

# Verifying Robocup Teams <sup>★</sup>

Draft

Clara Benac Earle<sup>2</sup>, Lars-Åke Fredlund<sup>1</sup>, José Antonio Iglesias<sup>2</sup> and Agapito Ledezma<sup>2</sup>

<sup>1</sup> LSIS, Facultad de Informática, Universidad Politécnica de Madrid  
fred@babel.ls.fi.upm.es

<sup>2</sup> grupo CAOS, Universidad Carlos III de Madrid  
{cbenac,jiglesia,ledezma}@inf.uc3m.es

**Abstract.** Verification of multi-agent systems is a challenging task due to their dynamic nature, and the complex interactions between agents. An example of such a system is the RoboCup Soccer Simulator, where two teams of eleven independent agents play a game of soccer against each other. In the present article we attempt to verify a number of properties of RoboCup soccer teams, using a methodology involving testing. To accomplish such testing in an efficient manner we use the McErlang model checker, as it affords precise control of the scheduling of the agents, and provides convenient access to the internal states and actions of the agents of the soccer teams.

## 1 Introduction

The analysis and verification of multi-agent systems is not an easy task due to their dynamic nature, and the complex interactions between agents. One method that is often advocated to verify such systems is model-checking. However, in performing model-checking on multi-agent systems two main issues arise: i) a model needs to be constructed, and ii) the state space is bound to grow too large. In this paper we propose an alternative approach to the verification of properties in multi-agent systems by means of testing, in particular we use a model checker to simulate RoboCup team and verify properties during such simulation runs. The tool we use, McErlang [10], permits precise control of concurrency and communication, and detailed access to the internal states of agents and communication channels.

The RoboCup Soccer Simulator, the soccer server [5], is a research and educational tool for multi-agent systems and artificial intelligence. It enables two teams of eleven simulated autonomous players to play soccer (football). A match is carried out in client/server style: the server provides a virtual field and simulates all movements of a ball and the players, and each client controls the movements

---

<sup>★</sup> This work has been partially supported by the FP7-ICT-2007-1 Objective 1.2. IST number 215868

of one player. Communication is done via UDP/IP sockets, enabling players to be written in any programming language that supports UDP communication.

Erlang [1] is a programming language developed at Ericsson for implementing telecommunication systems. The principal characteristics of Erlang, i.e., a clear separation between data and processes and a high level of abstraction thanks to its functional style, together with excellent support for developing distributed applications, makes writing code for RoboCup teams in Erlang an easy task. Indeed, undergraduate students at the IT university of Gothenburg have been developing such teams to compete in their local RoboCup simulation tournament. Given the rather complex nature of the application, and the availability of capable verification tools for Erlang such as e.g. McErlang[10] (a model checker for Erlang code), it seemed natural to try to use these verification tools in the task of analyzing some interesting properties of multi-agent RoboCup teams.

The use of tool support in the task of verifying multi-agent systems is recently attracting significant interest from the agent community. In [3], for example, a variant of the abstract agent-oriented programming language AgentSpeak, AgentSpeak(F), is proposed. By translating AgentSpeak(F) programs into Promela or Java, properties written in LTL can be model-checked with SPIN or the Java Path Finder [15], a general purpose model checker for Java. A difference between their approach and ours is that AgentSpeak is based on the BDI agent architecture while we do not consider any specific agent architecture. In [11] a combination of UML statecharts and hybrid automata was proposed for modeling multi-agent systems, and the method was applied to the task of model checking agents of the RoboCup rescue simulation league. In [4] a trace based approach is used to study a complex agent scenario.

This paper is organized as follows. In the next section we introduce the Erlang programming language, and in Sect. 3 a description of the McErlang tool is given. In Sect. 4 the implementation of a RoboCup soccer team in Erlang is explained. Our approach to checking properties on the soccer players is explained in Sect. 5, together with a discussion of the type of experiments we have carried out. We conclude in Sect. 6 with a discussion on the present results, and directions for future work.

## 2 The Erlang Programming Language

Erlang [1] is a programming language developed at Ericsson for implementing telecommunication systems. It provides a concurrent functional language, with constructs for process creation and communication via message passing. Erlang has support for writing distributed programs; *processes* can be distributed over physically separated processing *nodes*.

The software of commercial products written in Erlang is typically organized into many, relatively small, source modules, which at run-time execute as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. In practice, Erlang programmers predominantly use the OTP component library [13], which offers a number of

useful software components such as: a generic server component for client-server communication, a finite-state machine component, and a supervisor component that restarts failed processes. Our approach to model checking Erlang programs can verify software that is built using both the core message passing language and with these high level components.

The Erlang/OTP programming environment is a comparatively rich programming environment for programming systems composed of (possibly) distributed processes that communicate by message passing. Fault tolerance is implemented by means of failure detectors, a standard mechanism in the distributed algorithms community. Moreover there is a process fairness notion, something which often makes it unnecessary to explicitly specify fairness in correctness properties. The language provides explicit control of distribution, and a clean model of distribution semantics. For distributed processes (processes executing on separate nodes) the communication guarantees are far weaker than for processes co-existing on the same processor node.

### 3 The McErlang tool

McErlang [10] is a model checker which has been used to verify a number of distributed Erlang applications [10, 8, 9].

The internal construction of the model checker is parametric, enabling a user to easily change its configuration for different verification runs.

The input to the model checker is the name of an Erlang function which starts the execution of the program to verify, together with a special call-back module also written in Erlang which specifies the behavioral property to be checked (called the *monitor*), implementing an automaton, as we will see later. The output of a verification can be either a positive answer saying that the property holds, or a negative one together with a counterexample. The tool can also be used to generate the state graph of a program; this permits the use of other model checking tools like CADP [6] for the actual property checking.

Moreover, a tool user can also specify:

- the name of a *language* module providing an operational semantics<sup>3</sup>,
- the particular *verification algorithm* to use, (e.g., a safety property checker, a liveness property checker or simulation of the program in conjunction with a correctness property),
- the name of a *state table* implementation, that records encountered program states (typically a hash table),
- the name of an *abstraction module* that abstracts program states, and
- the name of a *stack module* that implements the stack of program states (storing all or some of the states occurring on the path from the first program state to the current one)

---

<sup>3</sup> Apart from Erlang, we have also for instance implemented a semantics for the web service specification language WS-CDL, thus providing a WS-CDL model checker[7]

### 3.1 Programming Language Semantics for Erlang

The language module should provide two functions implementing an operational semantics for the language: (i) `transitions` which given a state returns a list of all next actions executable by the program, and (ii) the function `commit` which given an action returns a concrete program state. The `transitions` function may not cause side effects outside the model checker environment (e.g., really writing out a file to the file system) whereas `commit` may (if used by the simulation algorithm).

The main idea behind McErlang is to re-use as much of a normal Erlang programming language implementation as possible, but adding a model checking capability. To achieve this, the tool replaces the part of the Erlang runtime system which implements concurrency and message passing, while still using the runtime system for the evaluation of the sequential part of the input programs.

The model checker has a complex internal state in which the current state of the runtime system is represented. The structure that is maintained by the model checker records the state of all alive processes (their process identifiers, mailboxes, computation state, etc). Moreover the global state kept by the model checker runtime system includes a structure to record process links, information about registered process identifiers, etc.

As Erlang lacks a good reflection capability, a program has to undergo a source-to-source translation before model checking; McErlang does this automatically. Essentially calls to functions with side effects in the module `erlang` (providing the standard API for process operations in Erlang) are substituted with calls to functions in the model checker provided module `evOS` instead (which are side-effect free, except that they modify the internal model checker data structures). In addition process sends and receives are transformed as well.

McErlang has built-in support for some Erlang/OTP component behaviours that are used in almost all serious Erlang programs such as the supervisor component (for fault-tolerant applications) and the generic server component (implementing a client-server component), and a module for programming finite-state machines. The presence of such high-level components in the model checker significantly reduces the gap between original program and the verifiable model, compared to other model checkers.

### 3.2 Correctness Properties

The model checker implements full linear-temporal logic (LTL) checking. Correctness properties are represented as Büchi automata (*monitors* coded in Erlang) which are checked using a standard on-the-fly depth-first model checking algorithm [12]. For efficiency, there is a dedicated safety property only checker available. A *monitor* checks whether the correctness property holds for the combination of the new program state and the monitor state. If successful, the monitor returns an updated monitor state (for safety checking). A *Büchi monitor* (automaton) is a monitor that additionally may mark certain states as accepting

states. As is well known [14], linear temporal logic formulas can be automatically translated to Büchi automata. Correctness properties can be implemented, therefore, as finite state machines where depending on the monitor state, actions leading to new states are accepted or not. Such correctness properties have full access to the internal state of the program run (including message queues, state of processes, and so on).

The memory aspect of monitors is implemented by sending along the old monitor state as an argument to the Erlang function implementing the monitor. Concretely a monitor defines two callback functions: `init(parameters)` and `stateChange(programState,monitorState,RunStack)`. The `init` function returns `{ok,monState}` where `monState` is the initial state of the monitor.

The `stateChange` function is called when the model checker encounters a new program state `programState`, and the current monitor state is `monitorState`, and the execution history (a subset of the program states, and actions, between the initial program state and the current one) is provided by the `RunStack` parameter. If a safety monitor finds that the combination of program and current monitor state is acceptable, it should return a tuple `{ok, newMonState}` containing the new monitor state. If future states along this branch are uninteresting the monitor can return `skip` (e.g., to implement a search path depth limit), any other value signals a violation of the correctness property implemented by the monitor. A Büchi automaton should return a set of states, each state either accepting `{accepting, state}` or not `{nonaccepting, state}`.

As an example, the code fragment in Fig. 1 implements a simple safety monitor that guards against program deadlocks: (a process is considered deadlocked if its execution state as recorded by the process data structure in the run-time system is `blocked`).

```
stateChange(State,MonState,RunStack) ->
  case lists:any(fun (P) -> P#process.status /= blocked end,
                State#state.processes) of
  true -> {ok, MonState};
  false -> {deadlock, MonState}
  end.
```

**Fig. 1.** Simple safety monitor

The syntax `variable#recordName.field` is used to access the field `field` of the record variable `variable`, of type `recordName`.

The Erlang language standard requires that process schedulers must be fair. The McErlang tool accordingly implements (weak) process fairness directly in its (liveness) model checking algorithm by omitting non-fair loops (i.e., ones that constantly bypass some enabled process) from the accepting runs.

### 3.3 Abstractions and State Tables

An abstraction abstracts a concrete program state into an abstract representation. It can be used to drastically reduce the checked state space of a program. The idea is inspired by the use of abstractions in [2]. A typical abstraction used in model checking is to compute a hash value from the state, and to use the hash value as the abstract state when checking for membership in the state table. However, program specific abstraction functions can also be implemented. For example, an abstraction could transform an integer variable into a boolean value, signaling whether the integer is less than zero. Clearly, there is in general no guarantee that such an abstraction is safe, i.e., that it does not cause a program failure to escape undetected (false positive).

As a second example we have implemented the usual abstraction of collapsing a whole state to a single integer (through hashing), and using a bit array table module to implement the state table. Thus, in a modular fashion, we have obtained an implementation of Holzmann’s bit-state hashing verification algorithm [12]. An implementation of a hashing abstraction thus becomes as simple as Fig. 2, where `erlang:phash2` is a built-in function which computes a hash value between `0..Size` for its term argument. Note that is an unsafe abstraction, although as proven in practise in many verifications, also a highly useful one.

```
-module(hashAbs).  
-export([init/1, abstractState/2]).  
  
init(Size) -> {ok,Size}.  
abstractState(State,Size) -> {ok,{erlang:phash2(State,Size),Size}}.
```

Fig. 2. Abstraction module for hashing

### 3.4 Using the Model Checker for Simulation

Recently we have added a simulation facility to the model checker, whereby instead of exploring the whole state space of an application only a single execution branch is followed. Which execution to branch to follow is by default a random choice, however finer control can be exercised by the monitor module above, which in addition to checking safety properties can mark certain states an “uninteresting”, preventing the model checker to examine them and instead choosing an alternative state in simulation mode.

The checking of the Robocup agents has necessitated implementation of “real-time support” for the McErlang model checker as well. The player agents are highly time dependent, and have to respond in a timely fashion to information sent from the soccer server by means of acutation commands.

Moreover the model checker had to be “opened up to the outside world”. Agents send commands to the soccer server using UDP sockets, and the soccer

server regularly broadcasts sensory information to all agents. To support sending UDP commands was trivial (we modified the Erlang API function that supports UDP sending), whereas receiving messages was a bit more tricky. The solution was to program a new Erlang process, constantly listening for incoming UDP messages. This (real) Erlang process keeps a map of virtual (simulated) Erlang processes to which incoming messages should be resent (using the virtual message communication mechanism). Thus virtual processes wanting to receive UDP messages on a certain UDP port communicates this to the UDP Erlang process, which in turn starts receiving and forwarding incoming messages on behalf of the virtual process.

Notable is also that when the application has been opened up to the outside world, no longer is the absence of any available enabled transitions a reason to halt the simulation run, since it could very well be that some external process is about to send a message to a (simulated) process. Thus upon a deadlock, with processes waiting to receive data, the simulator has to keep waiting (forever) for the possibility that external (originating outside the model checker) messages may arrive.

## 4 RoboCup teams in Erlang

The IT-university of Gothenburg has been organizing local RoboCup competitions for their students<sup>4</sup> as part of a course for undergraduate students enrolled in a software engineering and management program. Students were asked to develop in groups a RoboCup soccer simulation team in Erlang to play against teams developed by other groups. We have taken two such teams as a starting point for a case-study in verifying properties of complex multi-agent systems.

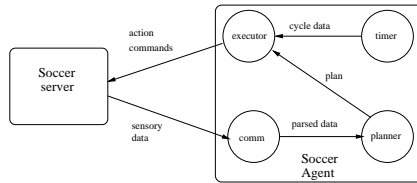
Below we briefly discuss the implementation of the (by far) less complex team to provide an insight into how agents can be programmed in Erlang.

As mentioned before, a RoboCup match is played in client-server style, with a soccer server and two teams of eleven soccer players each. In the case of the first team we have considered, each of the soccer players is composed of four Erlang processes as shown in Fig. 3: a communicator process, a planner process, an executor process and a timer process. All communication between the processes uses Erlang's built-in asynchronous message passing communication facility. Thus the total Erlang application comprises around  $11 * 4$  processes.

In each cycle, the agent process receives messages from the soccer server through an UDP connection containing sensor information, for example, a message "(see Time ObjectInfo)" reports the objects currently seen by the player in a simulation cycle of the soccer server. The messages are parsed and sent to a process which updates and stores the contextual knowledge of the agent, and also to the planner process which elaborates a plan (a list of actions) and sends it to the executor process. The executor receives also information regarding the time from the timer. During each cycle of the game, each player can send a limited

---

<sup>4</sup> see <http://www.ituniv.se/~jalm/ecc06/>



**Fig. 3.** The soccer server and one soccer player

number of action commands. The soccer server executes the commands at the end of the cycle and simulates the next cycle regarding the received commands and the previous cycles data. In our implementation, the executor process sends the action commands to the server, for example, the command “(kick Power Direction)” to accelerate the ball with the given power in the given direction.

To give a flavour of the Erlang code that implements the soccer players, we explain a fragment of the code in the executor module.

```

loop(_, [], ProcName) ->
    loop([wait], [wait], ProcName);

loop(LastPlan, [Action|NextActions], ProcName) ->
    receive
        newCycle ->
            execute(Action, ProcName),
            loop(LastPlan, NextActions, ProcName);
        {plan, NewPlan} ->
            waitNextCycle(ProcName),
            case isEqual(LastPlan, NewPlan) of
                true ->
                    execute(Action, ProcName),
                    loop(LastPlan, NextActions, ProcName);
                false ->
                    execute(head(NewPlan), ProcName),
                    loop(LastPlan, tail(NewPlan), ProcName)
            end
    end
end.

```

The first parameter in the `loop` function represents the currently executing plan (a plan is a list of actions) and the second parameter represents a list containing the actions of the plan that remain to be executed. The `wait` action only indicates that the player is not doing anything but waiting to receive other actions, for example, to kick the ball.

The function `loop` consists of two clauses: one for the case when no actions remain in the current plan, in which case the function `loop` is recursively called

with the same initialization arguments, that is, the player does not have any action to be executed and is waiting.

In the second clause some actions of the current plan remains to be executed; the second function argument is a list with a first element `Action` and a rest. Here the executor process waits to receive one of two types of messages: a `newCycle` message from the timer process or a `{plan, NewPlan}` message from the planner process.

Upon receiving a `newCycle` message, giving permission to start issuing commands to the soccer server, the `execute` function is called with the next action to be performed by the player, followed by a recursive call to the `loop` function with the rest of actions. If a new plan is received first however, the executor waits until the next cycle. Depending on whether the new plan is the same as the currently executing plan or not, the executor will take one of two actions. If the plans are the same, then it keeps on executing the plan. If they are not the same, then the last plan is aborted, i.e., the actions that had not been executed are ignored, and the new plan is executed instead.

In total the number of lines of codes for this Robocup team comprises around 3500 lines of Erlang code (including parses and so on), whereas the more complex (and better playing) team comprises around 8400 lines of Erlang code.

## 5 Checking Robocup Agents

Seen as a verification task, checking properties of a RoboCup team is very challenging. A team consists of eleven to a large extent independently acting agents with complex internal states, that cooperate to solve a common task in real-time. Unfortunately the hostile environment, i.e., the opponent team, strives to greatly complicate the task of solving the task. Moreover, the setting is not static, the number of opponents will vary, and in addition the soccer simulation server contains random features<sup>5</sup> that will alter the outcome of various actions of the agents.

To apply model checking techniques to such a verification problem one would have to construct, with substantial effort, a simplified model of the soccer server, the agents (for both teams) and their environment. Even so the real state space would be huge, and a model checking run would be unlikely to cover more than a very tiny fragment of that state space. For this reason we decided upon a different “verification” strategy: to 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. What we lose by not performing a complete verification, which could anyway never be complete due to the abstractions needed to obtain a verifiable model, we hope to gain by checking the actual source code of the agents.

---

<sup>5</sup> such as e.g. reporting all positional information to an agent with a possible slight error

Concretely to check a soccer team (comprising 11 agents) we ran a number of simulated soccer matches against a number of opposition teams. Each match consisted of two halves of 300 seconds each, with time ticks (events during which the soccer server calculates game changes, and transmits positional information to every player) every 100 milliseconds. These are configurable parameters in the soccer server; McErlang was sufficiently quick to keep up at the default settings. If deadlines are not met, then soccer agents would not act timely on sensory information resulting in bad playing; a symptom of such a problem is increasing message queues of processes, a property we did check during games. To accomplish such real-time execution, with over 50 simulated Erlang processes and checking safety monitors in every global system state, proves that the simulated runtime implementation is not overly slow.

Every message received by any process (implementing an agent) constitutes a new state; with well over 50 processes in a team the total number of possible states is large. In each encountered state during a game, we checked a number of correctness properties (formulated as automatons).

To check a team in varying situations the opposition teams were chosen with care. To evaluate defensive play we matched the team to check against good teams from previous international robocup competitions<sup>6</sup>. Concretely such teams include `fcportugal2004` and `tokyotech2004`, both from the 2004 international robocup competition<sup>7</sup>. For evaluating offensive play a particularly bad student team was selected as an opponent. Finally, to evaluate the team in a more fluctuating situation we played the team against itself. All games were repeated multiple times, to increase the coverage of the verification experiment.

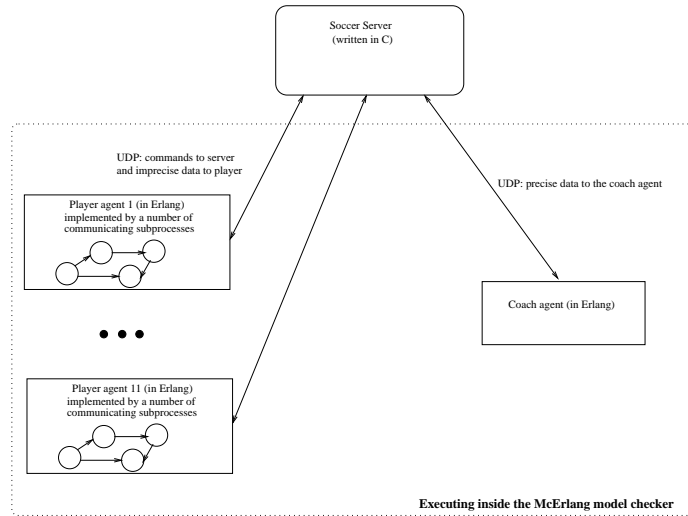
By using McErlang compared to using traditional testing frameworks we obtain a number of advantages:

- correctness properties can be elegantly expressed as automatons rather than sets of tests,
- compared to running a team under the normal Erlang runtime system, the McErlang tool provides detailed control of the scheduling of processes, and delivery of messages (which a traditional runtime system does not provide at all). Testing a multi-agent system under different assumptions regarding processes scheduling can often reveal errors that are difficult to reproduce using normal testing procedures,
- no or very little source code modification is necessary to interpret testing outcome (i.e., as all the team state – including all its agents, and all the processes implementing an agent – can be inspected, there is generally little need to export extra information from an agent).
- since we are using an untyped functional programming language (Erlang) we can treat programs (e.g., pending function calls, sent messages, etc) as data, and analyze such data using powerful data abstraction functions. Moreover

---

<sup>6</sup> as the level of play of the student teams is generally not very good, this was no difficult task

<sup>7</sup> <http://www.robocup2004.pt/>



**Fig. 4.** RoboCup verification setup

we can often reuse functions and data structures used in the program itself, when formulating correctness properties.

We use the monitor concept of McErlang to check properties of a RoboCup team programmed in Erlang during games with opponents. Monitors to check correctness properties of the team are written in Erlang as well, and have full access to the state of all agents (players), message in communication channels, and so on.

However, the states of player agents may of course not reflect reality, as they may have incorrect or simply insufficient knowledge of the state of the game. Clearly to determine whether a property holds, in general we need access to the state of the soccer server as well. As the server is not written in Erlang, McErlang does not have direct access to its internal state. However, by programming a “Coach agent” in Erlang<sup>8</sup>, that repeatedly gets truthful and complete situational information from the soccer server (e.g., ball position, and the position and movement of all players), we gain access, using the McErlang tool, to the complete simulation state.

In case a property violation is detected by a monitor, the complete trace of the simulation up to that point, including the states and actions of all agents and the coach, are available for further analysis in the McErlang debugger.

The experimental setup is depicted in Fig. 4; note that there is no direct communication between agents comprising a team.

<sup>8</sup> the coach interface is provided by the soccer simulation server

## 5.1 Correctness Property Classification

Roughly we can separate desirable properties of RoboCup teams into three kinds:

- **observable properties** can be evaluated by observing only the actions (or inactions) of an agent and its external stimuli, without considering the internal state of an agent
- **discrepancy properties** concern the difference between an agent's beliefs and the objective reality
- **internal properties** concern the general consistency of an agent, and the efficacy of its internal logic decisions

Externally observable properties can be decided solely by examining the data the soccer server sends to the coach process, and the actions (UDP data) sent from an agent to the soccer server. There are countless such properties that can be formulated and checked. For example: “players stay inside the playing field” (*op1*), “the goalie doesn't leave the goal area” (*op2*), “a pass cannot be intercepted by a player from the opponent team” (*op3*), and so on. An obvious externally observable property is that an agent may never crash (*nocrash*).

If we find that such an observable property is violated, the cause can either be that the internal logic of the agent is faulty, or that the agent is acting correctly but on faulty data.

An example of a discrepancy property is: “the difference between the believed position of a player and its real position must not exceed some safety margin” (*dp1*). Discrepancy properties requires us to examine both the objective state of the robocup simulation (the information sent to the coach process) as well as the internal beliefs (internal data structures) of the processes comprising a player agent.

Internal properties range from quite general properties such that: “the size of a message queue is never greater than some limit  $l$ ” (*mq*) to very specific properties. As an example, we can reformulate the property about safe passes (*op3*) above into a property about the internal state of agents: “the agent never attempts a pass when it knows that an opponent player may intercept the ball” (*ip3*). Note that it is perfectly possible for an agent to fail the property *ip3* while not failing *op3* (or vice versa) if the knowledge of the position of the players of the opposition team is particularly poor (a discrepancy property).

While it is easy to formulate such high-level properties in English, with much ambiguity, the challenge is to formulate these properties precisely, and to provide a framework for determining whether they are satisfied by robocup teams or not. Below we exemplify how this is achieved for two such teams.

## 5.2 Verification of the first RoboCup team

The first RoboCup team analysed by us was rather simplistic in nature, generally being reactive (every new sensor information causes a complete new plan to be formed) rather than proactive (players have long term plans that they attempt to realize).

We exemplify the specification of properties by formulating a simple observable property, i.e., that no player strays far outside the playing area (*op1*). As explained earlier, such a property can be checked by examining the accurate information sent to the coaching process, without considering the internal states of agents.

```
stateChange(State,MonState,Stack) ->
  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.
```

The function `stateChange` is called by the model checker every time a new state is generated. Its arguments are the new state `State`, the previous monitor state `MonState` (for this property the coordinates for the allowable area for a player), and the entire stack of program states leading to the current state<sup>9</sup>.

The implemented monitor begins by extracting the state of the “coach” process using the function `coach:getCoachState`. This function attempts to retrieve the process datastructure of the coach process using its name “coach”, and from that structure the internal datastructure that records the data sent from the soccer server. This is achieved by accessing the `expr` field of a process datastructure, that records the current state of the process (always waiting for a new UDP message to arrive from the soccer server) together with the state data (a part of which is the internal datastructure that records data sent by the soccer server).

```
getCoachState(State) ->
  %% Retrieve process named "coach"
  P = findProcessByRegisteredName("coach",State),
  case P#process.expr of
    {recv, {_,_},{_,CoachState}}, _} -> CoachState
  end.
```

---

<sup>9</sup> The stack implementation is also parametric and we frequently use a bounded stack which forgets old states when runs become too large

If this fails (e.g., probably because the coach process has not been created yet) the `try...catch` statement ensures that the simulation continues. Given the coach state, the `coach:getOwnPlayers` function returns the information sent by the soccer server regarding the team players (e.g., position and so on). It is then easy to compute whether any player strays outside the allowable area, and if so the monitor returns `error` which indicates to the model checker that an error has been encountered. As a result the simulator will offer the possibility to examine in detail the trace leading to the erroneous state detected by the monitor.

During testing, the checker quickly produced a run leading to a violation of this property.

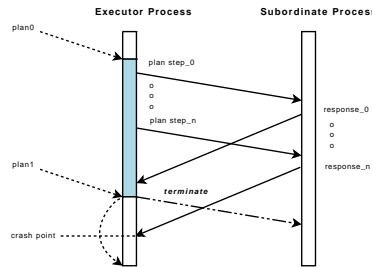
As another example, we formulated the *op2* property, i.e., that the goalie doesn't leave the goal area, and attempted to verify that property. This property is also easily checked using the knowledge from the coach process. Unfortunately the first team fails even this simple property: the goalkeeper was very far from his penalty area. However, from visual inspection (using the soccer monitor to view a game progress) the goalie did not appear to have left his penalty area. Program inspection found the source of the error: the team had assumed a fixed assignment of player numbers to their processes, whereas the soccer server could randomly assign player numbers. In other words, the player which resided in the penalty area was in fact not the goal keeper, thus the player did not have permission to handle the ball with his hands.

In conclusion, the first team analysed possessed grave problems indeed, and we didn't think it interesting to consider the analysis further at that point but continued with the second team.

### 5.3 Verification of the second RoboCup team

The second RoboCup team analysed is far more complex, comprising more lines of code, and having a much more complex internal state. Although generally playing much better than the first team, we were able to discover a number of bugs that had gone undetected using normal testing techniques. We illustrate the kinds of properties checked, and the bugs found, using small examples below.

**Observable properties** When trying to execute the second RoboCup team under the McErlang model checker, and playing a game, the program sometimes crashed with an error message. This had never been experienced when running the team outside the model checker. The reason for the crash was typical of a class of hard-to-reproduce errors which occur only for some very intricate sequences of concurrent actions. Players of the second team are composed of different processes; one such process responsible for executing plans have two states: `idle` when it awaits a new plan, and `execute` when it executes a received plan. The execution of each step of a plan is performed by a subordinate process, which reports the success or failure of the step back to the executor process. The executor process and its subordinates communicate using asynchronous message



**Fig. 5.** RoboCup Agent Bug

passing. The error occurs when a second plan reaches the executor process before it has finished executing a prior plan. In such a situation the process (correctly) terminates the subordinate process, resends the second plan to itself, and enters the `idle` state (awaiting the second plan sent to itself). However, it turns out that the subordinate process may have sent a message to the executor process that arrived after it was itself terminated, but arriving before the second plan (when the executor process was in the `idle` state). Moreover the executor process was not able to handle incoming messages from a subordinate process in its `idle` state, leading it to crash.

A graphical depiction of the error is given in Fig. 6, where the `idle` state (of the executor process) is in white, and the `executor` state is in blue.

During extensive testing of the team this error had never been seen, however using McErlang (which has a much less deterministic agent scheduler compared to the normal Erlang runtime system) the error was immediately discovered.

We also coded up the property of safe passes (*op3*); however, it turned out that agents could attempt quite unsafe passes. A possible reason for such unsafe behaviour is analysed below, in the formalisation of a logic property (*lp3*).

**A discrepancy property** A central correctness property is whether the agent's beliefs of the position of the ball in the field is accurate or not.

To formulate the property we have to compare the belief about the ball position, as retrieved from the internal store of an agent, compared to the real position of the ball as given by the coach agent. Clearly these values can be substantially different, for instance if the ball is kicked away behind the back of a player. However, we want to require the agent to eventually correct his ball estimate.

The formulation of the property will thus be parametric on two parameters: i) what is a bad estimate, and ii) for how long time must an estimate be continuously bad until an error is signalled.

The implementation of the property as a state monitor is as follows:

```
stateCheck(State, _Actions, Mst={Time, Parameters={BadEstimate, Interval}}) ->
try
```

```

%% Obtain coach information then check if game is halted, and obtain pos info
{ok, CoachState} = coach:getCoachState(State),
play_on == CoachState#coachInfo.play_mode,
CurrentTime = CoachState#coachInfo.time,
CoachPlayer = coach:getOwnPlayer(Number,CoachState),
CoachPlayerPos = coach:playerPos(CoachPlayer),
CoachBall = coach:getGall(CoachState),
CoachBallPos = coach:ballPos(CoachBall),

%% Obtain Agent internal data, including ball and player position
Player = kb:ask({player,myself}),
Ball = kb:ask(ball),

%% Calculate Players distance approximation and accurate distance
DistPlayer = dist(Player#player.position,Ball#ball.position),
DistCoach = dist(CoachPlayerPos,CoachBallPos),
Error = abs(DistCoach-DistPlayer)/DistCoach,

if Error >= BadEstimate ->
case Time of
%% First time bad estimate seen, set timer
ok -> {ok,{{until,CurrentTime+Interval},Parameters}};

%% Estimate bad during interval, report error
{until,EndTime} when EndTime =< CurrentTime -> badEstimate;

%% Estimate continuously bad, not end of interval
{until,EndTime} when EndTime > CurrentTime -> {ok,Mst}
end;

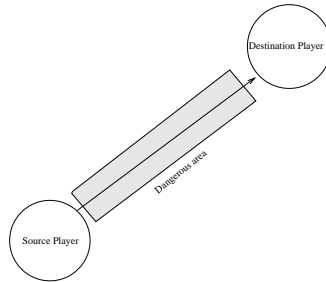
%% Estimate is good
true -> {ok,{ok,Parameters}}
end
catch ... end

```

The function `dist` calculates the distance between two points, and the function `kb:ask` (present in the Agent source code) returns the belief of the agent regarding its parameter. Note that the `kb:ask` function is called without a parameter specifying the player; this is because the monitor is executed in the context of the agent that caused the last program step.

The error is calculated as the quote between the believed distance to the ball and the real one, divided by the real distance.

There is indeed dubious beliefs in the second agent as well. In one run we found that with an error of 0.5 (meters) and 10 time units (intuitively a second), the model checker discovered a situation where a player thought the distance to



**Fig. 6.** Property *ip3*

be around 7.6 meters, for over a second, while the real distance hovered around 17 meters.

**A logic property** To illustrate the coding of a logic property we considered first the property (*ip3*): “the agent never attempts a pass when it knows that an opponent player may intercept the ball”.

We can illustrate the idea of the property using figure ??; there may be no opponent player in the gray zone around the (believed) path of the ball from its originating player to the destination.

In each state the formalisation of the property has to determine whether a pass attempt has been made. This turned out to be rather difficult, as the difference between what is a pass, shooting or just clearing the ball in a dangerous situation is hidden quite deep in the code (and all three operations are executed by sending a “kick” command to the soccer server). In the end it turned out to be easier to modify the player program in one point, by introducing a new (artificial) state labelled by a tuple encoding the operation of passing the ball, and the destination position of the corresponding kick:

```
{do_pass, TargetPosition}
```

After the introduction of this “probe state”, the property is easily specified. Essentially whenever a kick is made, it is necessary to retrieve from the player agent its beliefs about the opponent players and calculate whether any of these players are too close to the path of the ball between source and target.

Running an example quickly reveals that the second agent exhibits dubious behaviour. 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.

## 6 Conclusions

In this paper we have used the McErlang model checking tool to perform runtime verification on a set of agents comprising a RoboCup soccer simulation team written in the Erlang programming language. Correctness properties were specified as monitors (automatons) observing the detailed behavior and states of all the players in the team, and the opponent team. The agents checked were not modified nor abstracted in any way for the purpose of the study, rather the standard source code of the agents was used essentially unchanged. One of the key functionalities of the McErlang tool is the capability to observe the inner state of agents, and of coping with temporal agent behaviors.

The properties checked include a number of obvious correctness criteria for soccer play (respecting playing field boundaries etc), including also a number of properties that concern the inner logic of the agents. We aim to continue this experiment in order to formulate further more detailed properties regarding the internal state of agents (beliefs, plans, etc) of more RoboCup teams and agents, to further illustrate the practicality of the approach.

## References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.
2. T. Arts and L. Fredlund. Trace analysis of Erlang programs. *SIGPLAN Not.*, 37(12), 2002.
3. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.
4. T. Bosse, D. N. Lam, and K. S. Barber. Automated analysis and verification of agent behavior. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1317–1319, New York, NY, USA, 2006. ACM.
5. M. Chen, K. Dorer, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, J. Murray, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wang, and X. Yin. *RoboCup Soccer Server*, 2003. Manual for Soccer Server Version 7.07 and later (obtainable from `sserver.sf.net`).
6. J.-C. Fernandez, H. Gavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.
7. L. Fredlund. Implementing WS-CDL. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*. Universidade de Santiago de Compostela, November 2006.
8. L. Fredlund and C. B. Earle. Model checking Erlang programs: The functional approach. In *ACM Sigplan International Erlang Workshop*, Portland, USA, 2006.
9. L. Fredlund and J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

10. L. Fredlund and H. Svensson. McErlang: a model checker for a distributed programming language. In *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, 2007.
11. U. Furbach, J. Murray, F. Schmitsberger, and F. Stolzenburg. Model Checking Hybrid Multiagent Systems for the RoboCup. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *Proceedings of the RoboCup Symposium 2007*, 2007.
12. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
13. S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.
14. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. pages 332–344, 1986.
15. W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.