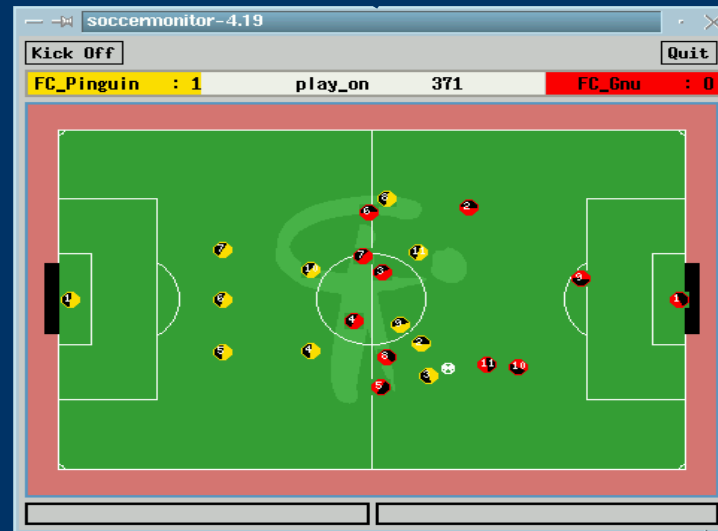


Debugging and verification of multi-agent systems



Clara Benac Earle
Lars-Åke Fredlund
Grupo Babel, FI, UPM

Partly
sponsored by
the DESTINO
project (INES)

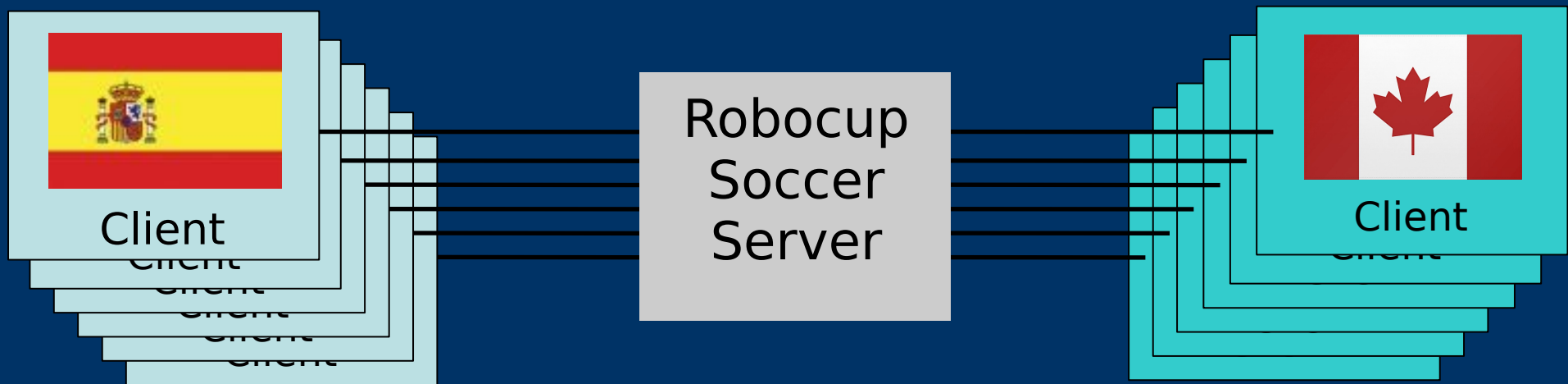
RoboCup

An initiative to promote research in robotics and multi-agent systems.

GOAL: By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world champions

RoboCup Simulation League

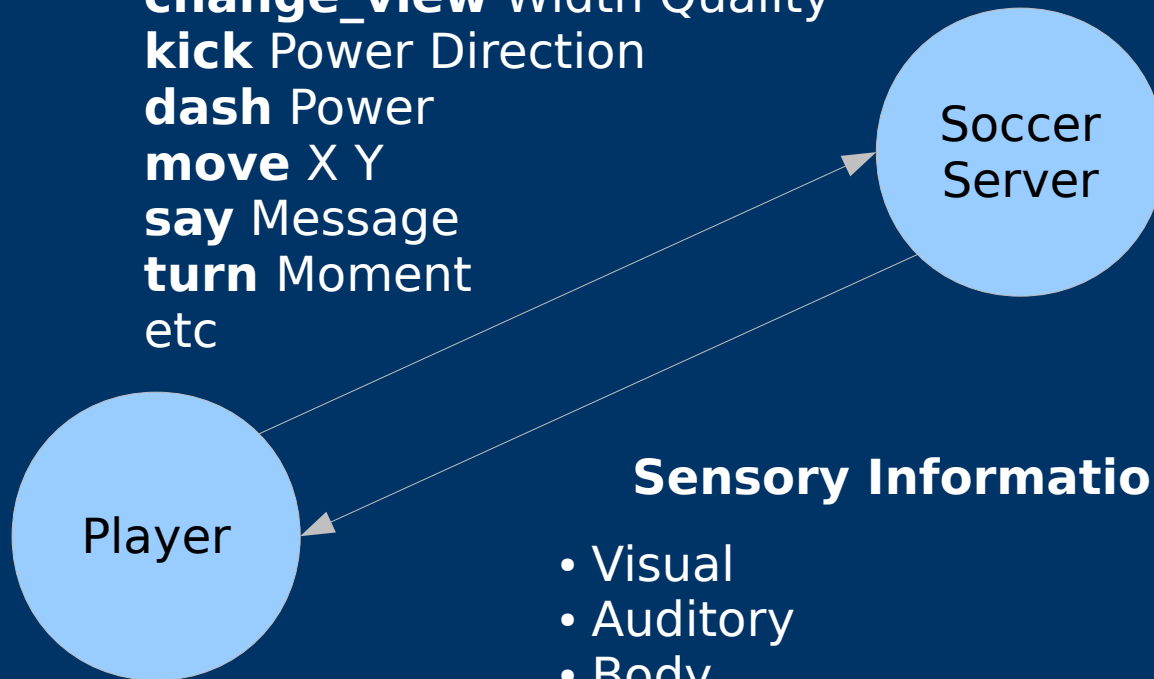
- Two teams of 11 simulated autonomous robotic players (agents) play a simulated football match in client-server style.
- Each client controls movements of one player. Communication between the server and each client is done via UDP/IP sockets.



RoboCup Communication

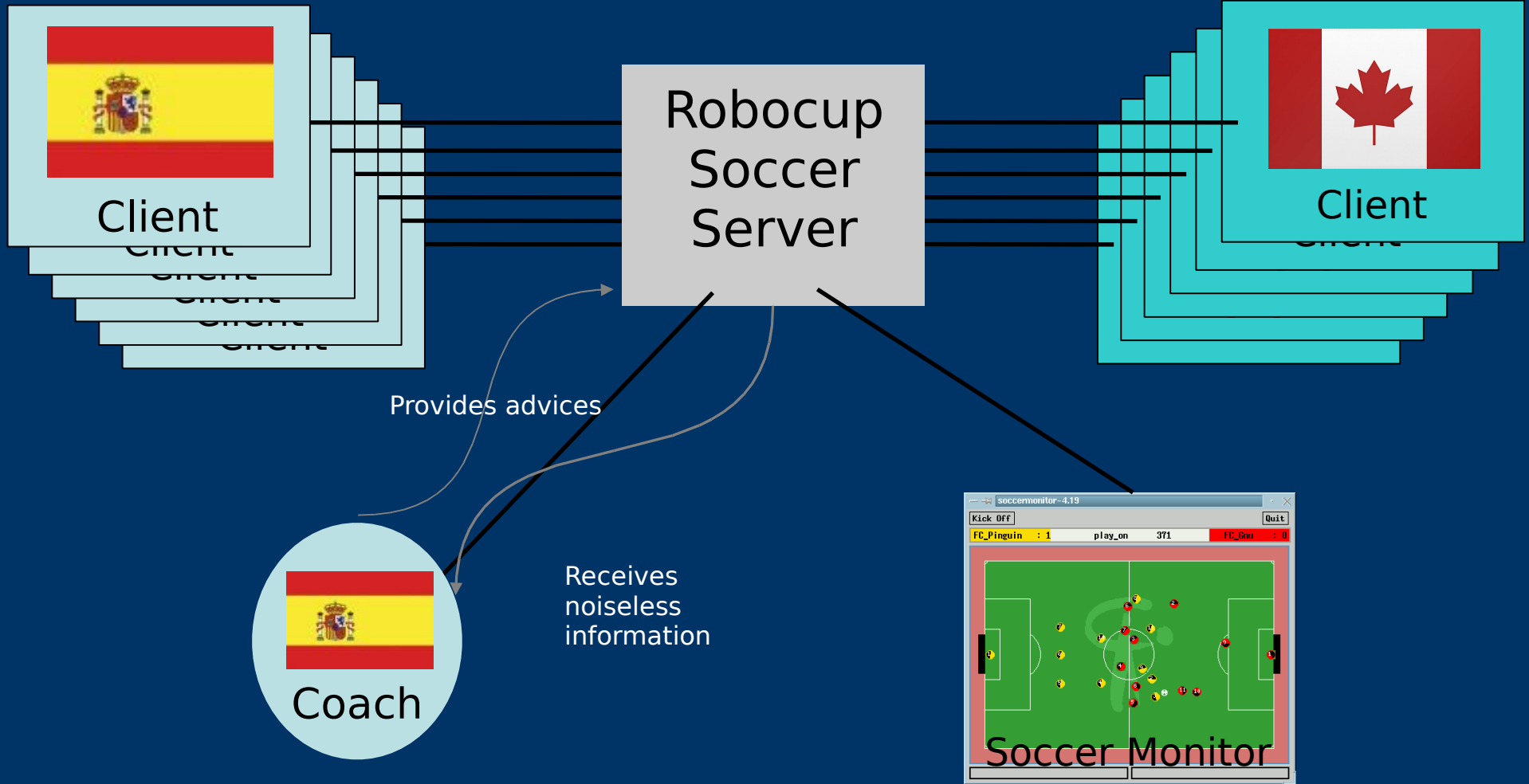
COMMANDS

catch Direction
change_view Width Quality
kick Power Direction
dash Power
move X Y
say Message
turn Moment
etc



Sensory Information

- Visual
- Auditory
- Body



The Challenge

We were given a set of RoboCup teams programmed in Erlang...

We wanted to verify that these teams play well or, if not, find what the problems were



Checking RoboCup Agents

To apply model checking techniques to such a verification problem we should construct a simplified model of:

the soccer server

+ our team

+ the environment (the opponent team)



Huge state space

Checking RoboCup Agents

Use the McErlang model checker for executing the agents, and to formulate correctness properties to check as monitors, but ...

...instead of model checking a team we used the model checker as a testing/simulation environment.



Erlang



www.erlang.org

- Functional programming language
- Concurrency oriented
 - Lightweight processes
 - No shared memory
 - Communication via asynchronous message passing
- Excellent support for developing distributed fault-tolerant real-time applications communicating via message passing
- Good libraries for developing large industrial applications
 - supervisor component (for fault-tolerance applications)
 - generic server component (implementing a client-server component)
 - module for programming finite-state machines, etc.

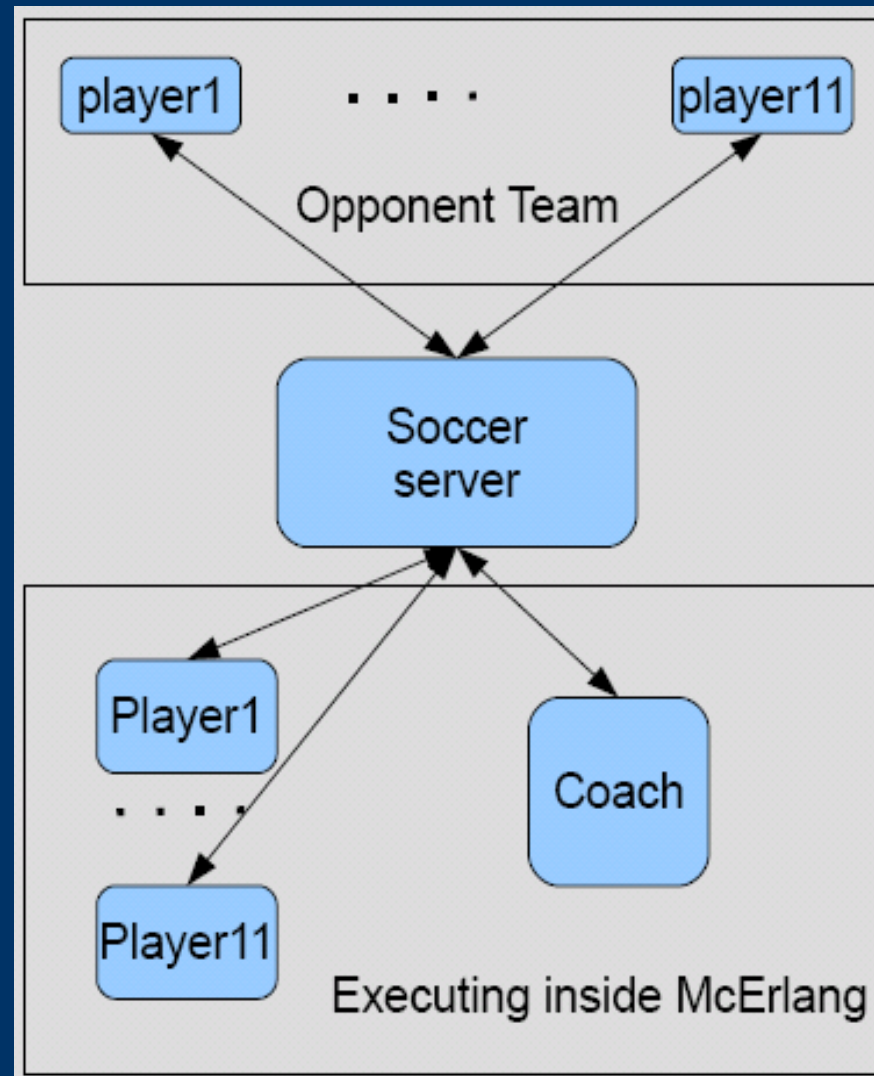
McErlang model checking approach

- *The lazy solution:* Execute Erlang functions in Erlang
- *The catch:* We need access to the combined resulting system state
- *Ideally:* Extract system state (processes, queues, function contexts) from the Erlang Runtime System. However...we have plenty of excuses not to mess with the Runtime System (written in C, distributed, complex, level of detail, etc)
- Instead we have developed a new runtime system for the process part of Erlang (this way we have full control of system execution: we can break communication links, kill processes, etc)

McErlang model checking approach

- Correctness properties as automata
 - The program to check is run in lock step with the automaton
 - An automaton state inspects the corresponding program state to see if it is a bad state
-
-

Checking RoboCup Agents



Correctness Property Classification

We can separate desirable properties of RoboCup teams:

Observable properties: evaluated by observing only the actions of an agent (no internal state).

Players stay inside playing field.

Goalie does not leave the goal area.

Discrepancy properties: difference between an agents' beliefs and the objective reality must not exceed some safety margin

Internal properties: the general consistency of an agent, and the efficacy of its internal logic decisions.

The agent never attempts a pass when he "thinks" that an opponent player may intercept the ball

Observable Property (I)

```
stateChange(State, MonState, Actions) ->
  try
    {ok, CoachState} = coach:getCoachState(State),
    AllPlayers = coach:getOwnPlayers(CoachState),

    %% Verify that all players are in the allowed area
    {LowerX, LowerY, UpperX, UpperY} = MonState,
    case lists:any
      (fun (P) ->
        {PosX, PosY} = P#player.position,
        (PosX < LowerX) orelse (PosX > UpperX) orelse
        (PosY < LowerY) orelse (PosY > UpperY)
      end, AllPlayers) of
      true -> error
      false -> {ok, MonState}
    end
  catch _ -> {ok, MonState} end.
```

Observable Property (II)

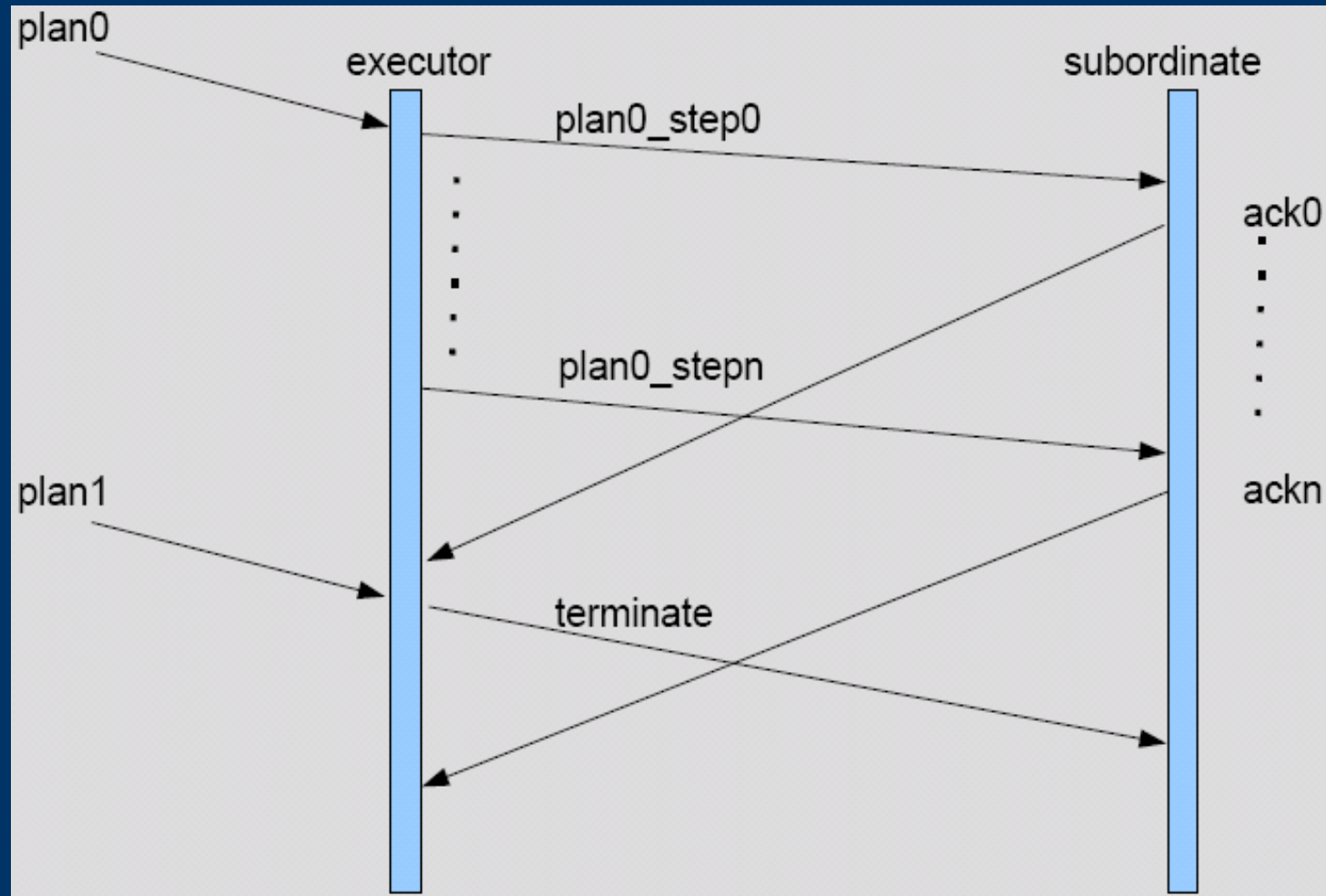
Executing the RoboCup Team under McErlang → Error?

Reason...

The executor process has two states

- Idle: waits for a new plan
- Execute: executes a received plan. A subordinate process executes each step of the plan

Observable Property: a bug



A Discrepancy Property

- We observed that players played badly

Why? Maybe their beliefs do not match reality...

- We formulate a property regarding the agent's beliefs of the position of the ball
- We found that with an error of 0.5m McErlang discovered a situation where a player thought the distance to be around 7.6m for over a second while the real distance hovered around 17m.



A Logic Property (I)

Another possible explanation for bad playing: the decision making of players is bad.

Example: a player never attempts a pass when it suspects that an opponent player may intercept the ball.



A Logic Property (II)

- We found a situation when, using as the unsafe distance 0.5 meters, an agent could attempt a pass in the following situation:

source player at $\{-23.9, -24.27\}$,
destination player at $\{-13.57, -8.4\}$, and
a known opposition player at $\{-17.56, -14.59\}$
which could quite easily intercept the pass.

Combining simulation and model checking

- Simulate:
 - Until an interesting state is found
the ball is in the goal area and the ball is close to an opponent player
 - Randomly
 - Then model check
 - Select a subset of agents to continue executing
let the player closest to the ball run
-
-

Conclusions

- We have been able to find bugs in RoboCup teams using the McErlang tool in simulation mode.
- We formalized several types of properties: observable, discrepancy and internal and checked them.
- To formalize internal logic properties we need to examine the internal state of an agent – McErlang allows us to reuse functions from the agent source code
- We have done some experiments combining simulation and model-checking and it seems promising.