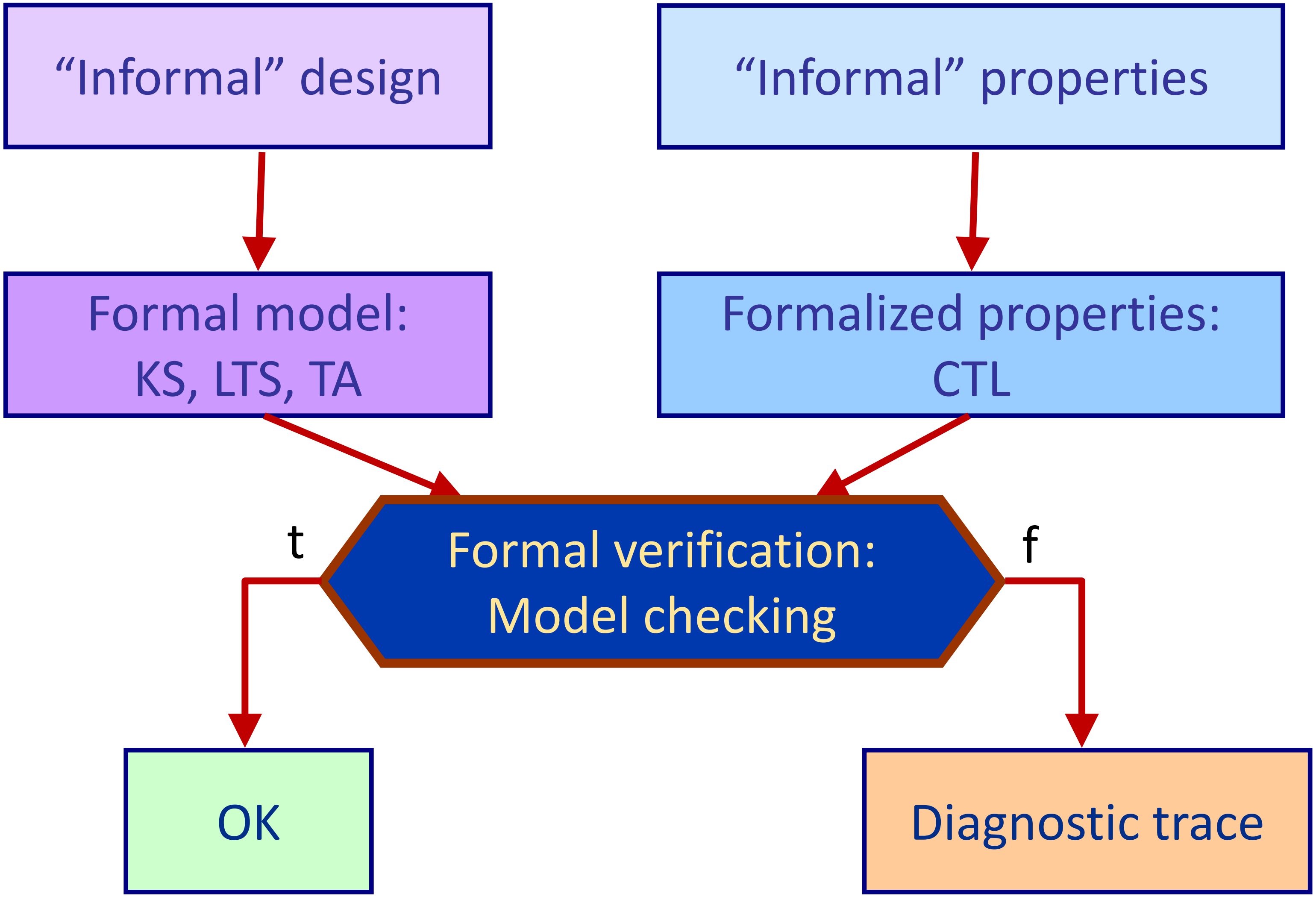
Model checking: Introductory examples

Istvan Majzik
majzik@mit.bme.hu

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

Formal verification: Goals



Example 1:
Mutual exclusion protocol

An engineering task

- Let us consider a concurrent (multi-process) system
- At most one process is allowed to access a shared resource at a time (mutual exclusion is required)
  - Example: Use of communication channel
  - Access to resource: “Critical sections” in the programs; at most one process is allowed to be in critical section
  - The platform (OS, framework) does not give support: no semaphore, no monitor, etc.
  - Only shared variables can be used (atomic reading/writing)
- How to do it?
  - Classical solutions (Peterson, Lamport, Fischer etc.)
  - Custom algorithm

Algorithm for mutual exclusion

- 2 processes, 3 shared variables (H. Hyman, 1966)
  - **blocked0**: process 1 (P0) wants to enter
  - **blocked1**: process 2 (P1) wants to enter
  - **turn**: which process is allowed to enter (0 for P0, 1 for P1)

P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
      while (blocked1==true) {
            skip;
      }
      turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

P1

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
      while (blocked0==true) {
            skip;
      }
      turn=1;
    }
    // Critical section
    blocked1 = false;
    // Do other things
}
```

Is the algorithm correct?

The model in UPPAAL (version 1)

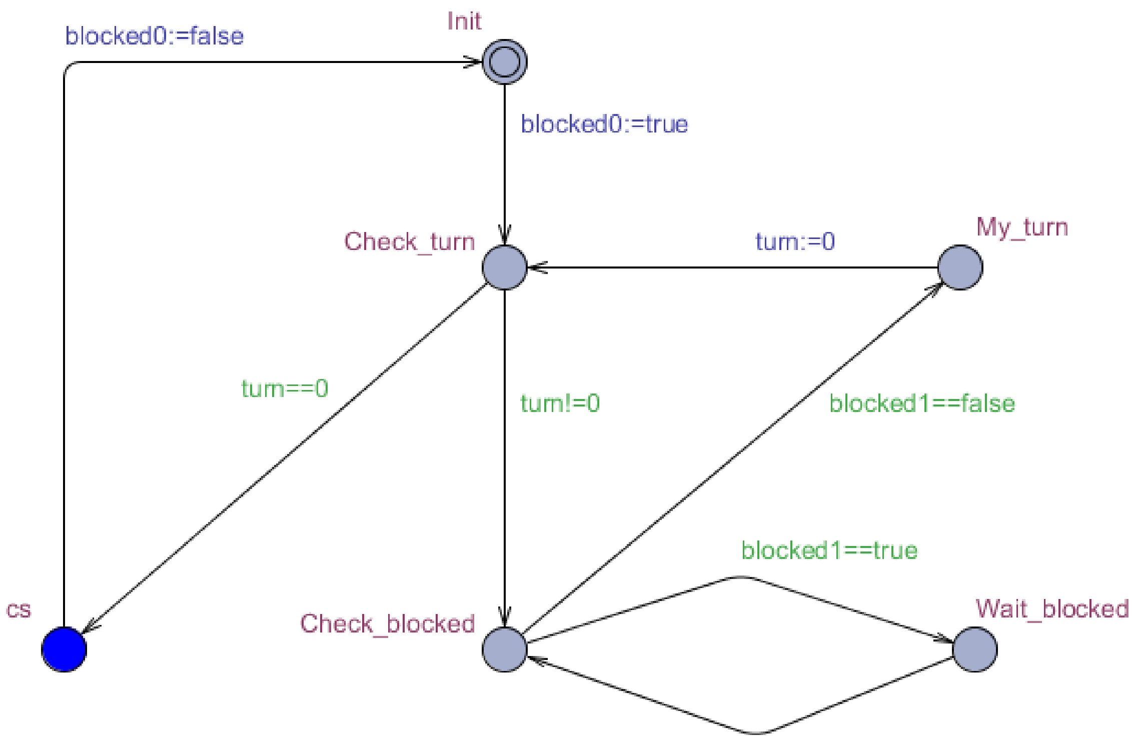
Declarations:

bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;

Used modeling artefacts:

- Global variables
- Variables with restricted domain

Automaton P0:



P0

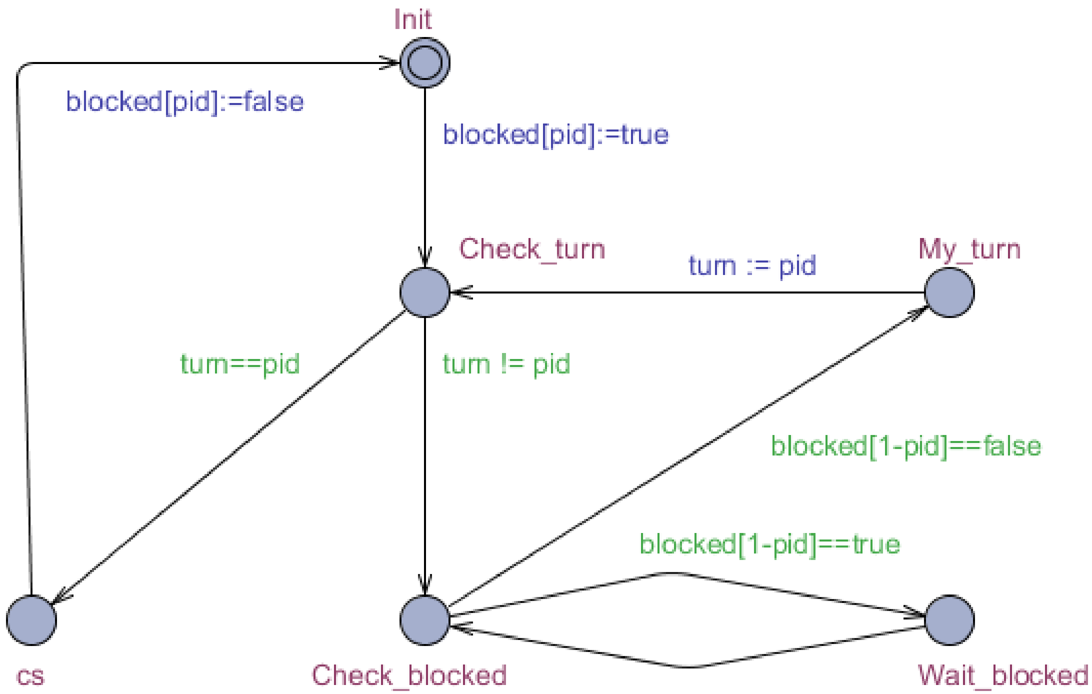
```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

The model in UPPAAL (version 2)

Declarations:

```
bool blocked[2];
int[0,1] turn;
P0 = P(0);
P1 = P(1);
system P0,P1;
```

Template P with parameter const int pid:



Used modeling artefacts:

- Global variables
- Variables with restricted domain
- Variables of array type
- Modeling common behavior with templates
- Template instantiation with parameters

P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Properties to verify in the example

- Mutual exclusion:
  - At most one process is allowed to be in the critical section
- The expected behavior is possible:
  - For P0 it is possible to enter the critical section
  - For P1 it is possible to enter the critical section
- Starvation freedom:
  - P0 will eventually enter the critical section
  - P1 will eventually enter the critical section
- Deadlock freedom:
  - It is not possible that processes are just waiting

How to do model checking in UPPAAL?

- Atomic propositions:
  - Values of variables can be referred: e.g., a!=1
    - Using integer arithmetic and bit operations
  - Control locations can be referred: e.g., Train.cross
    - For parameterized processes: forall, exists quantifiers
  - Deadlock (no action): Specific deadlock proposition
- Boolean operators:
  - and, or, imply, not, ? : (this latter is the “if-then-else”)
- Temporal operators: Restricted CTL
  - Notation: [] instead of G, and <> instead of F
    - This way we have CTL operators: A[], A<>, E[], E<>
    - [] is also interpreted on finite paths (till the last state)
  - Temporal operators cannot be nested
    - But there is a special operator: p-->q means A[] (p imply A<> q)

Configuring model checking in UPPAAL

- Set of properties can be provided
  - Model checking can be started one-by-one
- Diagnostic trace (counter-example or witness) can be generated
  - Some, shortest, or fastest
  - It is loaded into the simulator (for debugging)
- Search order in the state space:
  - Depth-first, random depth-first
  - Breadth-first
- State space representation:
  - Compact data structure
  - Under- / over-approximation
  - Hash table size can be specified

UPPAAL: Formalizing requirements

- Mutual exclusion:

  At most one process is allowed to be in the critical section

  A[] not (P0.cs and P1.cs)

  Labels for critical sections: P0.cs and P1.cs

- The expected behavior is possible:
  - For P0 it is possible to enter the critical section: E<>(P0.cs)
  - For P1 it is possible to enter the critical section: E<>(P1.cs)
- Starvation freedom:

  P0 will eventually enter the critical section: A<>(P0.cs)

  P1 will eventually enter the critical section: A<>(P1.cs)

- Deadlock freedom:

  It is not possible that processes are just waiting: A[] not deadlock

UPPAAL: Results of model checking

- Mutual exclusion is not ensured
  - Counterexample: specific interleaving between the processes (can be replayed in simulator)
- No deadlock
- The expected behavior is possible
- Starvation freedom cannot be checked without specification of timing
  - Trivial counterexample: Time elapses indefinitely in the initial location
    - Valid timed behavior in the model
    - Enforcing progress: urgent location, or location invariants
  - Starvation freedom?
    - The system is not starvation free (cyclic counterexample exists)

Fixing the algorithm: Mutual exclusion ensured

Hyman's algorithm

- For process P0
  (P1 analogously):

```
Hyman:

while (true) {
     blocked0 = true;
     while (turn!=0) {
       while (blocked1==true) {
               skip;
       }
       turn=0;
     }
     // Critical section
     blocked0 = false;
     // Do other things
}
```

Peterson's algorithm

- For process P0
  (P1 analogously):

```
Peterson:

while (true) {
     blocked0 = true;
     turn=1;
     while (blocked1==true &&
          turn!=0) {
               skip;
     }

     // Critical section
     blocked0 = false;
     // Do other things
}
```

Example 2:
Dice game

Dr. Tamás Bartha, BME

The problem

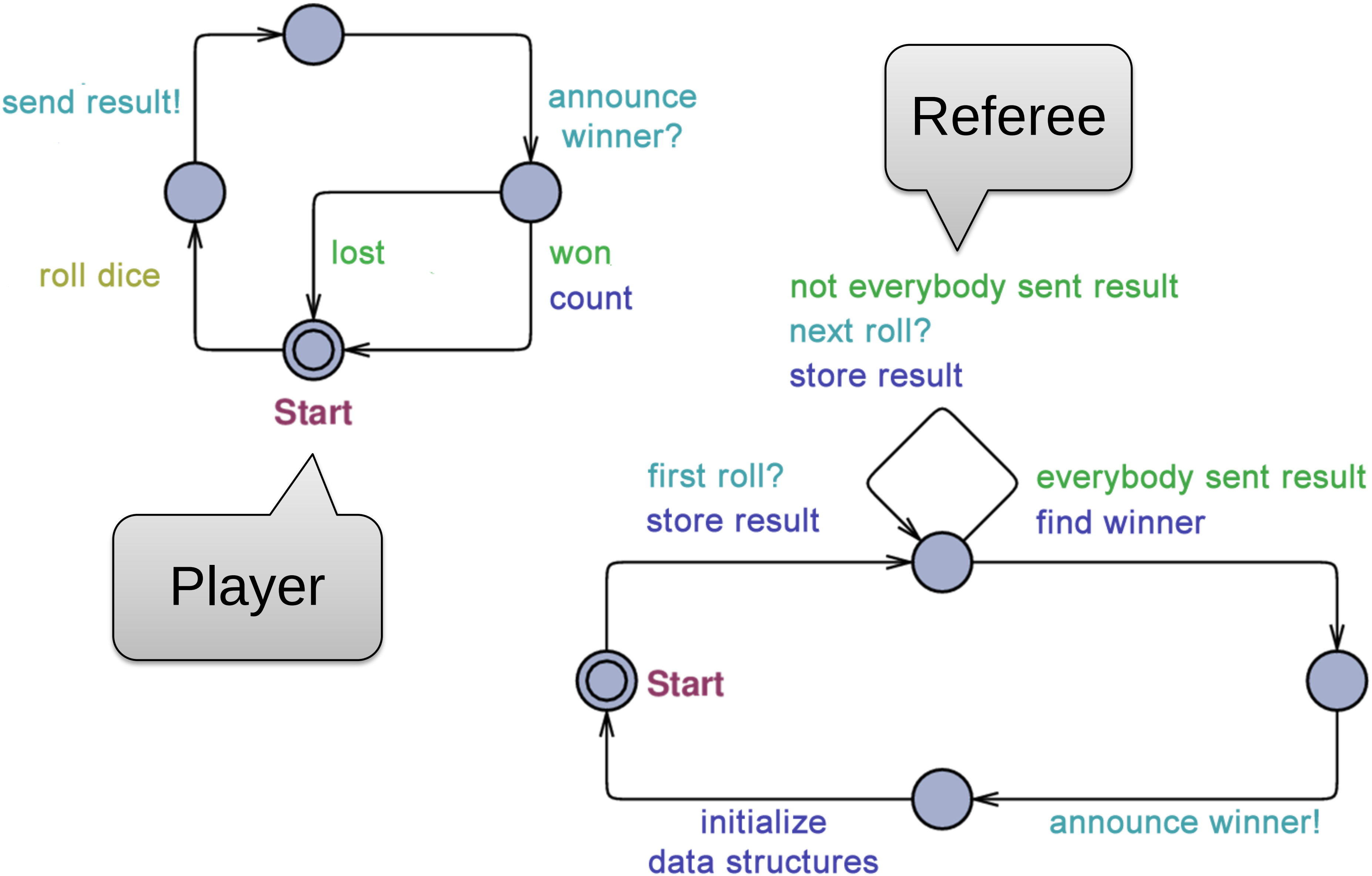
Game: Rolling a dice

- *n* players, 1 referee
- Each player rolls a dice once
- Then tells the result to the referee
- The referee
  - Collects all results
  - Finds the largest result(s)
  - Announces the winner(s)
- Players count the number of their winning rounds
- The winner of the game is who first won 10 rounds

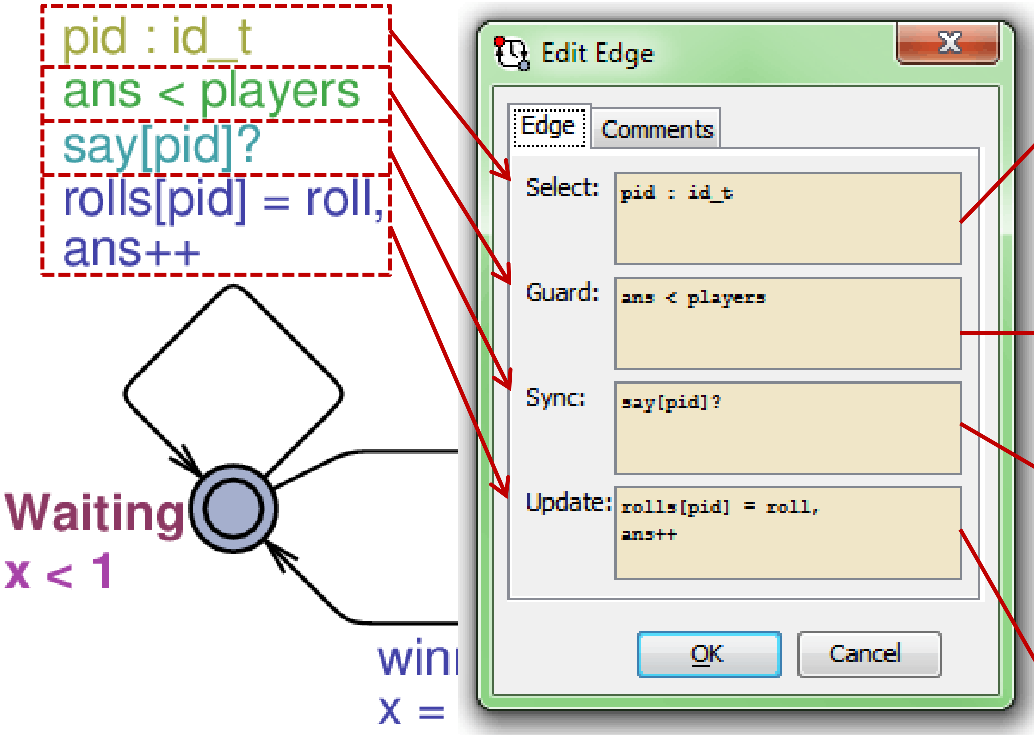
What do we have to solve:

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

Basic idea for the solution: Sketch of the models



Possibilities for modelling transitions in UPPAAL



- 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 » Guard » Sync » Update

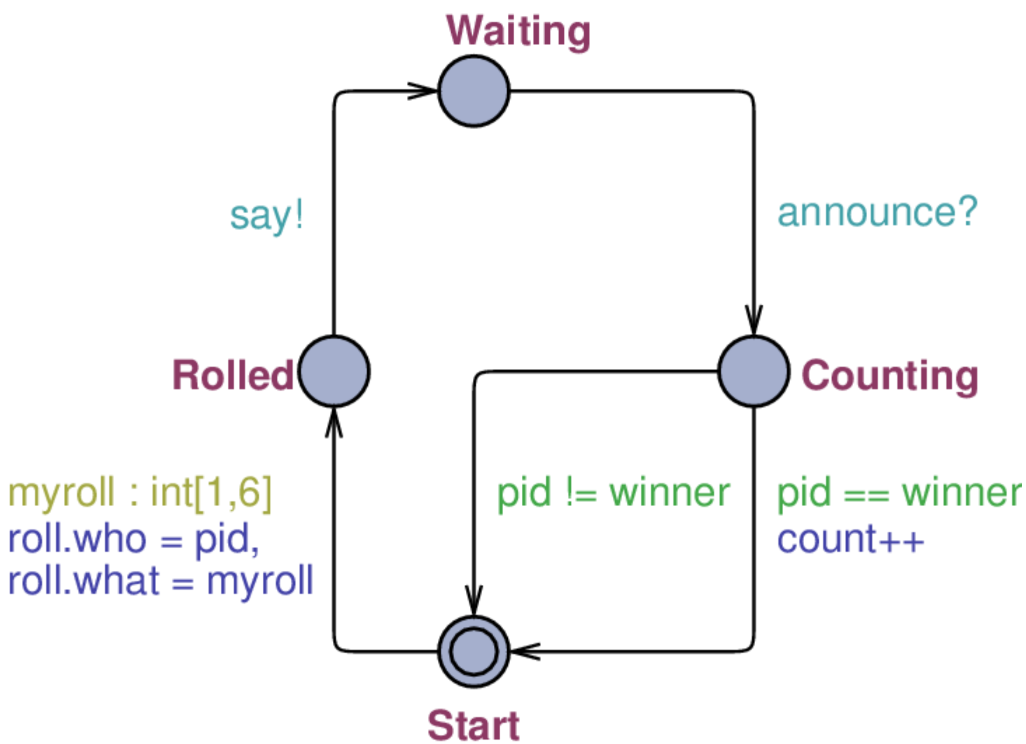
Solution: System and the player

System:

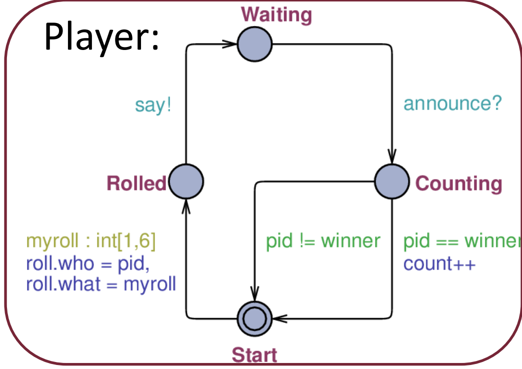
```
system Player, Referee;

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;
```

Player:

```
Player(id_t pid)

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



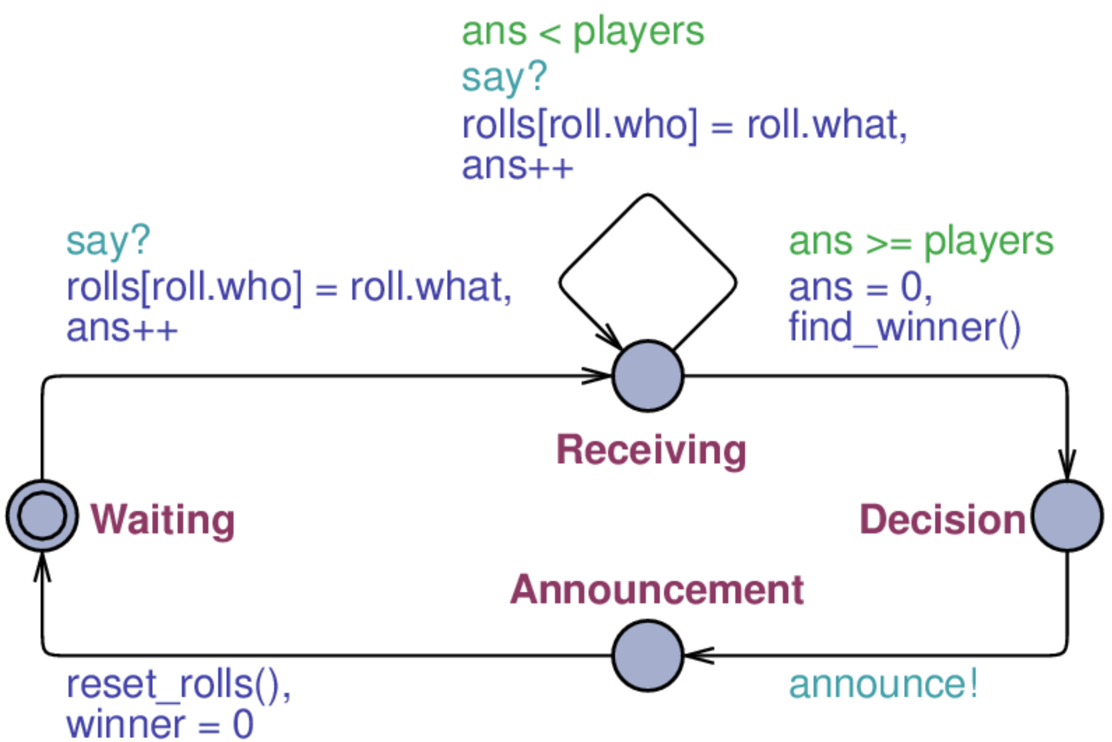
Solution: Referee



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;
}
```



Let’s check the behavior (dice_roll_1.0)

- On each path, there is a player who is the winner of the game
  - The count of the highest rolls reaches the value of wins

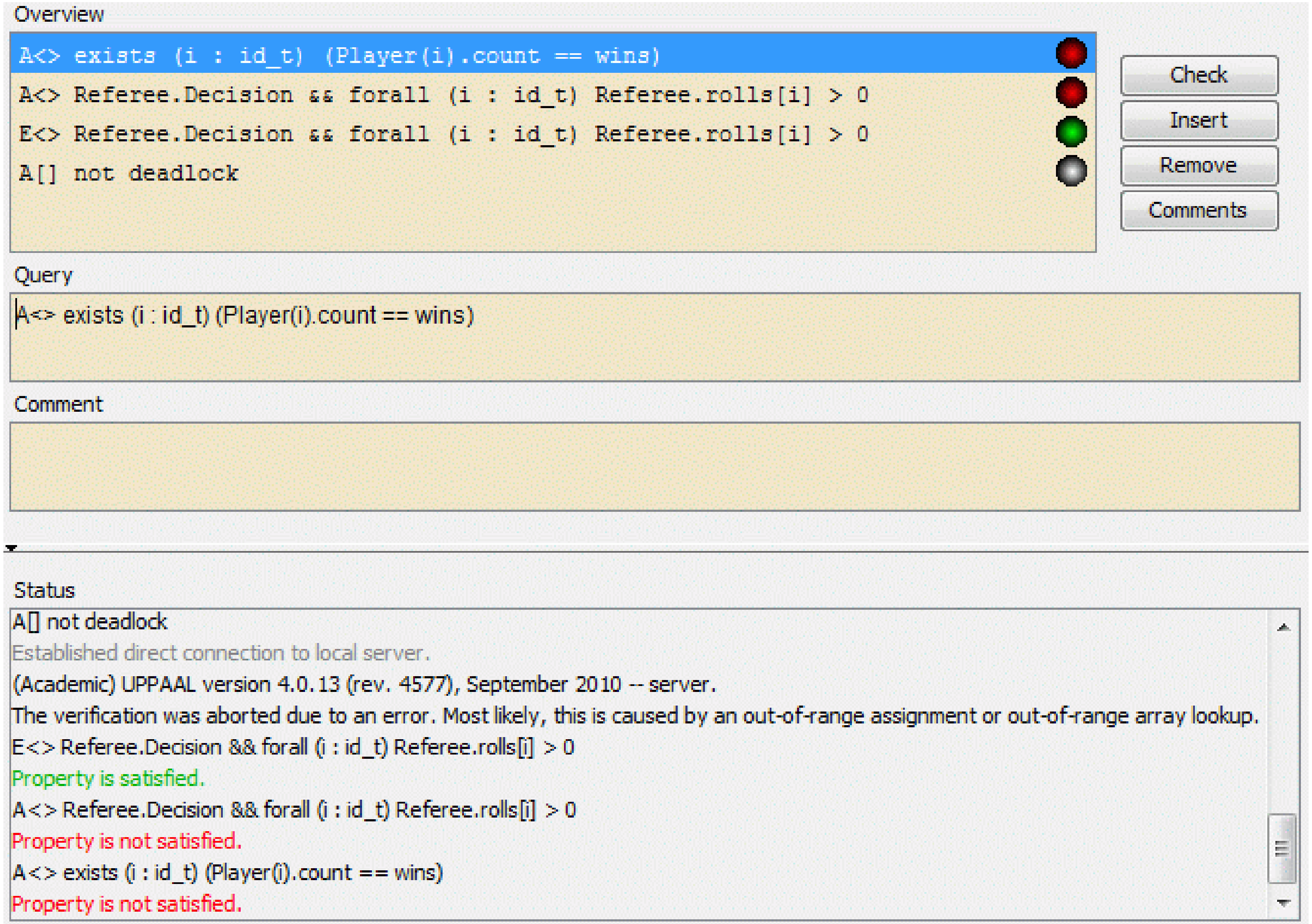
    A<> exists (i : id_t) (Player(i).count == wins)
- Referee decides if all players made their rolls
  - This happens at least once:

    E<> Referee.Decision && forall (i : id_t) (Referee.rolls[i] > 0)
  - This happens eventually on all paths:

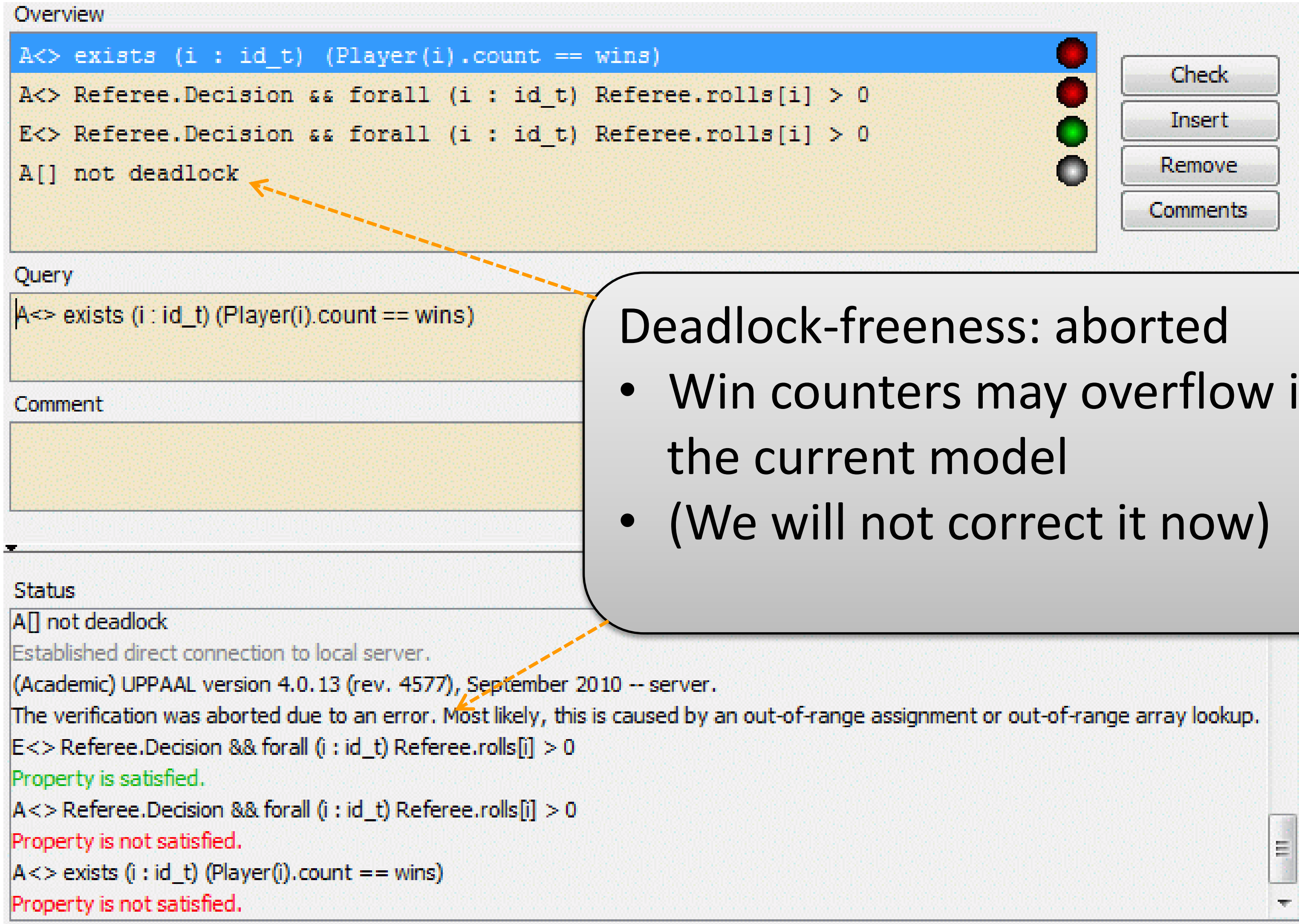
    A<> Referee.Decision && forall (i : id_t) (Referee.rolls[i] > 0)
- The system has no deadlock
  - There is no such state, which has no enabled transition to another state

    A[] not deadlock

Let’s check the behavior (dice_roll_1.0)



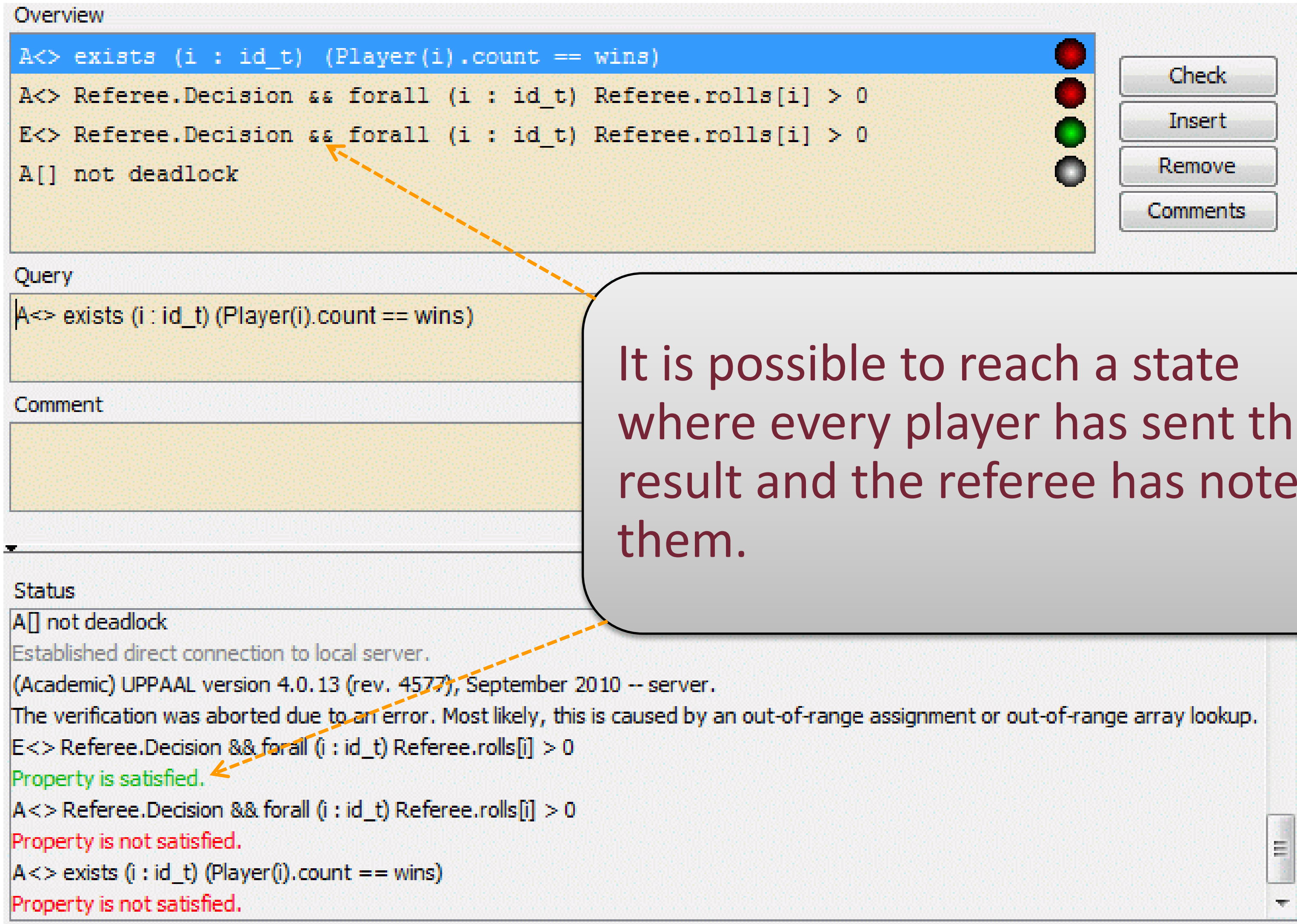
Let’s check the behavior (dice_roll_1.0)



Deadlock-freeness: aborted

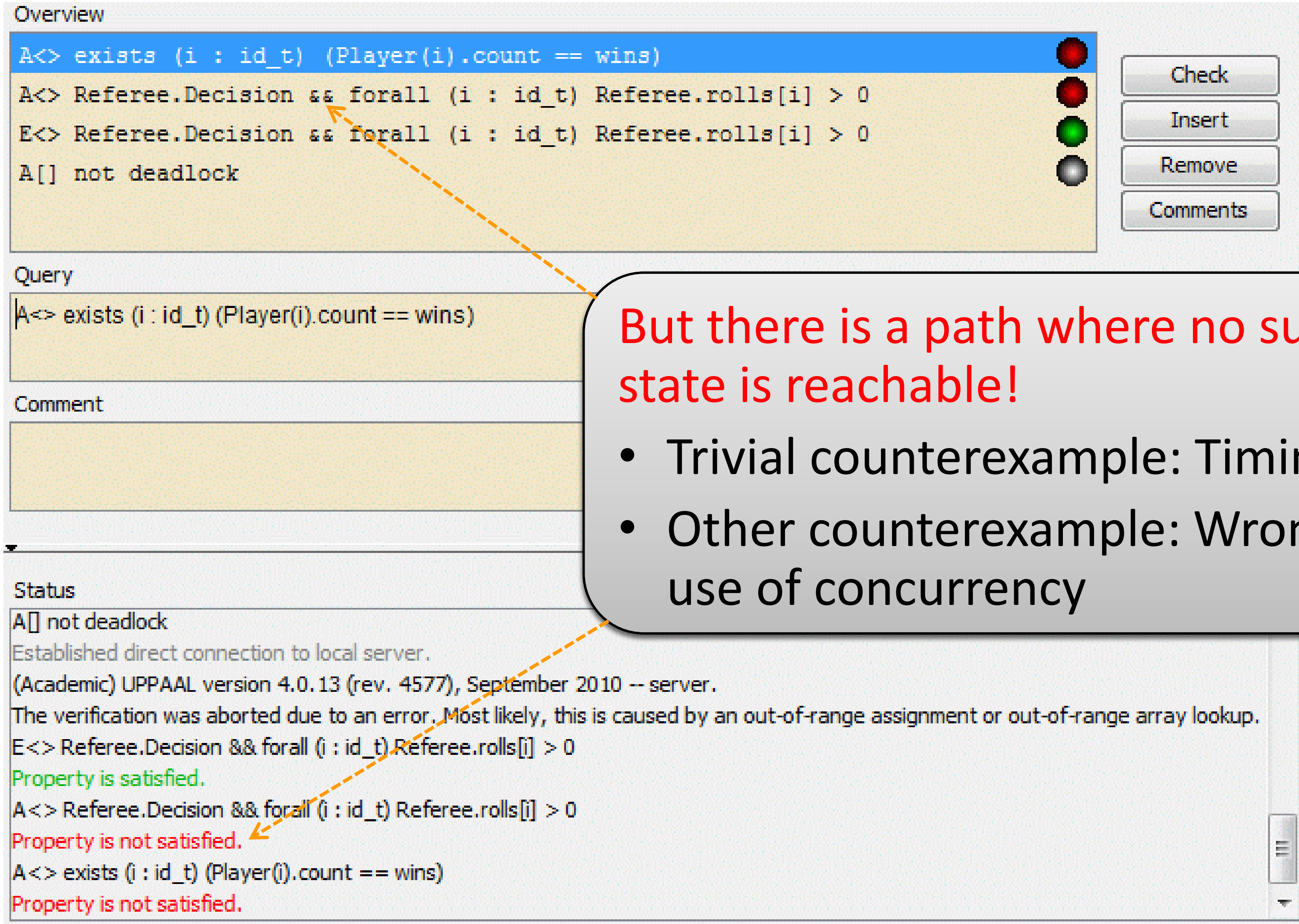
- Win counters may overflow in the current model
- (We will not correct it now)

Let’s check the behavior (dice_roll_1.0)



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 (dice_roll_1.0)

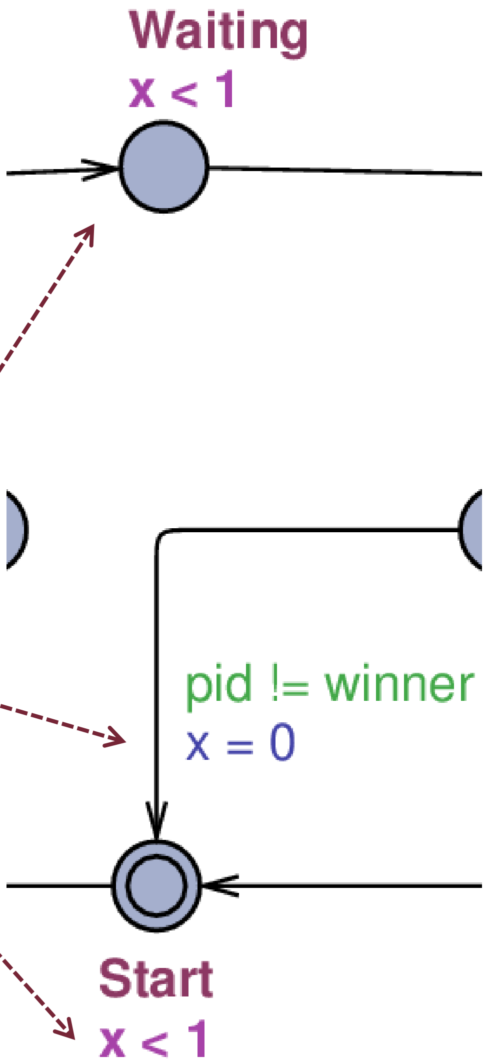


But there is a path where no such state is reachable!

- Trivial counterexample: Timing
- Other counterexample: Wrong use of concurrency

Avoiding trivial counterexample by state invariants

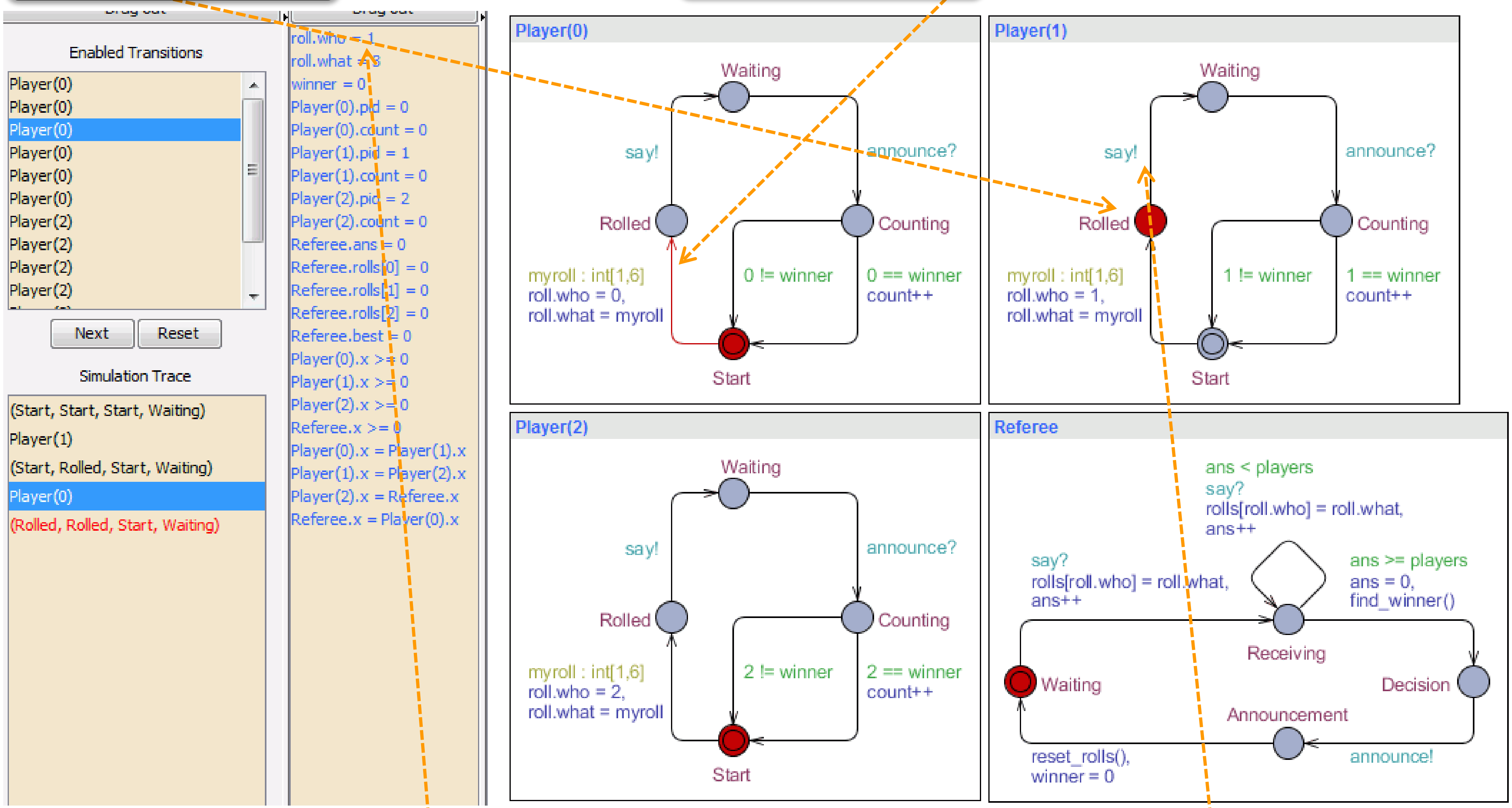
- If we examine all possible paths (e.g. A<>) then UPPAAL also checks the possibility of not leaving a state (if it is a valid behavior)
- Solution: State (location) invariant
  - Add a clock variable
  - Initialize when entering the state
  - Not leaving a state is valid until the state invariant holds
    (here in the example: for at most 1 time units)



Wrong concurrency – why?

Player(1) rolled

Player(0) rolls

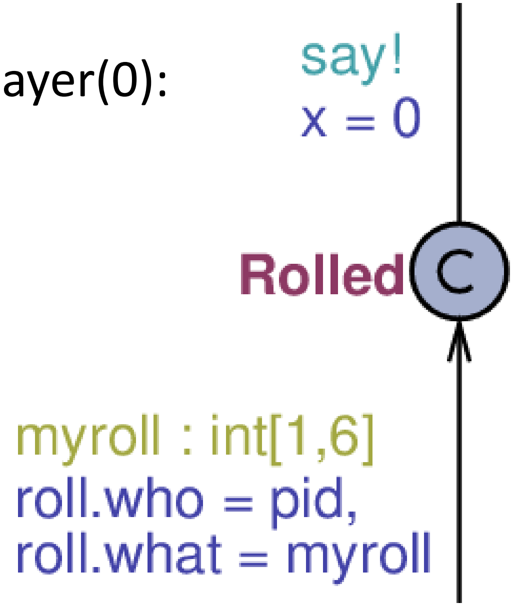


Player(0) will overwrite the shared variable

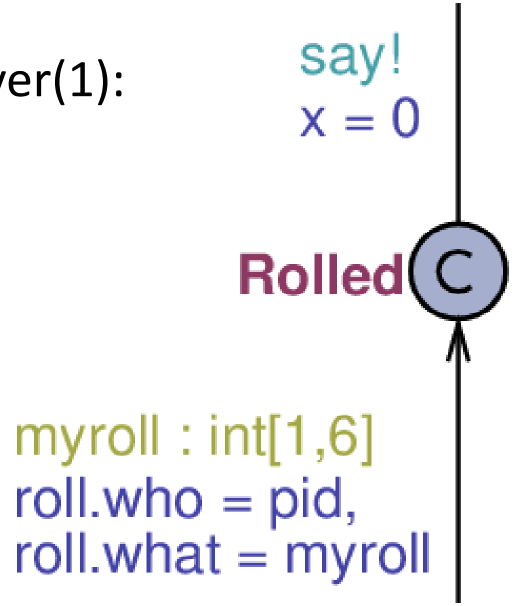
Player(1) will “send” wrong one

Avoiding wrong concurrency (dice_roll_1.1)

Player(0):



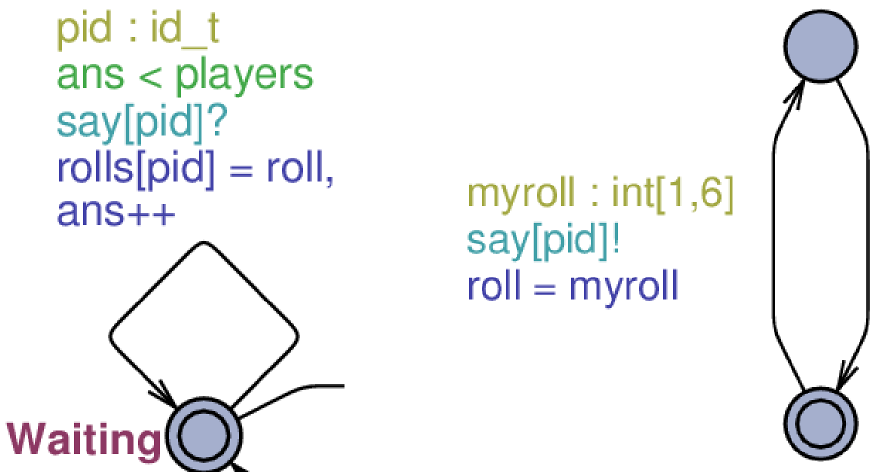
Player(1):



- Problem: Concurrent activities of the players on shared variable
  - Registering the results: writing to the roll shared variable
  - Communication with the referee: using roll with the say! transition
- Potential solution:
  - Implementing atomic “update and send” operations by introducing a “committed” state (it must be left instantly)

Special constructs that can be used (dice_roll_2.0)

- Monitoring an array of channels
  - The receiving process checks all channels "at once" using a Select construct
  - Synchronization is performed on the channel that is ready
    - Channel id can be used in the Update section
  - Model checker will examine all potential synchronizations



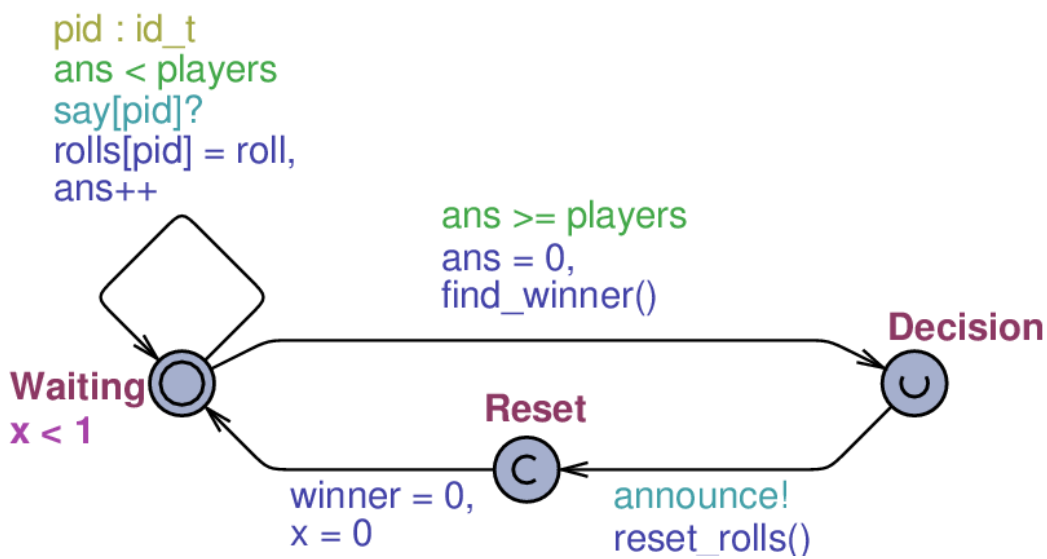
- Using iterators in functions

```
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;
}
```

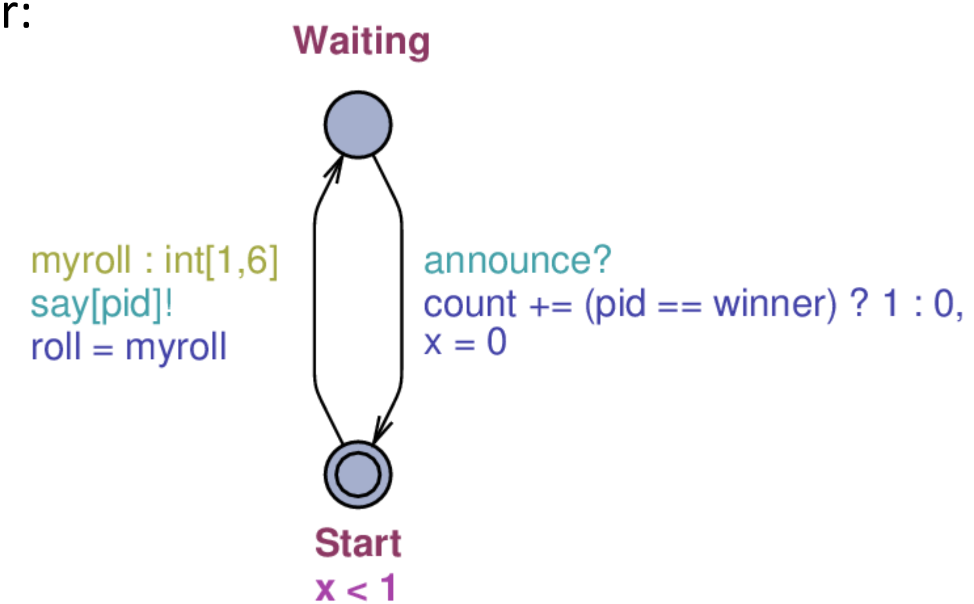
“Compact” model

- Using arrays of channels
- Applying operator “? :”
- Collecting results in a single state
- Using iterators
- Reset state can be omitted

Referee:



Player:



Other modeling advices and practices

- Order of evaluating arc expressions:
  Select » Guard » Sync » Update
  - On a synchronized arc, Update of the sender is evaluated before the Update of the receiver
  - Cannot test (in a guard) a global variable that was set by synchronized arc
- Using functions: Debugging is difficult
  - Statement by statement simulation is not possible
- When verifying properties such as A<> q, clock variables must be used to avoid the trivial counterexample (not leaving a state)
  - Note: A<> is also included in “leads to”: p --› q means A[] (p imply A<> q)
  - Do not forget to reset clock variables when necessary
- The model checker of UPPAAL cannot check deadlocks when using channel or automata level priorities (these should be avoided)