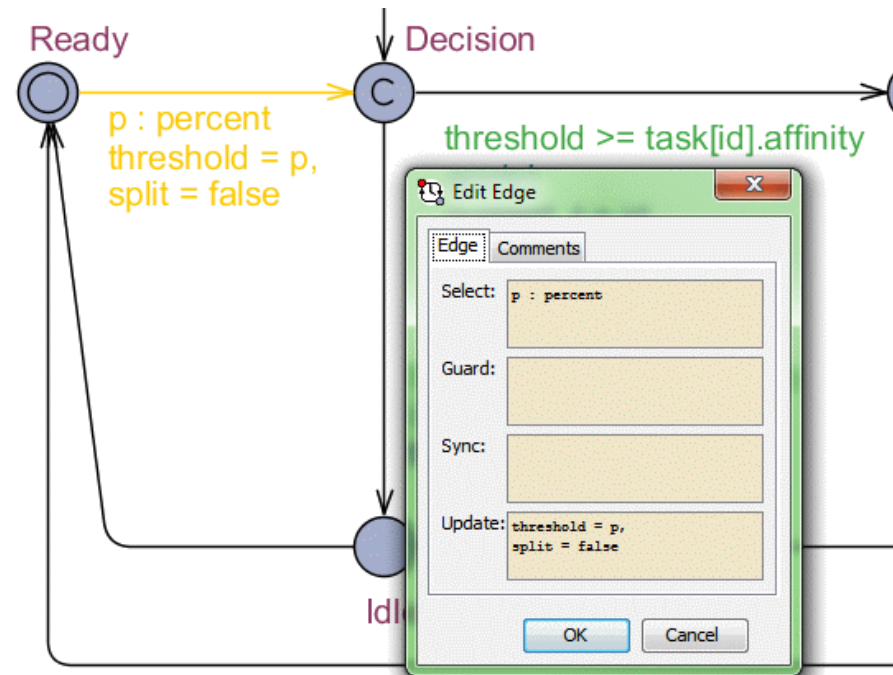
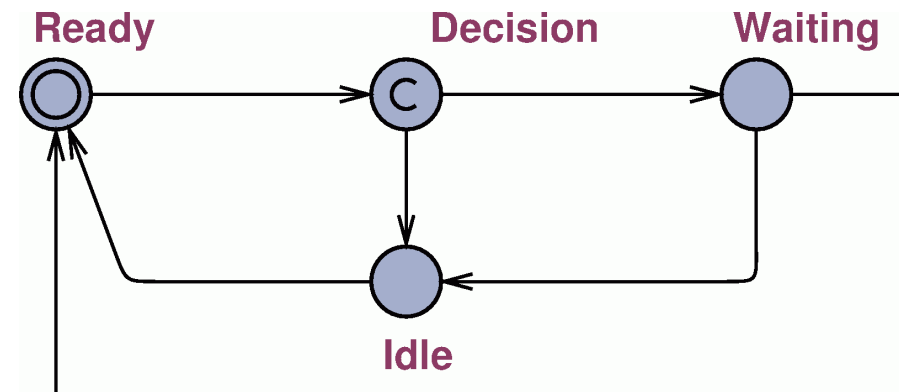
Modeling random choice: Alternatives

■ Simple solution

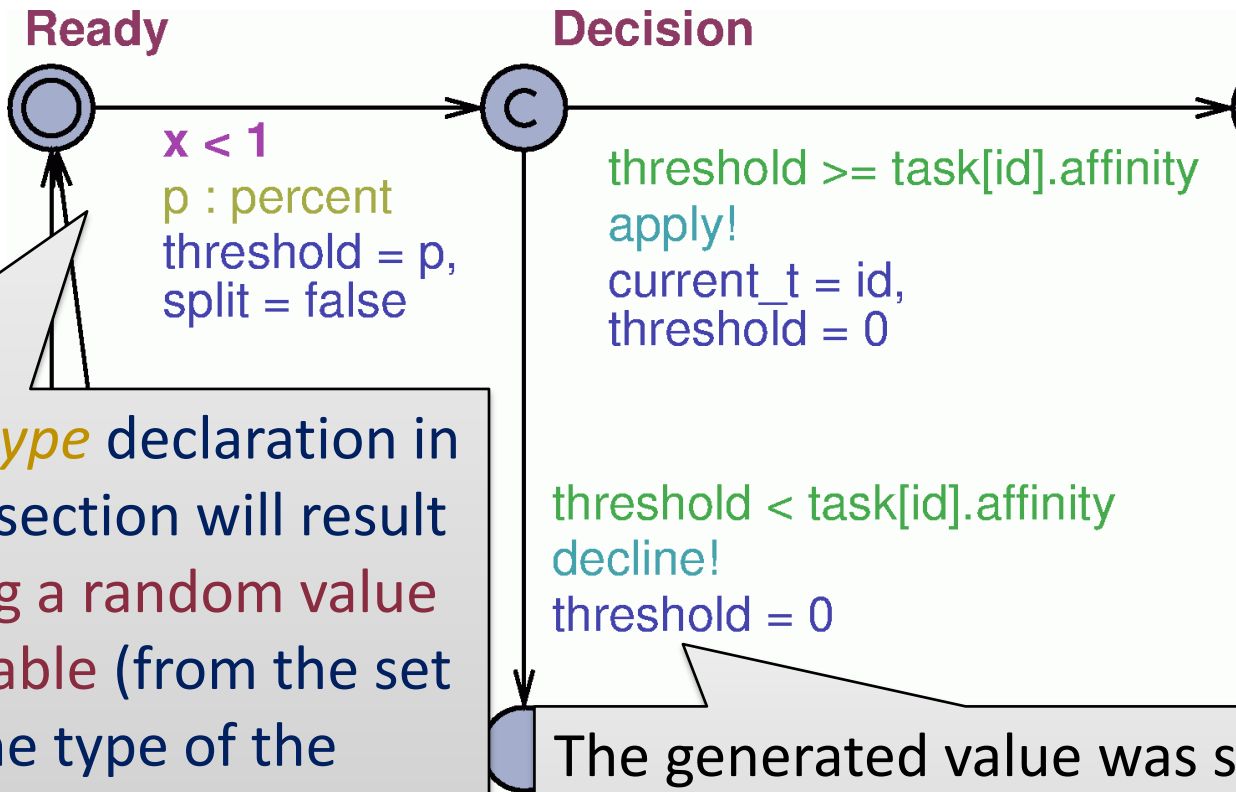
- UPPAAL chooses randomly from enabled transitions
- Is it what we want?
No, because probabilities should be proportional to the affinities of tasks

■ Correct solution

- Generate random value using the **select** construct of UPPAAL



Modeling random choice: The select construction



variable : type declaration in the Select section will result in assigning a random value to the variable (from the set given by the type of the variable) when taking the transition.

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

The generated value was stored in a local variable (here: threshold) thus it can be used in the proceeding steps. The purpose of the Committed state is that the two operations should not be interleaved by other operations.

Declarations in the model

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;

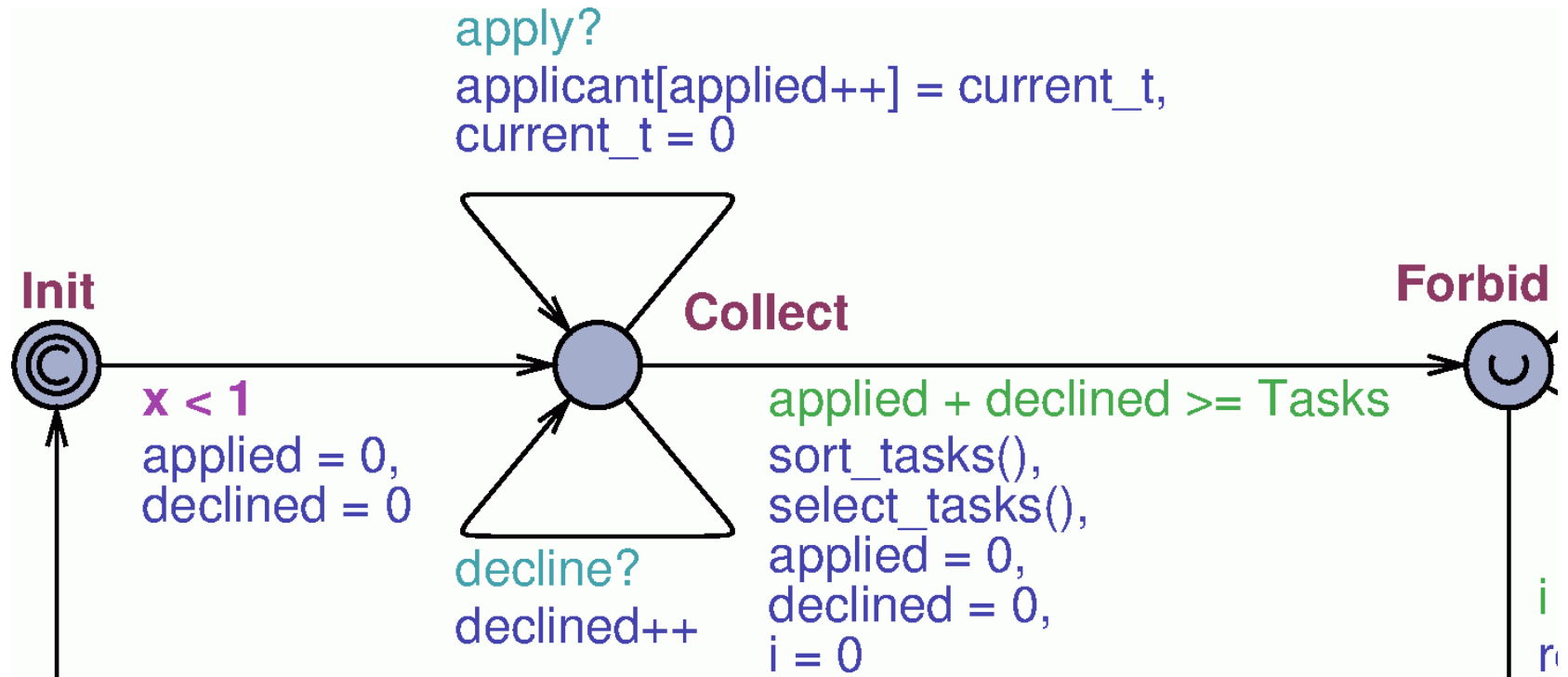
// affinity, demand, priority
const task_t task[Tasks] = {
    ...
    ...
};
```

Local (Task)

Name:	Task	Parameters:	t_id id
-------	------	-------------	---------

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

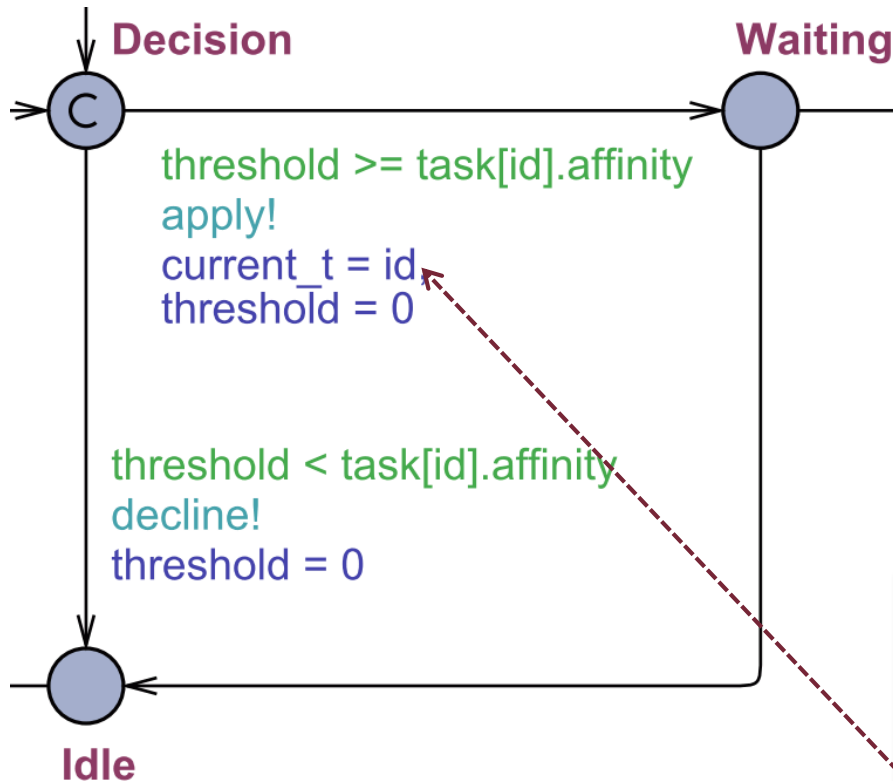
How does counting the tasks work in the scheduler?



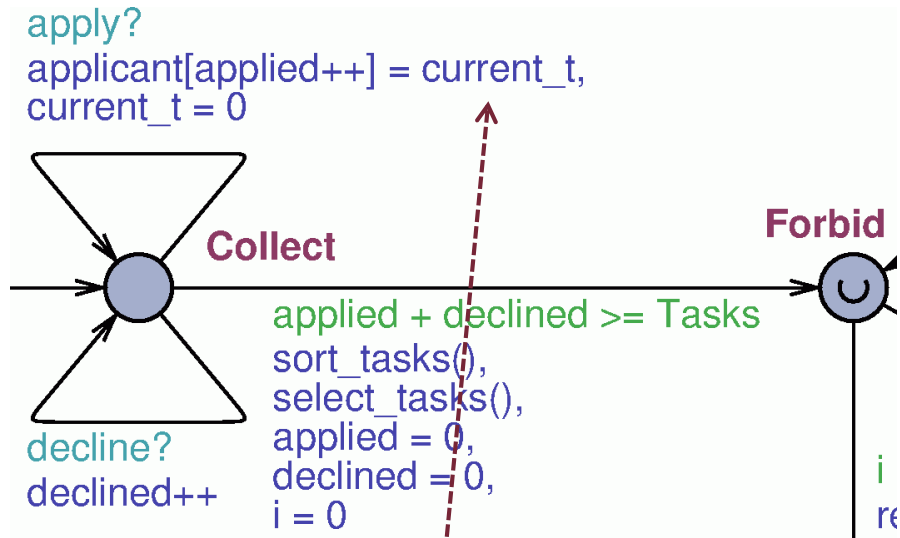
- We are staying in state **Collect** until each task either applied or declined
- Applied tasks are stored in a local array `applicant[]`
- Functions `sort_tasks()` and `select_tasks()` are selecting tasks when entering the state **Forbid**

Collecting applied and declined tasks

Task



Scheduler

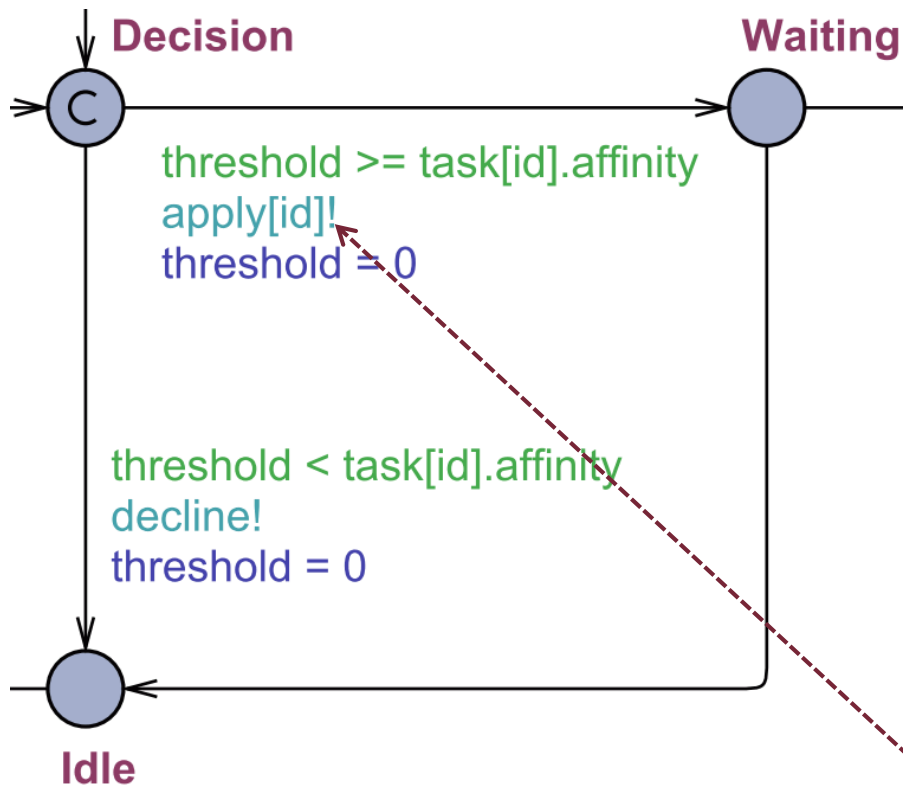


Using **synchronous communication** and maintaining a global variable.

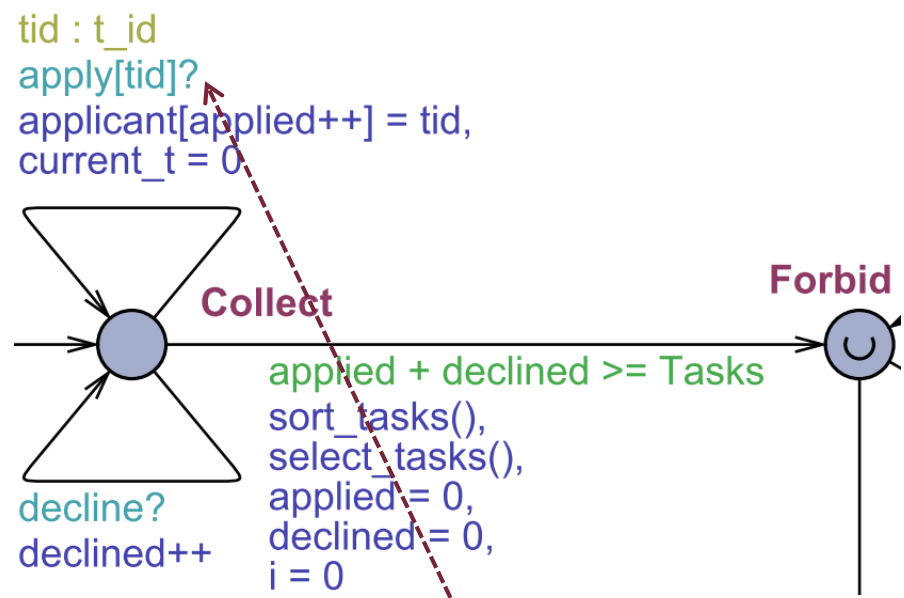
It is guaranteed by the model semantics that the **update part of the sender** is executed first!

Collecting applied and declined tasks: other solution

Task



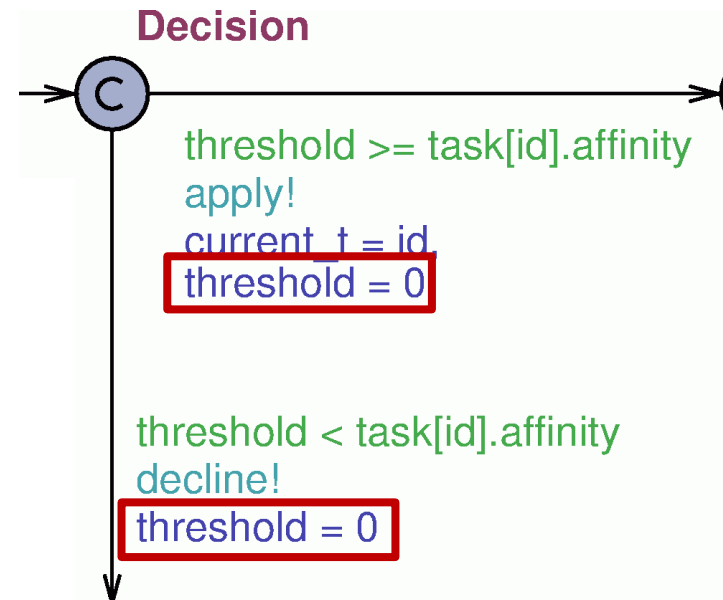
Scheduler



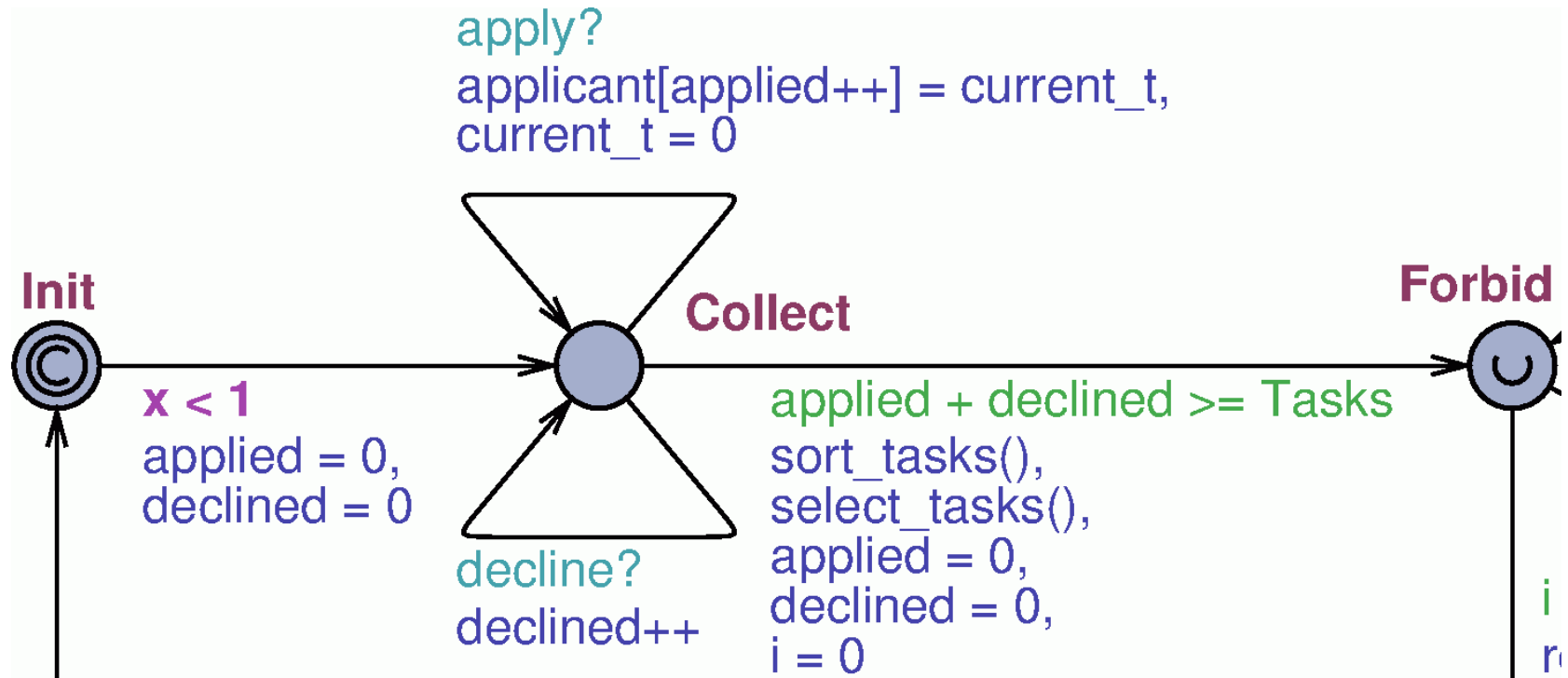
Modeling synchronous communication using an **array of channels**.
Task id is available as channel id.

Why should we reset temporary variables?

- Temporary variable
 - In the state space: A set of trajectories for each potential value
 - Results in large state space
 - Can be reduced by resetting the variable
- Other issue: Interleaving
 - Between two successive transitions of an automaton, the transitions of another concurrent automata may occur
 - It can be controlled by **committed states**



Let's get back to the model of the scheduler!



- Applied tasks are stored in a local array `applicant[]`
- Functions `sort_tasks()` and `select_tasks()` are selecting tasks
 - Ordering tasks decreasing by their priority and CPU requirement, while observing the limit of CPU load

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 CPU load limit is reached

- Let the task 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 (priority levels):

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

- Selected: 0, 2, 3

- Rejected: 4

Ordering tasks based on CPU requirement

```
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;
    }
}
```

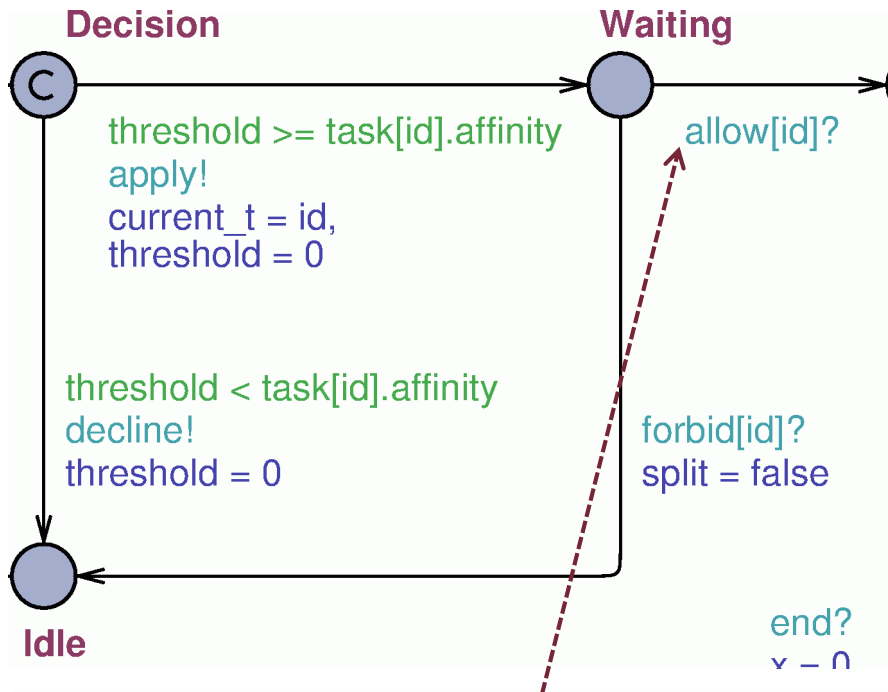
```
void insert_at(p_level pri, int[0,Tasks] pos, t_id tid) {
    int i;
    for (i = buffer[pri].length; i > pos; i--) {
        buffer[pri].task[i] = buffer[pri].task[i - 1];
    }
    buffer[pri].task[pos] = tid;
    buffer[pri].length++;
}
```

Selecting tasks while observing limits

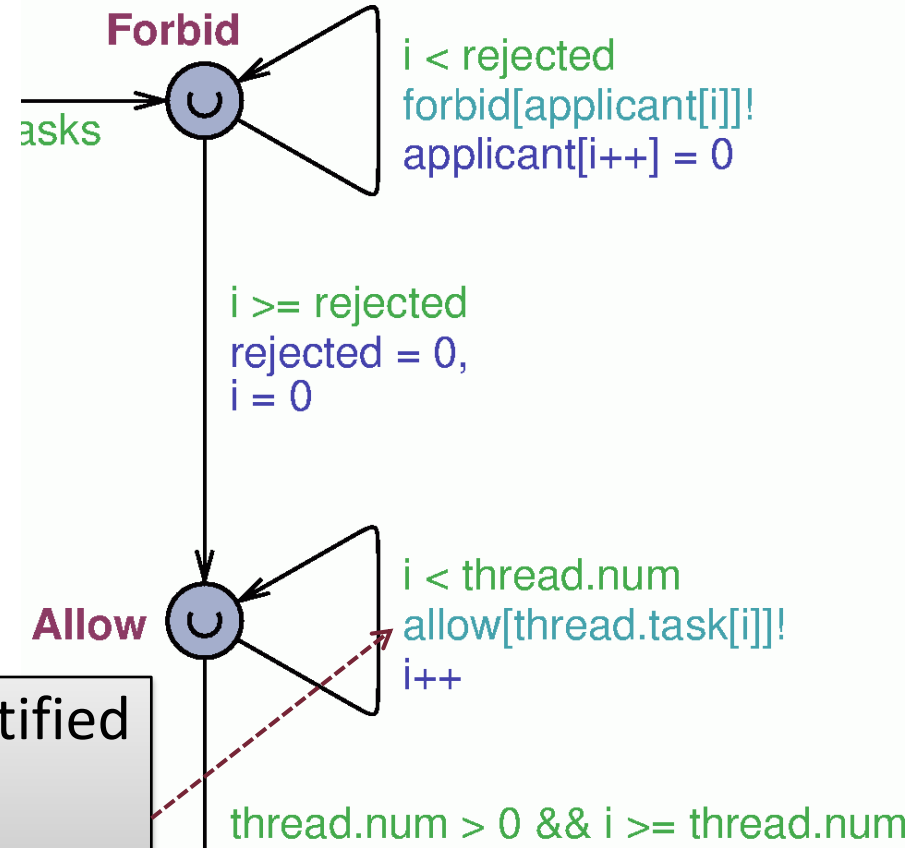
```
void select_tasks() {
    int i, pri;
    percent p = 0;           // Collecting demands
    rejected = 0;           // Number of rejected tasks
    thread.num = 0;
    for (pri = 0; pri < Levels; pri++) {
        for (i = 0; i < buffer[pri].length; i++) {
            if (p + task[buffer[pri].task[i]].demand <= 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

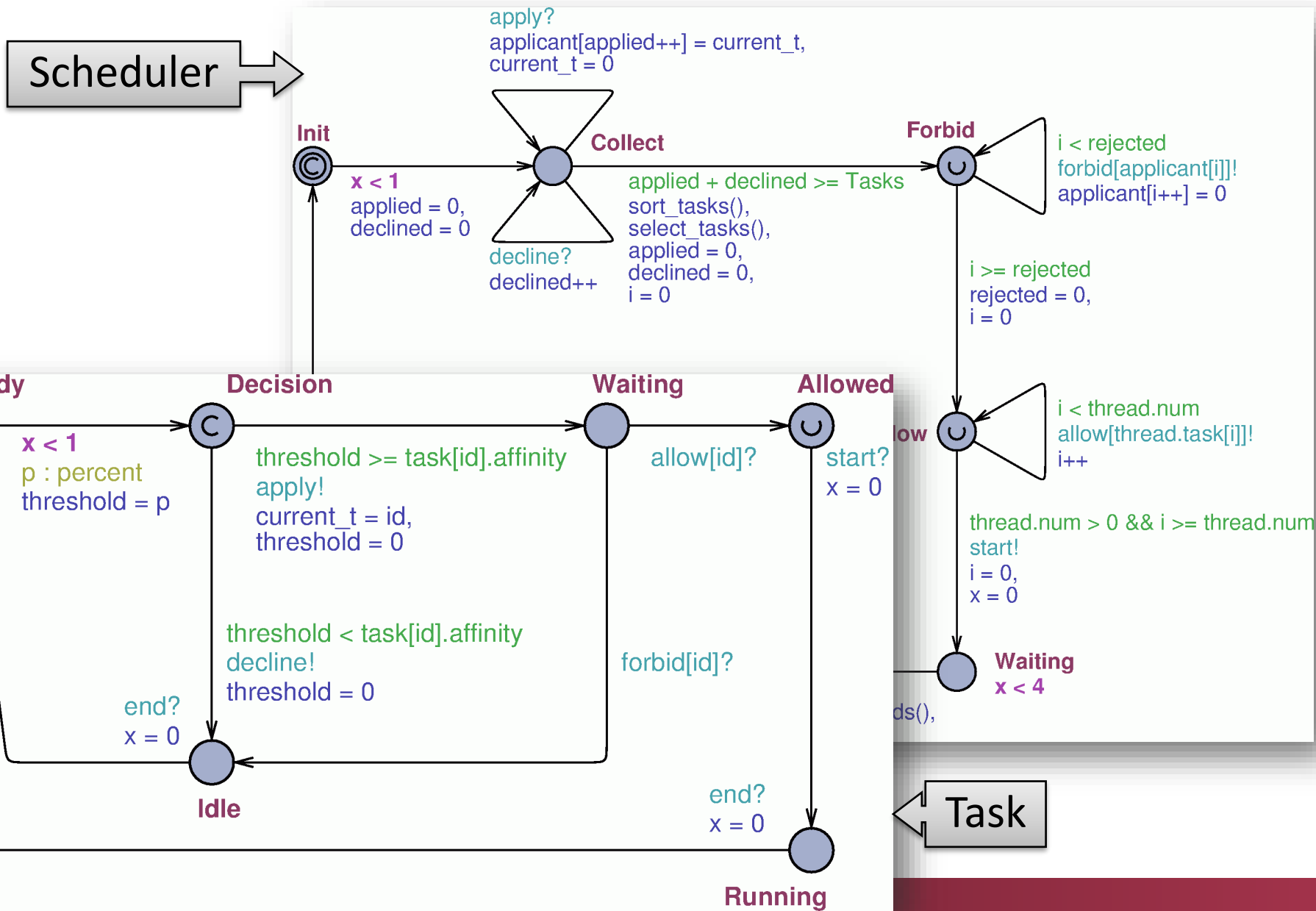


Scheduler



Selected and rejected tasks are notified individually on separate channels.
Temporary variables are reset.

The model already works (v1, without a CPU)



Intermediate checking of the model

- We already have a functional system
 - It is recommended to check this intermediate system
- Some requirements (with 4 threads):
 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. If a task is running then a thread is occupied.

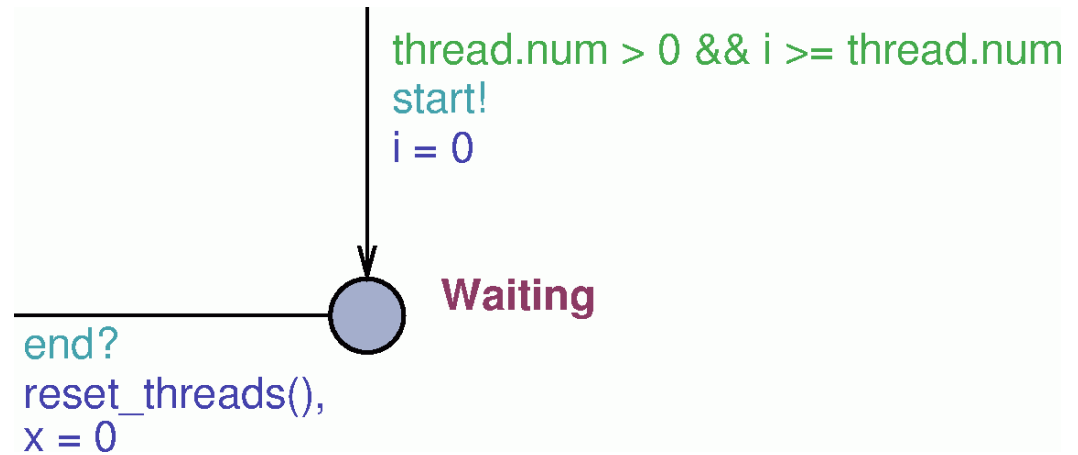
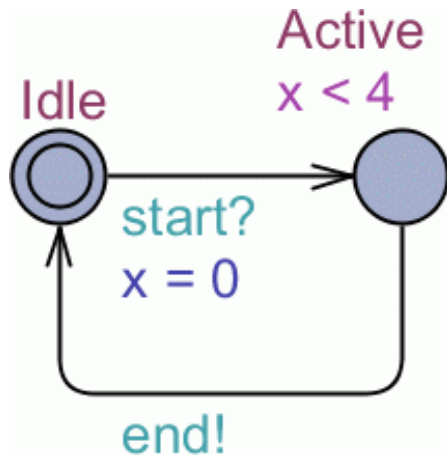
The screenshot shows a software interface with three tabs: "Editor", "Simulator", and "Verifier". The "Verifier" tab is active. Below the tabs is a section titled "Overview" containing a list of requirements. Each requirement is followed by a green circular status indicator. To the right of the list are four buttons: "Check", "Insert", "Remove", and "Comments".

Requirement	Status
A[] not deadlock	Green
E<> Scheduler.rejected > 0	Green
A[] Task(4).Allowed imply thread.num < 4	Green
E<> Task(0).Running && thread.num == 4	Green
A[] (exists (i : t_id) Task(i).Running) imply thread.num > 0	Green

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;
```

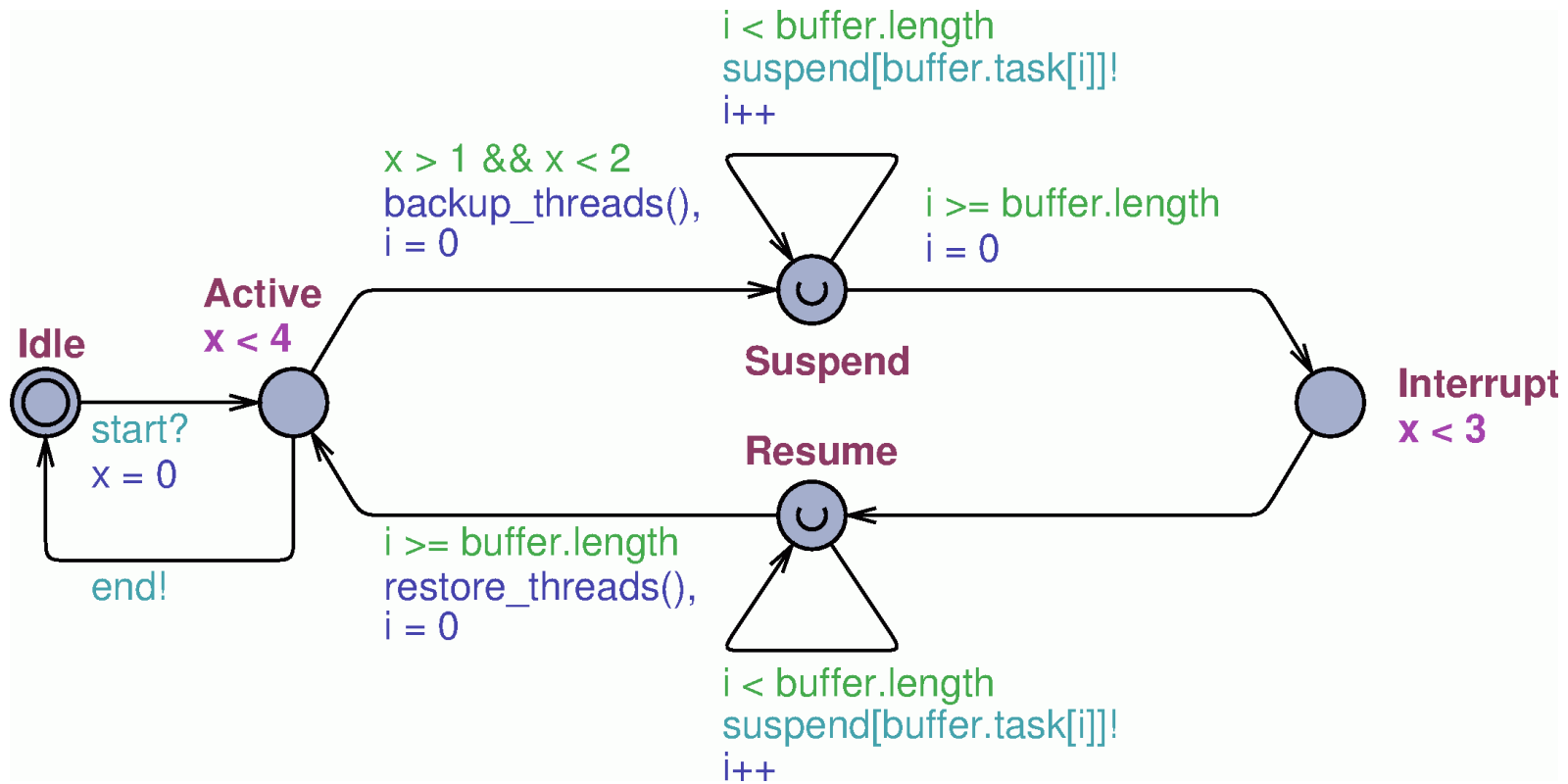
```
chan apply, decline;  
urgent chan allow[Tasks], forbid[Tasks];  
broadcast chan start, end;  
chan suspend[Tasks];
```

```
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) thus suspended
 - CPU load for serving the interrupt determines which tasks will be interrupted
 - At least as many tasks must be suspended (starting with the lowest priorities), that result in enough CPU capacity for the interrupt (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 an interrupt (v2)



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

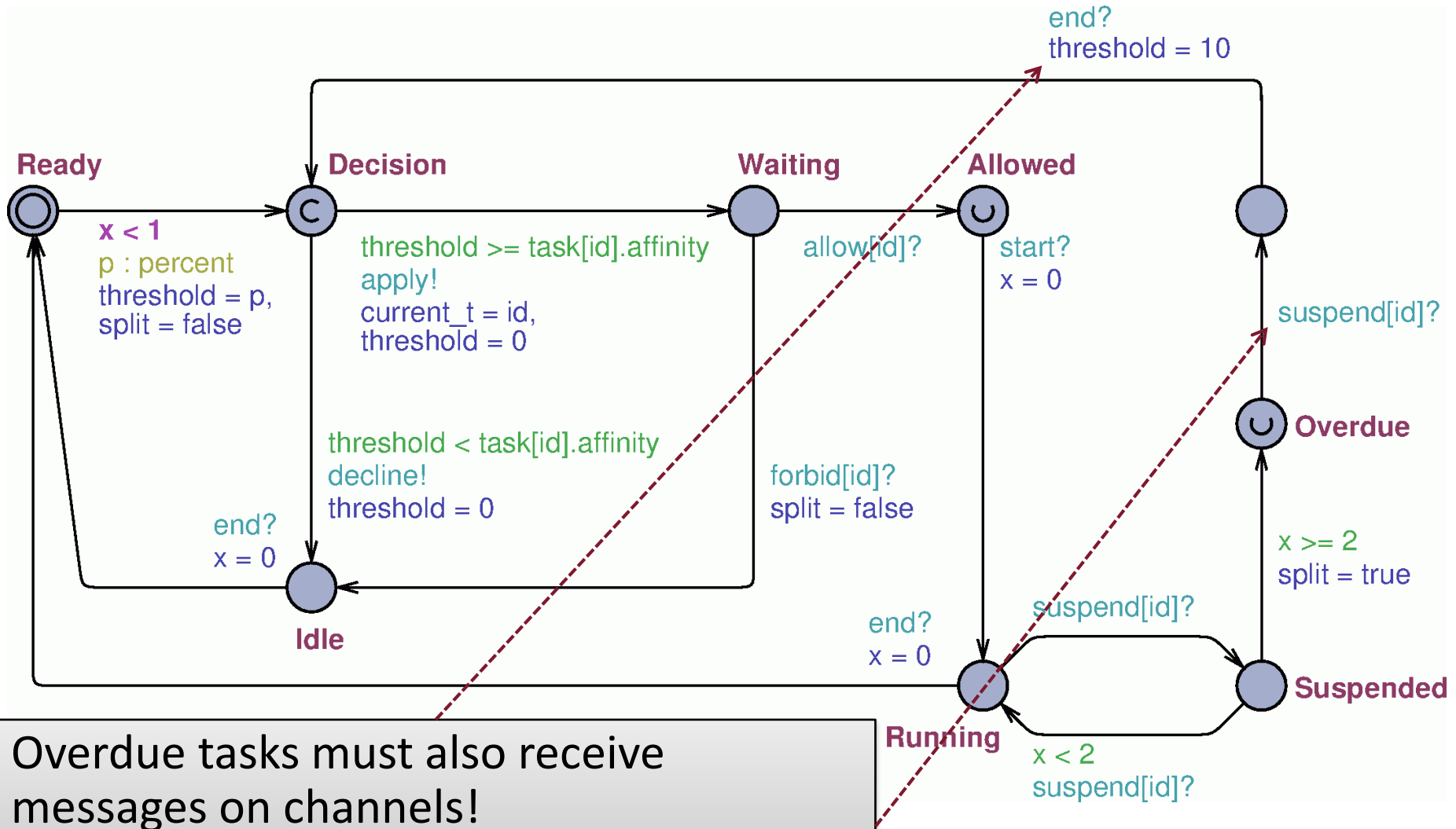
Selecting tasks for suspending

```
void backup_threads() {
    int i, p;
    t_id tid;
    for (i = 0, p = 0; i < thread.num; i++)
        p += task[thread.task[i]].demand;           // Collecting demands
    buffer.length = 0;
    for (i = 0; i < thread.num; i++) {
        if (p + i_demand > 10) {                   // Demand over limit
            tid = thread.task[thread.num - i - 1];
            buffer.task[buffer.length++] = tid;     // From thread to buffer
            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 moving to the state where they surely apply for running (after the end signal)
 - I.e., they skip the random choice of applying or declining

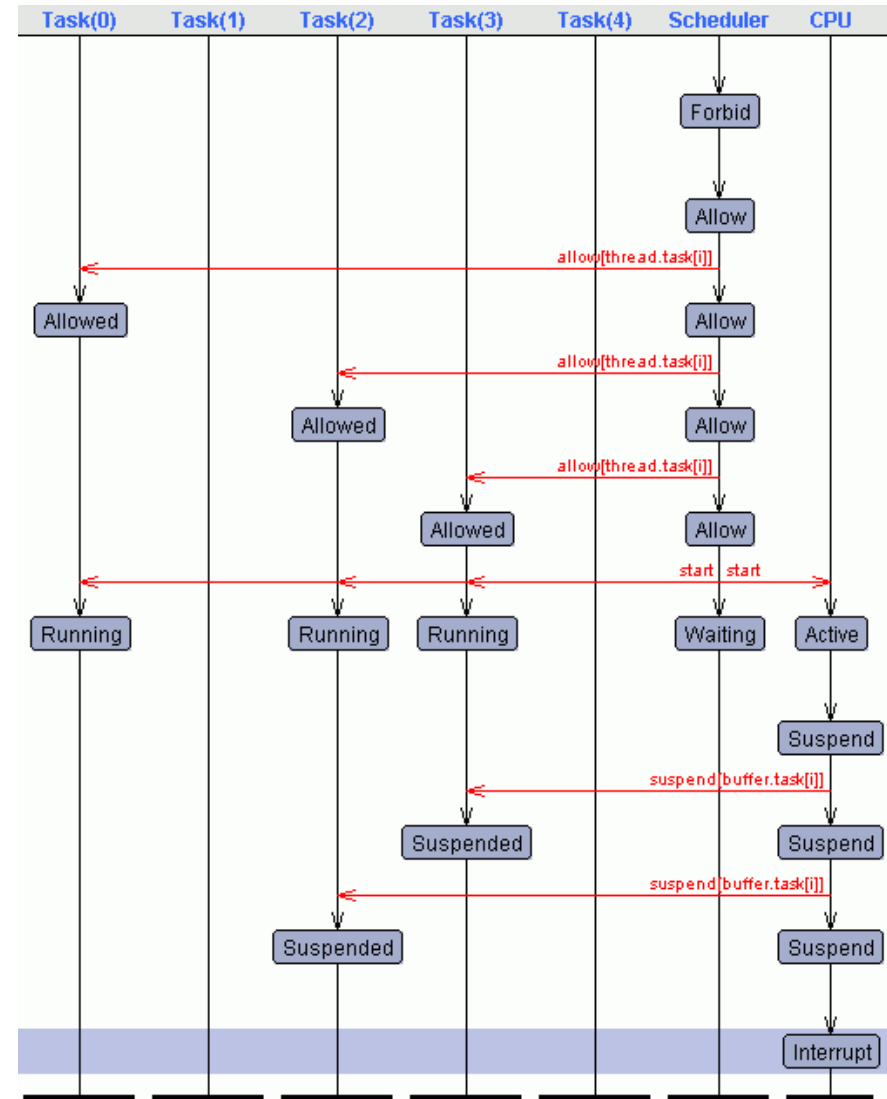
Extending the model of a task with overdue



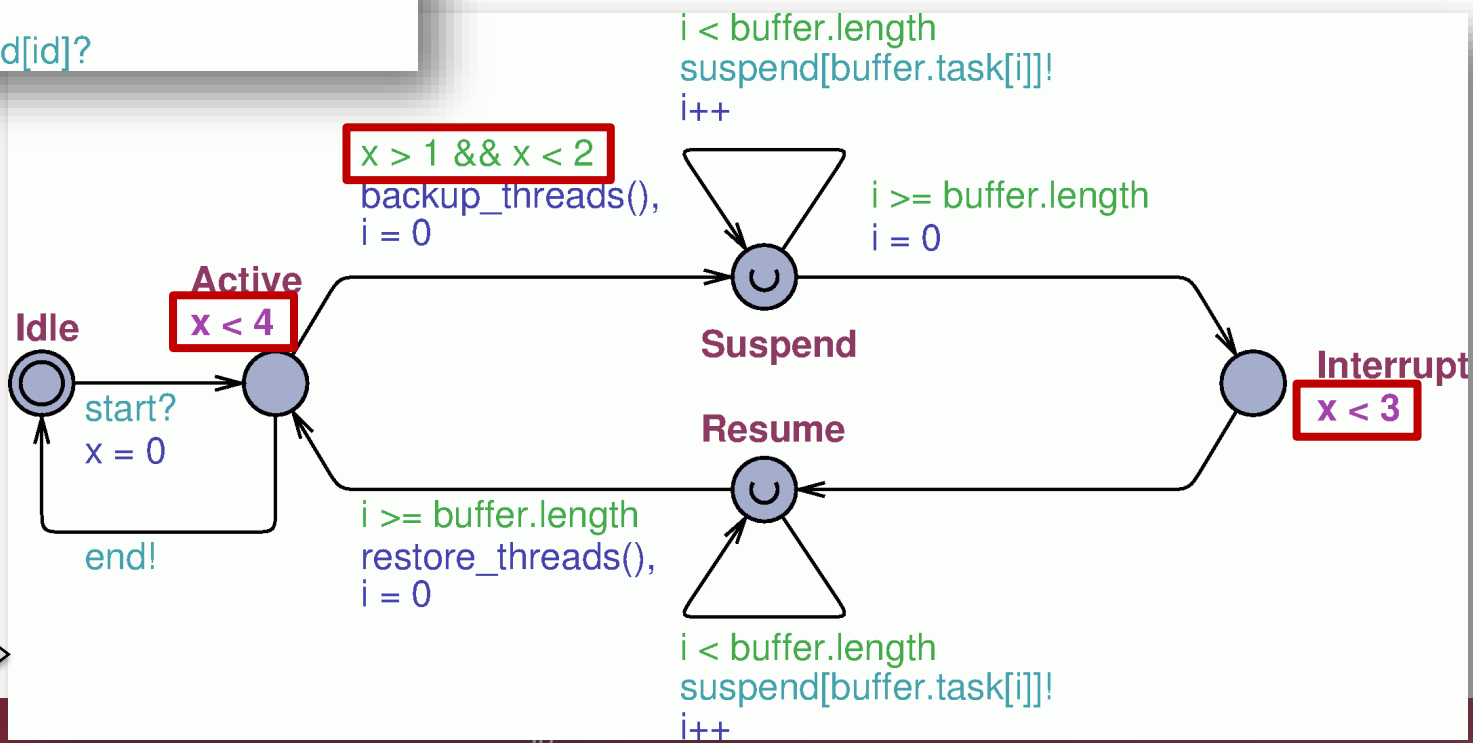
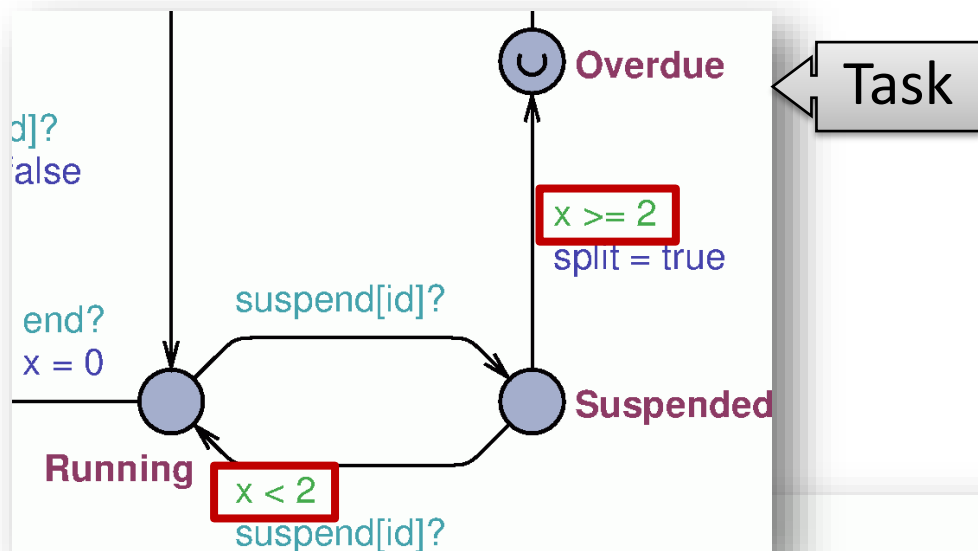
Overdue tasks must also receive messages on channels!
A threshold of 10 ensures application

We must introduce time limits

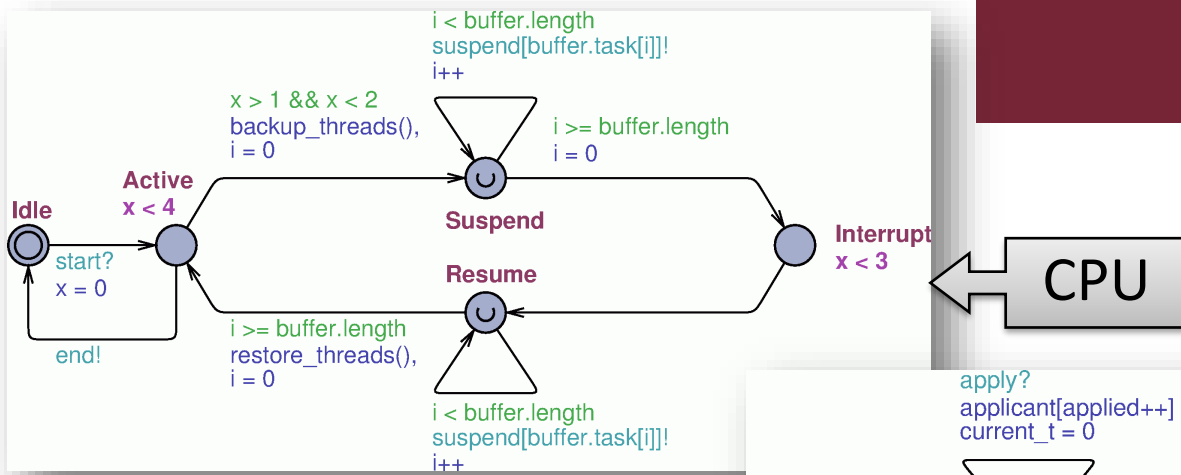
- A task has the following time limits
 - Clocks of the selected tasks, the scheduler and the CPU start at the same time
 - The CPU can be active for at most 4 time units
 - An interrupt can occur between the 1st and 2nd time units
 - The interrupt must last at most until the 3rd time unit
 - Suspended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



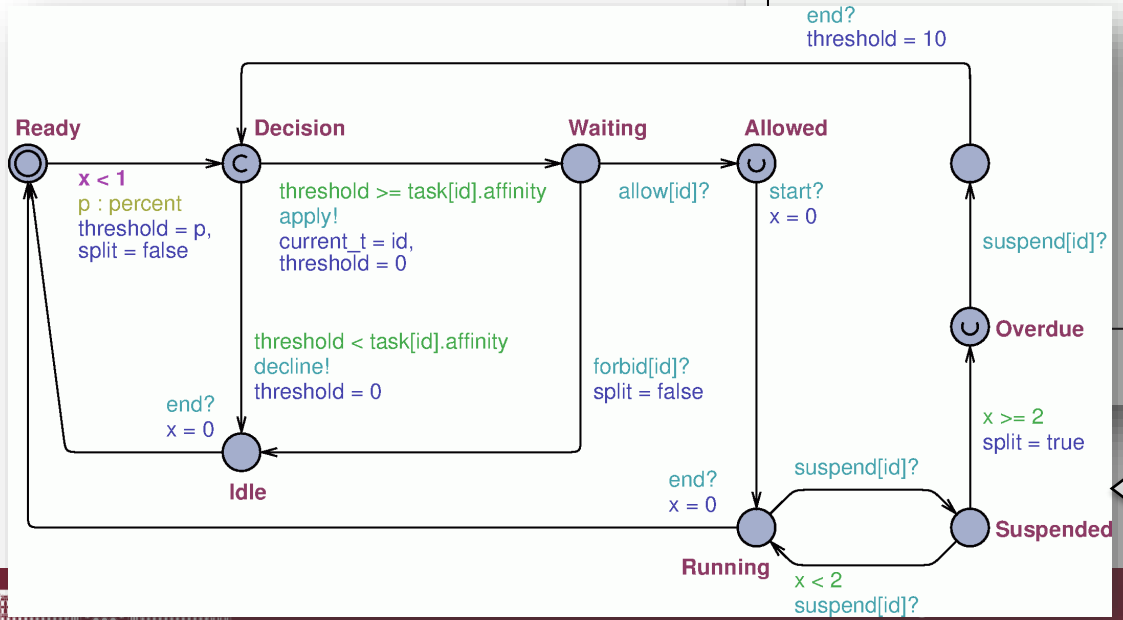
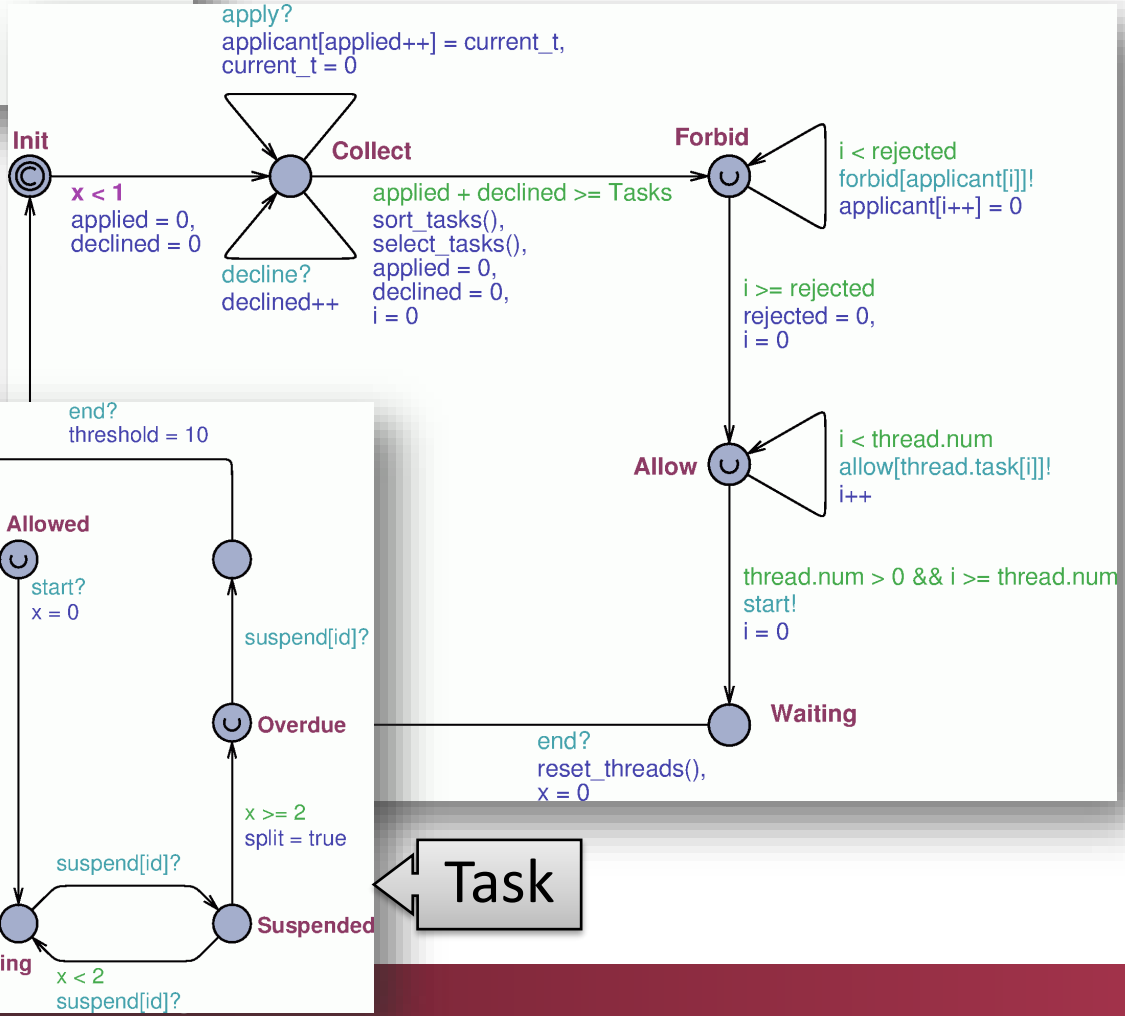
Time limits in the model



We are ready!



Scheduler












Task

Requirements to be verified (examples)

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 an execution 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.

Temporal logic expressions 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)` 