Instituto de Engenharia Electrónica e Telemática de Aveiro

RoboCup 3D Simulator
FCPortugal User's Guide v0.1

Hugo Miguel Gravato Marques

Supervisors : Professor Nuno Lau and Professor Luis Paulo Reis

August of 2004

Content Index

1 Introduction 1
1.1 Motivation 1
1.2 Organization 1
1.3 Who should see the manual? 2
2 Installation of the 3D Simulator 3
2.1 Packages needed 3
2.2 HowTO install 3
Expat 3
Spades 4
Boost 4
Ruby 4
ODE 4
Rcssserver3D 5
3 SPADES 6
3.1 General Overview 6
3.2 Components organization 6
3.3 States (Sense-Think-Act) 7
3.4 Agent Generic Input Format 8
3.5 Agent Generic Output Format 9
4 The 3D Simulation Server 10
4.1 Introduction 10
4.2 Communication Procedures 11
4.3 The orientation of the game and the coordinates' system 12
4.4 Sensations - Perceptors 13
Vision 13
Game State 13
Agent State 14
Extra Preceptors 15
4.5 Actions – Effectors 15
Create 15
Init 16
Drive 16
Kick 16
4.6 Physics 17
The agent movement 17
The ball movement (in the ground) 18
The ball movement (in the air) 19
The kick 19
4.7 Important Files 19
rcsoccersim3D 19
rcssserver3D 19
rcssmonitor3D 20

rcssmonitor3D-lite 20
agentdb.xml 20
rcssserver3D.rb 21
5 Start agent 22
5.1 Communication 22
5.2 Sketch of the agent 22
6 FCPortugal Agent 24
6.1 Organization 24
World state 24
Physics 25
Geometry 25
Skills 25
Actions 26
Utils 27
6.2 Scketch (behaviour) 27
6.3 Localization system 28
6.4 Physics 29
Agent movement 29
Ball movement (in the ground) 30
Velocity calculation 32
6.5 Skills 36
Move 36
Kick 36
Dribble 37
Interception 37
6.6 Actions 38
Pass 38
Shoot 38
Forward 38
6.7 Future work 38
Geometry 38
Physics 39
7 Bibliography 40

1 Introduction

1.1 Motivation

The RoboCup project encourages research on AI, namely on agents and robotics. Its goal is "by 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer" (Kitano et al 1995). RoboCup is constituted by 5 different types of leagues: small size robots, middle size robots, junior, legged robots and simulation. The scope of this report is focused on the last type – the simulation league. The simulation league, until now, has just been played in two dimensions using the 2D simulator server shown in Figure 1 (Chen et al 2002), but this year, a three dimension simulation league was created. The aim of the present document is exactly to make the introduction and use of RoboCup 3D Soccer Simulator faster and easier.

This research was supported by the FCT (Fundação para a Ciência e Tecnologia) organization in the scope of the POSI/ROBO/43910/2002 Project – "FC Portugal New Coordination Methodologies applied to the Simulation League"

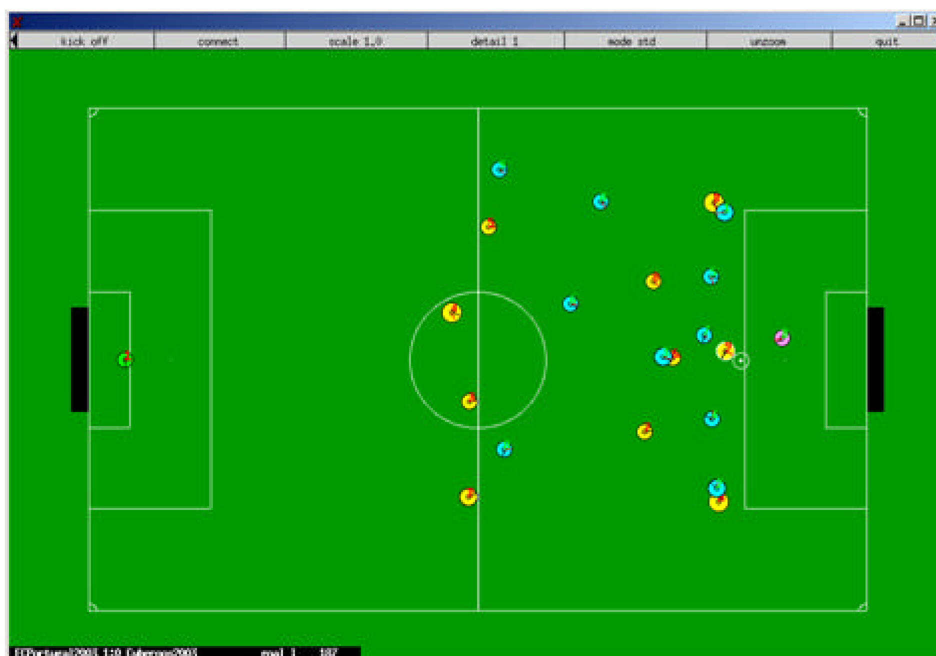


Figure 1 – Snapshot of the 2D simulator.

1.2 Organization

The manual is organized in 7 chapters. The first is the present one, which gives an overview of what can be found in the rest of the manual. The second indicates how to install RoboCup 3D Soccer Simulator (rcssserver3D) and which tools need to be installed. The third aims to

introduce the SPADES system which is very important to understand the functioning of the Soccer Simulator 3D. The forth explains how the 3D simulator server works. The fifth presents an agent test. Finally, the sixth explains the architecture of the FC Portugal agent and some details of its implementation.

1.3 Who should see the manual?

This manual was made to people who want to know the way the 3D simulator works. It is also made to help people who aim to build agents in order to interact with the RoboCup 3D Soccer Simulator. Finally it is very useful also for people who make part (or will make part in the future) of FC Portugal team and want to understand the current architecture and behaviour of the FC Portugal agents.

2 Installation of the 3D Simulator

2.1 Packages needed

Here are all the necessary packages to run soccer simulator 3D and the places where each of them they can be found.

- **Full SPADES System** - http://sourceforge.net/projects/spades-sim

  The SPADES (System for Parallel Agent Discrete Agent Simulation) is a generic middleware piece of software that allows creating agent-based distributed simulations.

- **rcssserver3D** - http://sourceforge.net/projects/sserver/rcssserver

  The Rcssserver3D is an agent-based distributed simulation of a soccer's game in 3 dimensions.

- **expat.h v1.95.7** - http://prdownloads.sourceforge.net/sourceforge/expat/expat-1.95.7.tar.gz

  The Expat is an XML parser toolkit.

- **boost** - http://sourceforge.net/project/showfiles.php?group_id=7586

  The Boost is a set of libraries that aims to establish an "existing practice" and to provide reference implementations so that the libraries are suitable for eventual standardization.

- **ruby** - http://www.ruby-lang.org/en/

  The Ruby is an interpreted scripting language for quick and easy object-oriented programming.

- **ode** - http://sourceforge.net/project/showfiles.php?group_id=24884

  The Open Dynamics Engine (ODE) is a software library for the simulation of Rigid Body Dynamics. It is useful for simulating things like vehicles, objects in virtual reality environments, and virtual creatures.

2.2 HowTO install

These are the basic steps to install rcssserver3D. One may need to install some other packages that are not present in the Operating System. Before you start installing the components download the all the above packages to a directory (`~/install-root` for example).

After decompress all packages the install-root directory should appear like:

```
~/install-root/
      expat-root/
      spades-root/
      boost-root/
      ruby-root/
      ode-root/
      rcssserver3D-root/
```

Expat

Install `expat.h` in case that you don't already have it. To install it:

- decompress `expat-1.95.7.tar.gz` (or later) to `install-root/expat-root/`
- `cd expat-root`
- `configure`
- `make`
- `su`
- `make install`

Spades

To install it:

- decompress `spades-0.91.tar.gz` (or later) to `install-root/spades-root/`
- `cd spades-root`
- `configure`
- `make`
- `su`
- `make install`

Boost

To install it:

- decompress `boost-1.30.2.tar.bz2` (or later) to `install-root/boost-root/`
- set environment variable `CPPFLAGS=[boost_root]` with `-I` option. Ex.: `export CPPFLAGS=-I/boost_root` (it is advisable to use the absolute path).

Ruby

To install it:

- decompress `ruby-1.8.1.tar.gz` (or later) to `install-root/ruby-root/`
- `cd ruby-root`
- `configure`
- `make`
- `su`
- `make install`

ODE

To install it:

- decompress `ode-0.039.tgz` (or later) to `install-root/ode-root/`
- `cd ode-root`

- `make` - to make sure that the options are correct look at file `ode_root/config/user-settings` (in principle they are)
- set environment variable `ODE=ode_root`

Rcssserver3D

To install it:

- decompress `rcssserver3d-0.1.tar.gz` to `install-root/rcssserver3D-root/`
- `cd rcssserver3D-root`
- `configure` – in case of errors take special attention to:
    - boost_root - make sure that boost_root appears on `CPPFLAGS` with –I option;
- `make` - in case of errors take special attention to:
    - ruby library - the name of the library can be incorrect on `RUBY_LDFLAGS` and `AM_LDFLAGS` of the makefile; check the name of the library and change them if needed)
    - `glut.h` - this OpenGL library is needed; make sure that you have it installed. Usually it may be found in `/usr/X11R6/include/GL` directory.
    - libGL - in case that the SO is not able to find lGL (libGL; first make sure that is installed (`find /usr/ -name libGL*`); if it still is not able to find it, substitute the expression "`-lGL`" in `LDADD` attribute on Makefile by file and its path
- `su`
- `make install`

3 SPADES

3.1 General Overview

The System for Parallel Agent Discrete Agent Simulation (SPADES) is a middleware system for agent-based distributed simulation (Riley 2003). It aims to provide a generic platform that implements the basic structure to allow the interaction between agents and a simulated world in such a way that the users do not have to worry about sockets, addresses, etc.

SPADES main features are:

- Agent based execution - support to implement sensations, thinking and actions.
- Distributed processing – support to run the agents-application on many computers.
- Results unaffected by network delays or load variations among the machines – SPADES ensure that the events are processed in the appropriate order.
- Agents can be programmed independently from the programming language – the agents can be programmed in any language once it provides methods to write/read to/from PIPEs.
- Actions do not need to be synchronized in the domain – the actions of the agents can take effect at varying times during the simulation.

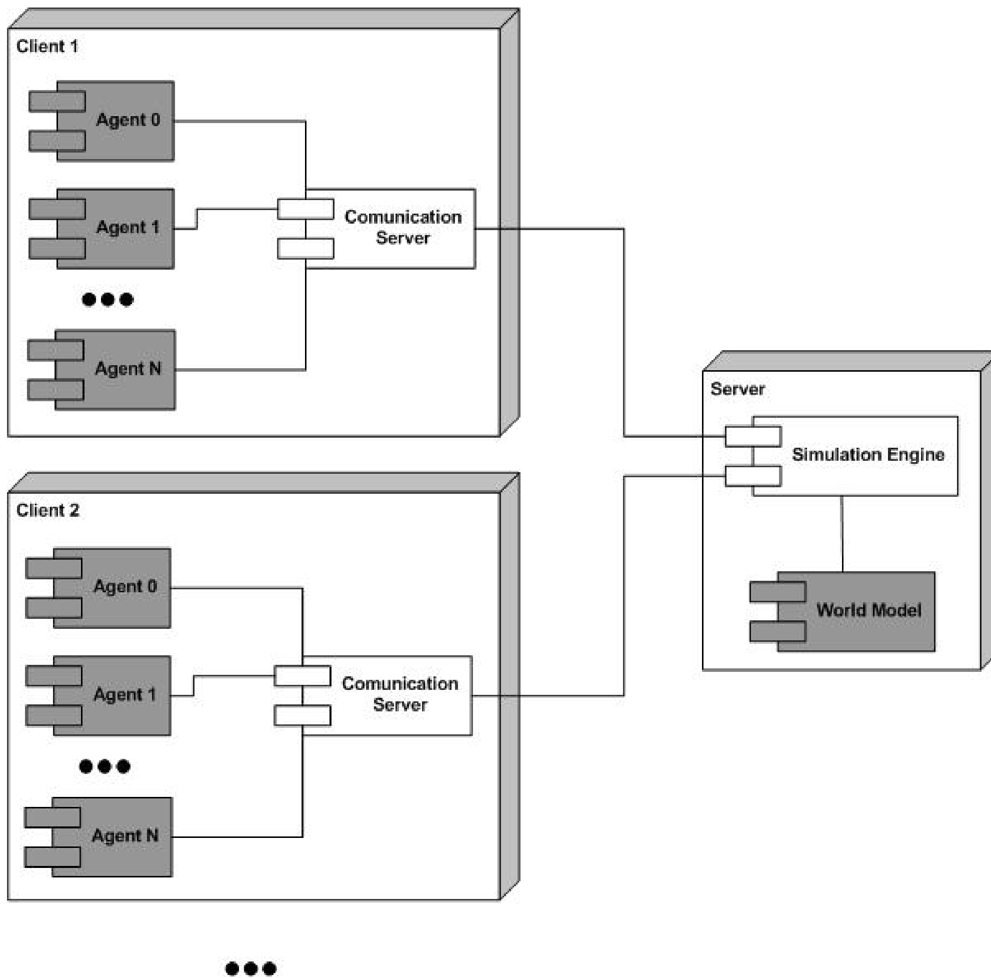
To simplify the creation of simulations SPADES provides also:

- An abstraction for an event based simulation to the world model – the world model can create simulation events and process them.
- An abstraction for sensations and actions to the agent – the agent simply needs to read/write to/from a PIPE.
- A tracking of the amount of computation made by each agent.
- The network management – the user do not have to worry about sockets, addresses, etc.
- The management of the distribution of the simulation among the machines – the results of the simulations are not affected by the number of machines used, network load, machine load, etc.
- Logging facilities – logging for errors and trace execution as well as a record of the states of the simulations can all be managed by SPADES.

3.2 Components organization

SPADES components are organized in a client-server architecture (Figure 2). The Simulation Engine and the Communication Server are supplied as a part of SPADES; while the Agents and the World Model are built by the user and run upon the formers. The Simulation Engine is a generic piece of software that allows creating specific world models upon it. It runs on the server side and provides the interaction and communication between the agents and their world via the Communication Server. On the same machine it must run the specifications of the World Model that has the characteristics of the environment where the agent will act.

Distributed along the clients are the Agents and the Communication Server. The Communication Server must be present in all client machines[1] and provides the communication between the Agents and the Simulation Engine. It receives messages from the Agents and sends them to the server and vice-versa. The Agents also run on the client side.



●●●

Figure 2 – Components organization.

3.3 States (Sense-Think-Act)

SPADES implements what it calls the sense-think-act cycle in which an agent receives sensations and replies with actions. That means that an agent is only able to act as a reply to a sensation message. However, it is capable of requesting its own sensations, but the principle

[1] Unless, of course, everything is running in only one machine.

remains - a sensation must always precede an action. Following this, SPADES provides an action called `request time notify` that returns an empty sensation (`time notify`) and by receiving it the agent is able to respond with actions.

In Figure 3 we can see the sense-think-act cycle and the time where each of its components run. From A to B a sensation is sent to the agent. After receiving the sensation (from B to C) the agent decides which actions will be executed; then from (C to D) the actions are sent do the server.

In many agents, the sense, think and act components may be overlapped in time (like in Figure 3); there is just one restriction – the thinking cycles for one agent can not be overlapped. This constraint makes sense, since just a single processing unit is used per agent, and thus, just one sensation at time can be processed.

Lastly, SPADES does not require that the components of the cycle have the same latency.

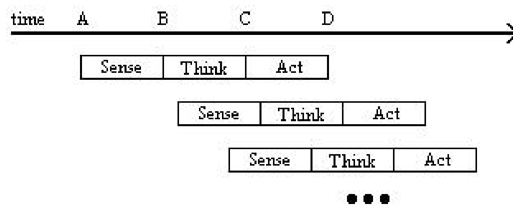


Figure 3 – Sense-Think-Act Cycle.

3.4 Agent Generic Input Format

This section describes the format of the messages sent by the simulation server to the agent [1]. Throughout this section the variables used are:

- `time` – simulation time (an integer).
- `data` – data string to be interpreted by the simulation server, thus it must obey its specific protocol.
- `token` – text string without spaces.

The messages sent by the server to the agent are:

- `Ddata` – initialization message – After the agent start up this message is sent by the server in order to inform the agent that it is able to proceed with the the initialization procedures. After the initialization is complete the agent should sent a *done* message (see next section).
- `Stime time data` – sensation message – this message gives the agent information about the state of the world from its self point of view. The first *time* is the time that the message was sent while the second *time* is the time the message was delivered to the agent. The agent can reply with act messages and it must finish with a done message.
- `Itime time data` – inform message – The goal as well as the protocol is the same the sensation message above; however this one does not start a thinking cycle - its just informative. The agent should not reply this message.

- `T`*`time`* – time message – This is a time notify message and is respective to the time when the message was requested. The agent can reply with act messages and it must finish with a done message.
- `O`*`time`* – time message – this is a time notify that does not starts a thinking cycle and thus, the agent should not reply to it.
- `X` – exit message – this message notifies the agent that it must shutdown. No reply should be sent by the agent.
- `M` – migration message – not applicable to Soccer Simulator 3D.
- `K`*`time`* – think time message – This message informs the agent of how much thinking time was used for the last thinking cycle.
- `E`*`token`* – error message – This message informs the agent that the error specified within the token string had occurred.

3.5 Agent Generic Output Format

This section describes the format of the messages sent by the agent to the simulation server [1]. Throughout this section the variables used are:

- *`time`* – simulation time (an integer).
- *`data`* – data string to be interpreted by the simulation server, thus it must obey its specific protocol.

The messages that can be sent by the agents to server are:

- `A`*`data`* – action message – This message informs the server about the actions taken by the agent.
- `R`*`time`* – request time message – This message requests the time to the serv er. The *time* is the time for which the notification is requested.
- `D` – done thinking message – This message informs the server that the thinking cycle is complete. All actions and requests must be sent before the done thinking message.
- `M` – migration message – not applicable to Soccer Simulator 3D.
- `X` – exit message – This message informs the server that the agent will exit. It can be sent at any time.
- `I` – initialization done message – this message informs the server that the agent initialization is complete.

4 The 3D Simulation Server

4.1 Introduction

As stated before the simulator runs upon the SPADES, and uses ODE to calculate the physical interactions between the objects of the world (Russell 2003). The graphical interface is implemented using OpenGL.

Figure 4 – Snapshoot of the 3D simulator server.

The 3D simulation server (RoboCup Soccer Server 3D Maintenance Group 2003) allows twenty two agents (eleven from each team) to interact with the server in order to play a simulated robotic soccer game (Figure 4). Each agent receives sensations about the relative position of the other players, the ball, the flags and the goals and other information concerned with the game state and conditions. At this time the information about the positioning of the objects in the world is given by an awkward omni-vision that allows the agent to receive visual information in 360 degrees. The agents have the shape of a sphere.

Replying each sensation an agent sends actions like drive or kick. Driving implies applying a force on its body with a given direction and kicking implies applying a force on the ball

radially to the agent. Each sensation is received on every 20 cycles of the server and each cycle takes 10 ms.

4.2 Communication Procedures

The server establishes the communication with the agents by sending a done ('D') message (Riley et al. 2003). When the agent receives this message it should execute its initialization procedures and when it finishes them it must send an init done ('I') message. After that the server starts to send sensations and the agent replying with actions. Every set of actions must finish with a done ('D') message.

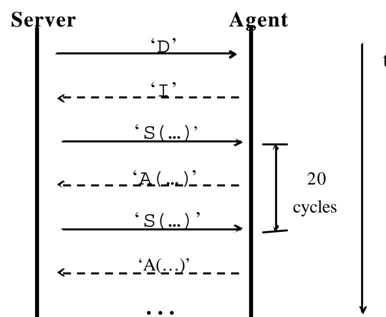


Figure 5 – Communication between the agent and the server.

As already stated, each sensation is received in every 20 server cycles, which means that an agent should only be able to execute actions within 20 cycles intervals. However, this is not quite so, since an agent can ask the server to receive a sensation in a given cycle by sending a request time notify ('R') message. The format of the message is the following:

R*time*

where time is the time at which the server must reply. This procedure makes the server reply with an empty sensation ('T') at the cycle given. As stated before the only reason to receive an empty sensation is to be able to act in a given time between two sensations. During the experiments made one could check that if a request time notify was asked for an earlier period than the current cycle, the server allowed the actions to be performed anyway. The format of the message received is:

T*time*

where time, again, is the server cycle in which the message was sent.

An example where the request time notify makes sense is when an agent wants to kick the ball in given position. In the 3D server simulator, to kick the ball in a given direction, an agent must place itself quite accurately. Thus, because the interval between sensations is too long (200ms), it can happen that an agent that is running to the kicking point at cycle t is before that point and at cycle $t + 1$ has already passed the point. To surpass this, one can predict the

time that the agent arrives to the desired position and ask to receive a time notify message at that time in order to be able to kick at the right moment.

By receiving this sensation the agent is able to respond with action messages.

An action takes 10 cycles to reach the agent (send delay) and the action sent starts to take effect at the time the next sensation is sent by the server like Figure 6.

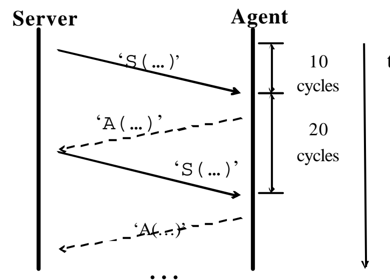


Figure 6 – Communication delays.

4.3 The orientation of the game and the coordinates' system

The field has 8 landmarks – one flag in each corner and 2 goals in each frame – which can be used to calculate the localization of each agent on the field (Figure 7). The landmarks are fixed, meaning that their identification do not vary neither with the team nor the game half. The side of each team remains unchanged during the second half of the game - the left team attacks always to the right side and the right team always attacks to the left side. The x axis is always directed to the attack of the team, while the y axis changes according with the side of the team.

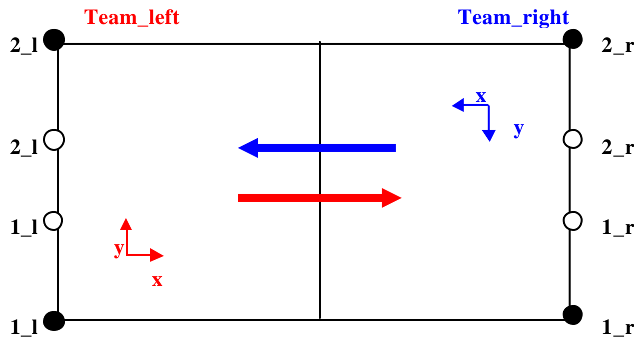


Figure 7 – The flags and the coordinates system used on the 3D simulation server.

4.4 Sensations - Perceptors

A perceptor is a mechanism that allows the agent to receive information from the outside world. The present version of the server implements 3 types or perceptors:

Vision

Localization: rcssserver3D-root/plugin/soccer/visionpercector/visionperceptor.cpp

It allows an agent to have information about the position of the ball as well as the position of every player and landmark on the field. The position of the objects is given by polar coordinates (Figure 8) with origin on the agent and with the x axis horizontally pointed to the attack direction. The first coordinate indicates the length of the vector from the player itself to the object, the second coordinate indicates the horizontal (the plane of the game) angle (a) and the third coordinate indicates the vertical angle. The angles vary between *[-180º, 180º]*.
The vision system has two types of errors: a calibration error and a noise on the parameters. The calibration error is relative with the camera position and, for each axis, the error is uniformly distributed between *[0.005, 0.005]*. This error is calculated once and remains constant during the complete match. The noise error in the three parameters is normally distributed, for all, around 0.0 and the sigma varies according with the type of parameter. The distance error has $s = 0.0965$, the horizontal angle error (x-y plane) has $s = 0.1225$ and the latitudinal angle error has $s = 0.1480$.

```
Sint int (Vision
             (Flag (id string) (pol float float float)) ...
             (Goal (id string) (pol float float float)) ...
             (Ball (pol float float float))
             (Player (team string) (id int) (pol float float float)) ...
         )

Sample:
S12 10 (Vision (Flag (id 1_l) (pol 77.13 27.82 -0.32)) ...
               (Goal (id 1_l) (pol 68.31 3.07 -0.35)) ...
               (Ball (pol 14.88 0.03 -1.22))
               (Player (team FCPortugal) (id 1) (pol 29.53 57.84 -0.40)) ...
       )
```

Game State

Localization: rcssserver3D-root/plugin/soccer/gamestateperceptor/gamestateperceptor.cpp

The game state perceptor gives the agent information concerning the game, like its number and team, the sizes of the objects, the current time and the current playmode. The sample shows the information concerning the agent with the number 1 of the FC Portugal team before the game start.

```
Sint int (GameState
               (unum int)
               (team string)
               (FieldLength float)
               (FieldWidth float)
               (FieldHeight float)
               (GoalWidth float)
               (GoalDepth float)
               (GoalHeight float)
               (BorderSize float)
               (AgentMass float)
               (AgentRadius float)
               (AgentMaxSpeed float) // do not use - this information is wrong
               (BallRadius float)
               (BallMass float)
               (time float)
               (playmode string)
           )


Sample:
S12 10 (GameState (unum 1) (team FCPortugal) (FieldLength 104)(FieldWidth 72)
                  (FieldHeight 40) (GoalWidth 7.32) (GoalDepth 2)
                  (GoalHeight 0.5) (BorderSize 10) (AgentMass 75)
                  (AgentRadius 0.22) (AgentMaxSpeed 10) (BallRadius 0.111)
                  (BallMass 0.43) (time 0) (playmode BeforeKickOff))
```

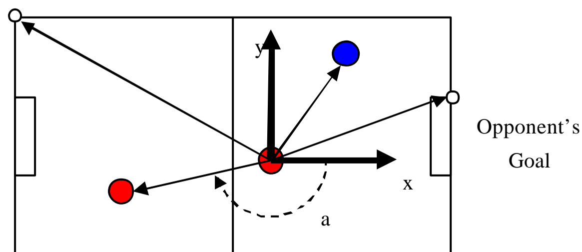


Figure 8 - The way the preceptor indicates the localization of objects in the field.

Agent State

Localization: rcssserver3D-root/plugin/soccer/agentstateperceptor/agentstateperceptor.cpp

The agent state perceptor informs the agent about its current condition. At the moment just battery and temperature are considered but temperature is not even used.

```
Sint int (AgentState (battery float) (temp float))

Sample:
S12 10 (AgentState (battery 100) (temp 23))
```

The sample shows that the agent has full battery (100) and its temperature is 23 ºC.

Extra Preceptors

Localization: ~/.rcssserver3d/rcssserver3D.rb

One can add extra perceptors to the server by changing the *rcssserver3D.rb* file. This is especially useful for example to give the exact locations of the objects in the field. Thus, to add a perfect vision perceptor one just has to add the following lines to the *rcssserver3D.rb* file.

```
perfectPerceptor = new('VisionPerceptor', path+'PerfectPerceptor')
perfectPerceptor.setSenseMyPos(true);
perfectPerceptor.addNoise(false);
perfectPerceptor.setPredicateName("Perfect");
```

This will add a vision sensation starting with the string “Perfect” to the sensation message. The perfect sensation will not have noise and it will have an extra expression named “mypos” followed by the exact coordinates of the agent in the field.

This can be extended to other preceptors just by changing the predicate name (the expression that will start the message) and the preceptor properties.

4.5 Actions – Effectors

An effector is a mechanism that allows the agent to produce an action on the world. There are 4 types of effectors that an agent is able to use:

Create

Localization: rcssserver3D-root/plugin/soccer/createeffector/createeffector.cpp

This is the first action that an agent must perform. It simply creates the agent in the world:

```
Action (create)

Sample:
A(create)
```

The sample shows the way to create an agent.

Init

Localization: rcssserver3D-root/plugin/soccer/initeffector/initeffector.cpp

The “init” action must follow the “create” action. It allows associating an agent to a given team:

```
Action (init (unum int) (teamname str))

Sample:
A(init (unum 0) (teamname FCPortugal))
```

The sample allows initializing the agent as belonging to the FC Portugal team. The *unum* informs about the PID of the agent. In case of *unum* = 0 the server automatically attributes a number to the agent and informs it about its *unum* using the *GameState* preceptor. Otherwise the agent stays with the number it sends.

Drive

Localization: rcssserver3D-root/plugin/soccer/driveeffector/driveeffector.cpp

It allows moving the agent by applying a force to it:

```
Action (drive (pol float float float))

Sample:
A(drive 20.0 10.0 0.0)
```

The sample shows how to move the agent by applying a force of (20.0; 10.0) on the horizontal plane. The agent is not able to jump yet, so applying a force on the z axis is useless. The maximum length of the force that can be applied is 100.0. The drive force is applied until an action disables it. That means that if one does not want to apply a given drive force anymore it must send a drive action with all the parameters set to 0 (zero).

The noise on each axis is normally distributed around 0.0 with $s = 0.005$.[2]

Kick

Localization: rcssserver3D-root/plugin/soccer/visionpercector/visionperceptor.cpp

It allows kicking the ball if an agent is at a kickable distance[3] from the ball:

[2] Values taken from *rcssserver.rb* (section 4.7) – *setSigma()* function of the drive effector..

[3] *Kickable distance = AgentRadius + BallRadius + KickMargin*. *KickMargin* is defined in *rcssserver.rb* (section 4.7) using *setKickMargin()* function of the kick effector.

```
Action (kick float float))

Sample:
A(kick 20.0 100.0))
```

The sample shows a kick with maximum strength (100.0) with a vertical angle of 20.0º.

The noise on the kick effector is quite different from the drive effector. In the horizontal plane (x-y) the angle error is very low – normally distributed around 0.0 with *s = 0.02*. The latitudinal angle error is normally distributed around 0.0. The sigma factor varies – *s=0. 9* at the end positions (at 0 and at 50 degrees) and *s=4.5* towards the middle of the range. The kick power error is normally distributed around 0.0 and *s=0.4.* [4]

4.6 Physics

The physical interactions of the game are made in a discrete way that is, in every cycle the new forces to be applied to the bodies, their current positions, velocities, etc, are calculated. Every cycle is simulated to take approximately 0.01 ms.

To gather the necessary information to get the physical results some prints had to be made in the following files:

- Drive information: *rcssserver3D-root/plugin/soccer/driveeffector/driveeffector.cpp*
- Kick information: *rcssserver3D-root/plugin/soccer/kickeffector/kickeffector.cpp*
- Game information[5]: *rcssserver3D-root/plugin/soccer/soccerruleaspect/soccerruleaspect.cpp*
- Physical information: *rcssserver3D-root/lib/oxygen/physicsserver/dragcontroller.cpp* and *ode-root/ode/src/ode.cpp*

The agent movement

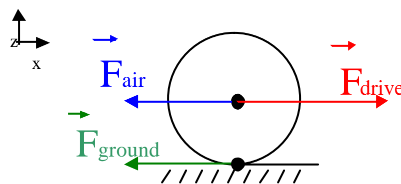


Figure 9 – Forces applied to the body of an agent when the drive effector is used.

During a regular drive, the body of an agent is under the influence of three forces: the drive force, the drag force caused by the contact with the ground and the force caused by the friction with the air.

[4] Taken from *rcssserver.rb* (section 4.7) – *setNoiseParams()* function of the kick effector.

[5] Like the current time and cycle.

The drive vector is given by the agent, and its maximum length is 100.0. To convert the drive vector to a force vector (Newtons) one must apply the following formula:

$$\vec{F} = \vec{D} * DriveFactor \quad (1)$$

The *DriveFactor* is defined in file *rcssserver3d.rb* (section 4.7) using the function *setForceFactor()* of the drive effector.

An error is also applied, so the real force can be a little bit stronger or weaker. The distribution of the error is defined in *rcssserver3D.rb* file (see section 4.7).

The force of the ground was taken from the source code and is given by:

$$\vec{F}ground = \vec{v} * DragFactor \quad (2)$$

The *DragFactor* is defined in file *rcssserver3d.rb* (section 4.7) using the function *setLinearDrag()* of the agent's drag controller.

The friction with the air is calculated internally in ode package, thus not very clear for the end-user. To examine how the FC Portugal team came near the problem see section 6.4.

The ball movement (in the ground)

The movement of the ball in the ground, when no kick or collision is detected, depends only of its initial velocity. The same is to say that the force applied in a given time to the ball depends only of its velocity at that time (Figure 10).

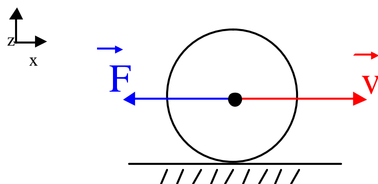


Figure 10 – Force applied to the ball when no kick or collision is preformed.

The fo rmula of the force applied to the ball was taken from the source code:

$$\vec{F} = \vec{v} * DragFactor \quad (3)$$

where F is the drag force and $DragFactor = 0.3$. [6]

[6] Taken from *rcssserver.rb* (section 4.7) – *setLinearDrag()* function of the drag effector

The ball movement (in the air)

Not studied yet.

The kick

The maximum kick power is 100 and the vertical angle varies between *[0º,50º]*. To convert the kick power into a force (in Newton) one must multiply it by a *ForceFactor = 0.7*.[7] The force is applied to the ball during 10 server cycles (0.1ms). During that interval the drag force given by equation (4) is also applied.

The kick is radial in the horizontal plane, which means that the agent must position itself according with the direction it wants to kick Figure 11.

4.7 Important Files

To run the 3D simulation server and to launch our agents some files may have to be changed (or used). The *rcsoccersim3D* file is a script that starts the server (*rcssserver3D*) and the monitor (*rcssmonitor3D-lite*). The *agentdb.xml* has some configuration information concerning the agent. Finally, the information about the game variables are set in *rcssserver3D.rb* file.

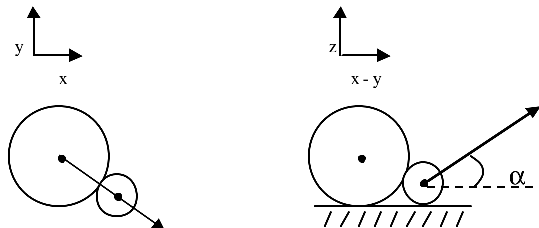


Figure 11 – The kick direction.

rcsoccersim3D

- The *rcsssoccersim3D* file, by default, is in */usr/local/bin* directory. The file is a script that starts the server and the monitor.

rcssserver3D

The *rcssserver3D* file, by default, is in */usr/local/bin* directory. The file is the binary that allows starting the server. The only argument identified that the simulator is able to receive is:

- `--agent_db_fn` followed by the `agentdb.xml` localization. It allows the server to find the `agentdb.xml` file.

[7] Values taken from *rcssserver.rb* (section 4.7) – *setForceFactor()* function of the kick effector

rcssmonitor3D

The *rcssmonitor3D* file, by default, is in */usr/local/bin* directory. The file is the binary that allows starting the regular monitor and it does not receive any argument. This monitor is very heavy and it does not provide any interaction with the user. Thus it is strongly recommended to use the *rcssmonitor3D-lite* instead.

rcssmonitor3D-lite

The *rcssmonitor3D-lite* file, by default, is in */usr/local/bin* directory. The file is the binary that allows starting the monitor. The arguments it can receive are:

- `--help` to display the list of arguments;
- `--port` to specify the port number (12001 by default);
- `--server` to specify the server host ('localhost' by default);
- `--logfile` to run the monitor as a log viewer, specifying the logfile to read (it cannot be used together with `--server` option);

This monitor is able to receive commands from the user by the monitor and the mouse. The possible commands are:

- Rotate the camera – moving the mouse with the left button pressed or using the arrow keys (, , and);
- Move the camera – using the keys:
    - W – up;
    - S – down;
    - A – left;
    - D – right;
- Change the camera to focus the ball or to focus the field– using 'C' key;
- Turn the radar on / off – using '2' key;
- Start the game – using 'K' key;
- Drop the ball – using 'B' key;
- Quit the monitor – using 'Q' key;

agentdb.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<agentdb
  xmlns:adb="http://spades-sim.sourceforge.net/agentdbxml.html"
  xmlns="http://spades-sim.sourceforge.net/agentdbxml.html"
  adb:version="0.91"
>

<!--  This file specifies the different agent types and how to execute the
agents -->
     <agent_type_external name="fcportugal">
       <inputfd>3</inputfd>
       <outputfd>4</outputfd>
       <timer>default</timer>
       <exec_line>/usr/local/bin/agenttest</exec_line>
     </agent_type_external>
</agentdb>
```

The *agentdb.xml*, by default, is in */usr/local/share/rcssserver3D/* directory. This is a XML file with some configuration data concerning the agent. In the file one may have more than one agent type. The *agent_type_external* property allows starting an agent as an external process, which will be the case of FC Portugal agents. The basic properties that must be specified for external agents are:

- `inputfd` – defines the file descriptor of the pipe where the information received by the user is read. Its value must be above 2, since file descriptors 0, 1, 2 are standard input, standard output and standard error respectively.
- `outputfd` - defines the file descriptor of the pipe where the information sent by the user is written. It must obey the same constraint above.
- `timer` – defines the timer which will be used to calculate the processing time of the agent. Jiffies or perfctr (usually by default) may be used.
- `working_dir` – defines the working directory of the agent.
- `exec_line` – gives the full command line to start the agent.

More information about *agentdb.xml* can be found in [1].

rcssserver3D.rb

The *rcssserver3D.rb* file, by default, is in *~/.rcssserver3d/* directory. The file is the configuration file concerning the server variables like objects' mass and size, field and goals dimensions, the game length, etc, or to set the effectors' and perceptors' properties, like the noise that will be added. It also can be used to add variants of the preceptors built, like to add a perfect vision preceptor which indicates the positions of the objects in the field without errors.

5 Start agent

5.1 Communication

The communication between the server and the agents is achieved using pipes. The agent code needed to communicate with the server is very simple. One does not even have to implement the pipes' opening methods.

To send messages to the server:

Comment [HGM1]: de onde eh que vem as variaveis msg_data e buffer e qual o tamanho maximo de cada msg

```
void putOutput(const char* out)
{
   strcpy(msg_data, out);
   unsigned int len = strlen(out);
   unsigned int netlen = htonl(len);
   memcpy(buffer,&netlen,sizeof(netlen));
   write(4, buffer, len + sizeof(netlen));
}
```

To receive information from the server:

```
Int getInput()
{
   ssize_t sz = read(3, buffer, sizeof(buffer));

   if ((unsigned int)sz < sizeof(unsigned int)){
      return 0;
   }

   unsigned int len;
   memcpy(&len, buffer, sizeof(unsigned int));
   len = ntohl(len);

   // zero terminate received data
   buffer[sizeof(unsigned int) + len] = 0;

   return len;
}
```

5.2 Sketch of the agent

The agent sketch consists in an infinite cycle in which it receives messages, processes the messages (“think”) and replies with actions.

```
Do forever

   msg = ReadMessageFromPipe();

   Case msg[0]:

      'D': ProcessInitMessage();
           Break;

      'S': ProcessSensation();
           Think();
           Act();
           Break;

   End Case;

End Do Forever;
```

6 FCPortugal Agent

6.1 Organization

The FC Portugal Agent 3D (FCPAgent) is composed by 6 main packages:

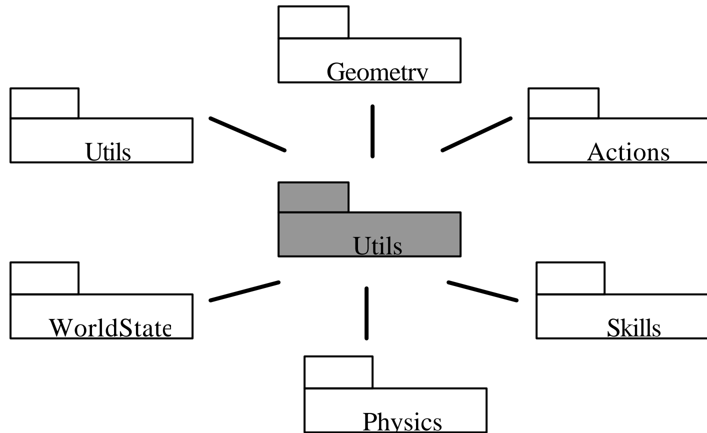


Figure 12 – Packages used by FCPAgent.

World state

Localization: FCPagentRoot/FCPAgent.cpp

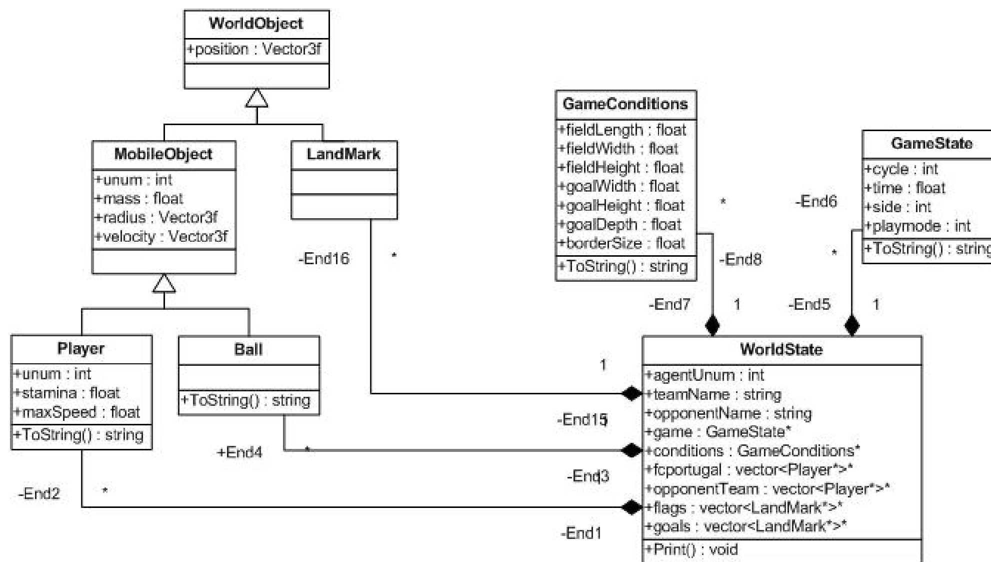


Figure 13 – The WorldState Architecture

The world state package is probably the most complex one. It has all the information that the FCPAgent needs to decide which action it should take. There are three kinds of information that the WorldState needs: information about the objects (like players, landmarks and the ball), information about the conditions of the game (like field length, goal width, etc) and the state of the game (like the current play mode, the result, the time, etc).

Physics

Localization: FCPagentRoot/physics/

The physics package aims to reproduce the physical interactions between the bodies in the world as accurate as possible in the way the server does. At the present one can estimate the velocity and the acceleration of an object, the current forces applied in a given body, and the breaking distance and time of an agent body applying a given force.

Geometry

Localization: FCPagentRoot/geometry/

On the geometry package, two classes are implemented - the Vector3f and Vector2f. Each of them provides methods to manipulate and to produce calculations with 3D vectors and 2D vectors respectively. It is also used a class named Vector and a geometry package that were included on the source code of the FC Portugal 2D agent.

Skills

Localization: FCPagentRoot/skills/

The skills are the low-level actions that an agent is able to perform. Kicking the ball, moving their body, intercepting the ball, or dribbling are samples of agent's skills. These are also the ones implemented at the moment by FC Portugal team. The scheme of the skills architecture is in Figure 14.

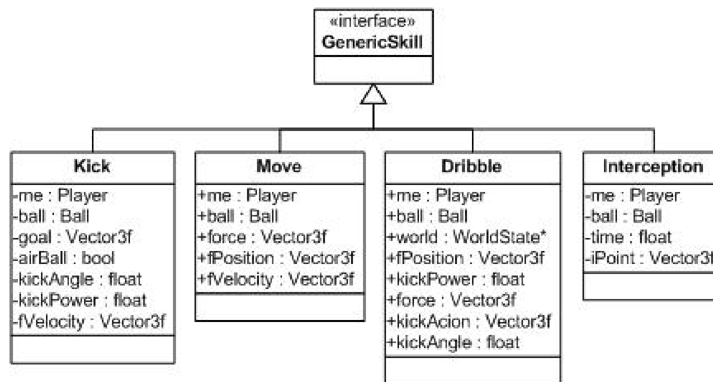


Figure 14 – Skills architecture.

Every skill implements the *GenericSkill* interface. When a skill is initialized it immediately computes the necessary calculations to execute itself. However, the initialization does not execute the skill. Every skill has a method named *Execute()* that allows its execution.

Actions

Localization: FCPagentRoot/actions/

An action is a group of skills that, together, produce higher lever behaviours. Sample of actions may be: passing, shooting, making forwards, etc. The actions in themselves are not yet implemented in the FC Portugal team. However, the architecture of the FCPAgent already supports the implementation of passes, shoots and forwards.

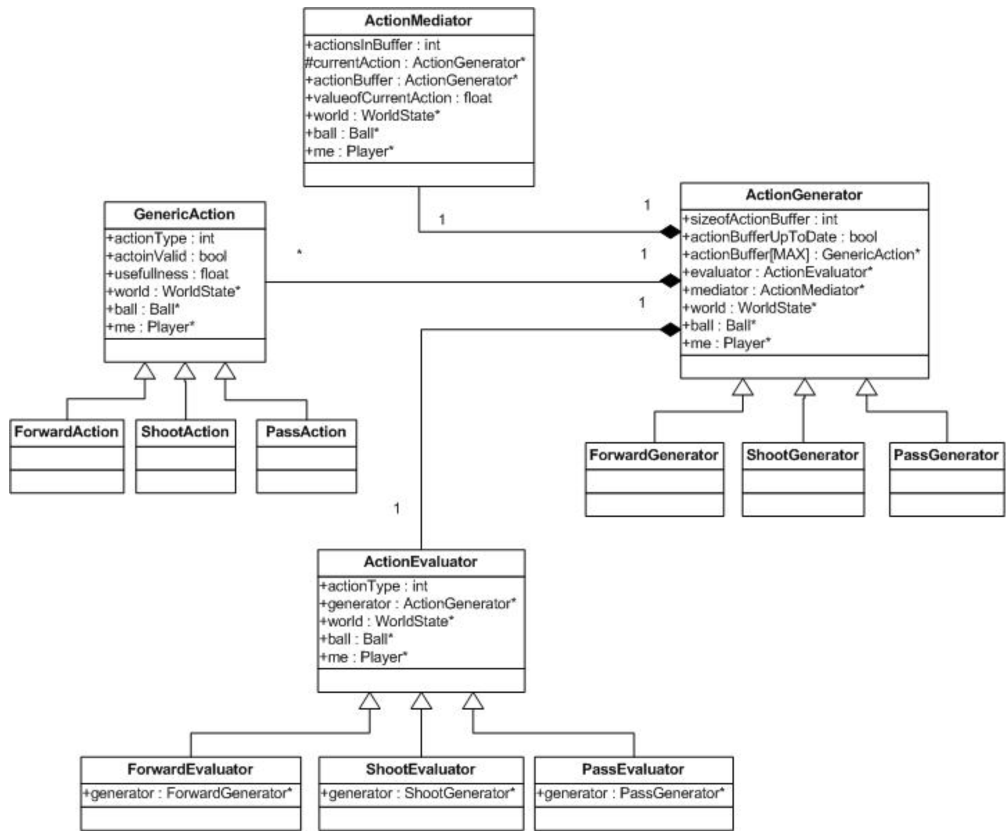


Figure 15 – Actions Architecture.

In order to produce an action 4 main classes are involved: a mediator, an evaluator, a generator and the action itself. The generator (*ActionGenerator*) is the class that allows the creation of potential actions that are able be preformed. There are 3 classes that extend the *ActionGenerator*, one per type of action– pass, shoot and forward. Each class is able to return a set of actions of its type to be considered for future evaluation. The actions returned by each generator have their one properties according with its type and all of them extend the *GenericAction* class. The evaluation of the actions created is done by the evaluator (*ActionEvaluator*). This is a class that enables the agent to estimate the usefulness of every action generated. The evaluator has also 3 classes (one per type of action) that extend it; each

of them has its one evaluation components that allow them to estimate the usefulness of a given action of its type.

To join everything together FCPAgent has a mediator (*ActionMediator*) that is able to call the functions necessary to generate and evaluate every type of action and to decide which action will be preformed.

Utils

Localization: FCPagentRoot/utils/

The package *Utils* was made to contain classes that do not have a direct relevance on the agent behaviour but help to make some tasks easier. Samples of the operations of those classes are the creation of log files, communication with the simulator, a message parser and a message composer to send the actions to the simulator.

6.2 Scketch (behaviour)

Localization: FCPagentRoot/FCPAgent.cpp

```
Case (Playmode) equals
{
   BeforeKickOff:
      MoveAccordingSBSP();

   FCPortugalKickOff:
   {
      If (nyNumber == 9)
         RunToTheBallAndKickIt();
      Else
         MoveAccordingSBSP();
   }

   OpponentKickOff or OpponentKickIn or
   OpponentCornerKick or OpponentGoalKick:
   {
      MoveAccordingSBSP();
   }

   FCPortugalKickIn or FCPortugalCornerKick or
   FCPortugalGoalKick or PlayOn:
   {
      If (MyMoveToTheBallAccordingSBSP())
         RunToTheBallAndKickIt();
      Else
         MoveAccordingSBSP();
   }
}
```

Algorithm 1 – Agent's sketch.

The behaviour of the agent is very tied to SBSP (Situation Based Strategic Positioning) (Reis and Lau 2001). The SBSP, was used by FC Portugal team on the 2D simulator) and consists in assigning the strategic position on the field to each agent given the position of the ball and the current situation. The player that runs to the ball is the one that has the closest distance

from its strategic position to the ball. Each agent is differentiated by its number (Reis, Lau and Oliveira 2001).

The general behaviour of the agent is still very incomplete. Before the kick-off every agent places itself on the SBSP position. Every time the ball is assigned, by some reason (a corner or goal kick or whatever), to the opponent team the FC Portugal team places itself according with the SBSP algorithm. On the opposite situation (the ball assigned to the FC Portugal team) the team behaves as during the game – the player with the best interception time tries to catch the ball and to kick it, while the other players move according with the SBSP. The position to where the ball is kicked depends on where the ball is situated in the field. If the ball is near FC Portugal goal then the players try to kick it to the sides, in order to do not give it to the opponents, otherwise FC Portugal agents try to shoot in the opponent's goal direction.

```
RunToTheBallAndKickIt()
{
   If (ballPosition.x < -30.0)     // the ball is in fcportugal area
   {
      // kick the ball to the sides
      If (ballPosition.y > 0.0)
         kickGoal = vector( -10.0, 30.0 );
      Else
         kickGoal = vector( -10.0, -30.0 );
   }
   Else
      kickGoal = goalPosition;  // kick to the frame


   If (PlayerInPositionToKick( kickGoal ))
      Kick( kickGoal );
   Else
      MoveToPositionToKick( kickGoal ));
}
```

Algorithm 2 . Behavior of the agent has the ball or is meant to run to catch it.

6.3 Localization system

Localization: FCPagentRoot/WorldState.cpp

The agent gets the objects position by the vision preceptor, which gives the relative position of all objects in the world in polar coordinates. The absolute position of the landmarks is set at the time the agent receives the field dimensions and the goals position. This information is usually sent by the server sent in the second or third sensation.

The localization algorithm is quite straightforward. The agent starts to seek the closer landmark. If that landmark (*closerLandmarkRelativePos*) is closer than 20.0m the agent determines its position by the absolute position of that landmark and its relative position to the agent. If the landmark is farther than 20.0m the agent combines, using a simple average, the position of the closer landmark with the position of the second closer one to determine its position.

```
closerLandmark = GetCloserLandmark();
point1 = closerLandmark.pos - ToCartesian(closerLandmarkRelativePos);

If (myDistanceTo(closerLandmarkRelativePos ) > 20.0)
{
  secondCloser = GetSecondCloserLandmark();
  point2 = secondCloser.pos - ToCartesian(secondCloserLandmarkRelativePos);
  point1 = (point1 + point2) / 2;
}

myPosition = point1;
```

Algorithm 3 – Localization algorithm.

During the experiments made the maximum error of the algorithm was around 20.0cm and its frequency was very low.

6.4 Physics

Localization: FCPagentRoot/physics/PhysicalModel.cpp

Agent movement

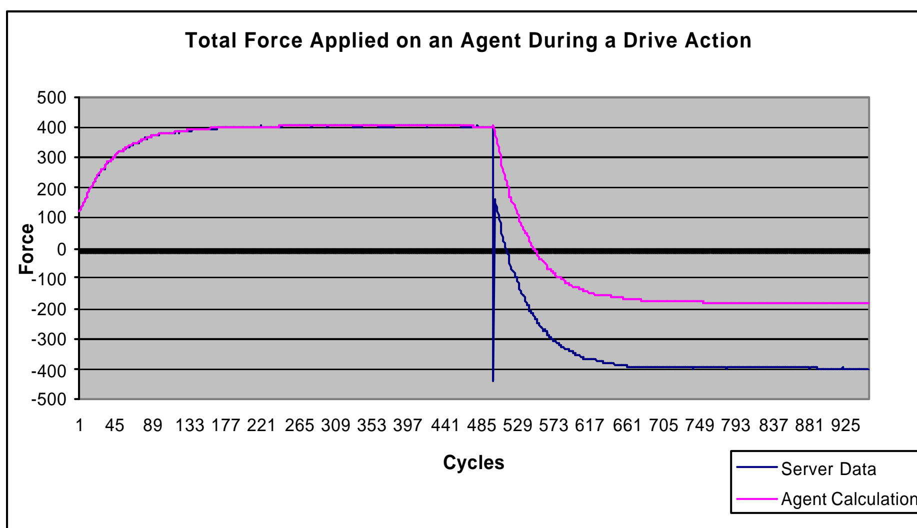


Figure 16 – Graphic with the forces applied in the agents body during a drive action.

To calculate the force in the agent caused by the friction with the air quadratic regression was used. It is given by:

$$\vec{F}air = A * \vec{v}^2 + B * \vec{v} + C \qquad (4)$$

where A = -0.840906, B = 179.955348 and C = 112.347136.

The graphic in Figure 16 shows the performance of formula used during a two step movement – acceleration using maximum force in the positive direction of the x axis followed by an acceleration also using maximum force in the opposite direction. One can see that in the first part of the movement (accelerating on the direction of the x axis) the approximation used is very good since the agent calculation is very near the server data. However, the second part, immediately after the agent starts to brake, one can see that there is a big difference between the server data and the agent's approximation, meaning that the agent's calculation is clearly not good enough.

The tests made during the programming only used acceleration on the positive direction of the x axis, which was a big mistake. To improve the formula probably another type of regression must be used. Another solution could pass through dividing the problem in two different problems – accelerating in the positive direction and accelerating in negative direction – but that may not solve the problem of braking.

Ball movement (in the ground)

To simulate the ball movement in the ground the following calculations were made. From the force equation,

$$F = ma \qquad (5)$$

and since,

$$a = \frac{\partial v}{\partial t} \qquad (6)$$

by replacing (3) and (6) in (5), one gets

$$\frac{1}{m^2}\partial t = \frac{1}{Kv}\partial v \qquad (7)$$

which simplifying results in

$$v = e^{-\frac{K}{m^2}t} \times A \qquad (8)$$

where $A = v0$ (the initial velocity).

Making these calculations, one is able to determine the velocity of the ball after a given amount of time, as well as the opposite situation – determine the time to reach a given

velocity. However, the distance made by the ball in a given time, as well as the time needed to make a given distance, may also be necessary. Thus, to reach the formula of the distance as function of time one can start from,

$$v = \frac{\partial x}{\partial t} \qquad (9)$$

Replacing (9) in (8), one gets

$$\frac{\partial v}{\partial t} = e^{-\frac{K}{m^2}t} \times v0 \qquad (10)$$

which, simplifying, results in

$$x = -2e^{-\frac{K}{m^2}t} \times v0 + D \qquad (11)$$

D can be found by calculating the formula for $t = 0$,

$$D = 2 \times v0 + x0 \qquad (12)$$

where *x0* is the position at $t = 0$.

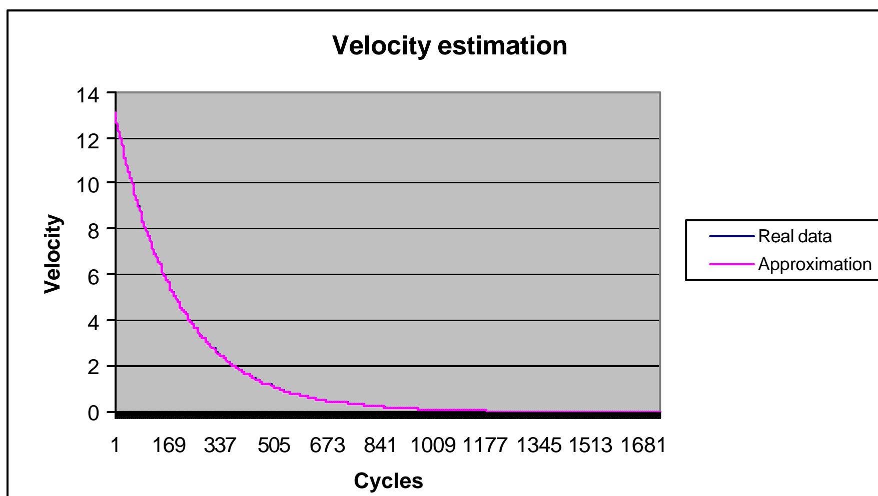


Figure 17 – Graph showing the velocity difference between the server values and the approxi mation values given by the formula used.

Finally, replacing (12) in (11),

$$x = -2e^{-\frac{K}{m^2}t} \times v0 + 2 \times v0 + x0 \qquad (13)$$

One has the distance as a function of the time.

Using the information collected from the server one can see that the result obtained is very close to the simulator's calculations (Figure 17). One cannot either distinguish between the real values and the approximation made.

Velocity calculation

The velocity calculation uses two different kinds of computation – one when the agent is faraway from the ball and another when the agent is closer the ball.

The first one uses the three last positions received by the agent and assumes that the acceleration is constant throughout the interval (which is an approximation, since in the simulator it is not quite so). So assuming p_0, p_1 and p_2 as being at time t_{-2}, t_{-1} and t_0 respectively, from the movement equations one has,

$$\begin{cases} p_1 = p_0 + v_0 t + \frac{1}{2} a t^2 \\ p_2 = p_1 + v_0 . t + \frac{1}{2} a t^2 \end{cases} \qquad (14)$$

Assuming the time difference between t_{-2} and t_{-1} to be the same as between t_{-1} and t_0, one can have,

$$\begin{cases} p_1 = p_0 + v_0 t + \frac{1}{2} a t^2 \\ p_2 = p_0 + v_0 .2t + \frac{1}{2} a.(2t)^2 \end{cases} \qquad (15)$$

which, simplifying,

$$\begin{cases} p_1 = p_0 + v_0 t + \frac{1}{2} a t^2 \\ p_2 = p_0 + 2v_0 t + 2at^2 \end{cases} \qquad (16)$$

Solving these equations to get v_0 and a,

$$\begin{cases} v_0 = \dfrac{4p_1 - 3p_0 - p_2}{2t} \\ a = \dfrac{p_2 - p_0 - 2v_0 t}{2t^2} \end{cases} \qquad (17)$$

Finally, by solving the velocity equation[8]

$$v = v_0 + 2at \quad (18)$$

one has the current velocity of the agent.

The second method uses the movement of the agent relative to the movement of the ball.

```
SetVelocity(body, pos0, time)
{
   pos1 = body.previousPos;
   pos2 = body.currentPos;

   v0 = (4 * pos1 - 3 * pos0 - pos2) / (2 * time);   // initial velocity
   a = (pos2 - pos0 - (v0 * time * 2)) / (2 * time * time) ;   // acceleration
   v = v0 + a * time * 2;   // current velocity

   if (myDistanceTo(ball) < 1.0)
   {
      // calculation based on the movement of agent relative
      // to the movement of the ball

      v = ((body.position - ball.position)-(body.prevPos - ball.prevPos))/time
   }
}
```

Algorithm 4 – Set velocity method.

Some experiments were made to discover the performance of this scheme. The agent was programmed to run to the ball and to kick it in different directions, forcing it to get near and farther the ball and consequently forcing it to use both velocity calculation methods. By looking to the first to graphics (Figure 18 and Figure 19) one may be tempted to say that the results were very good since most of the times the agent's calculation meets the server data. However one cannot forget that most of the times the agent is moving using almost constant acceleration (like when it is running at maximum speed). The problem gets bigger when a sudden change on the acceleration is preformed. If one takes a closer look around the t = 29s (Figure 20) and t = 45s (Figure 21) one can see that the calculations are not so accurate any more. The agents perform a set of moving forward / brake that cause sudden changes on the acceleration.

[8] The original formula do not multiply the *at* component per *2*. However, in this case since v_0 was calculated for time t_{-2} and the velocity is calculated for t_0, the time between equals *2t*.

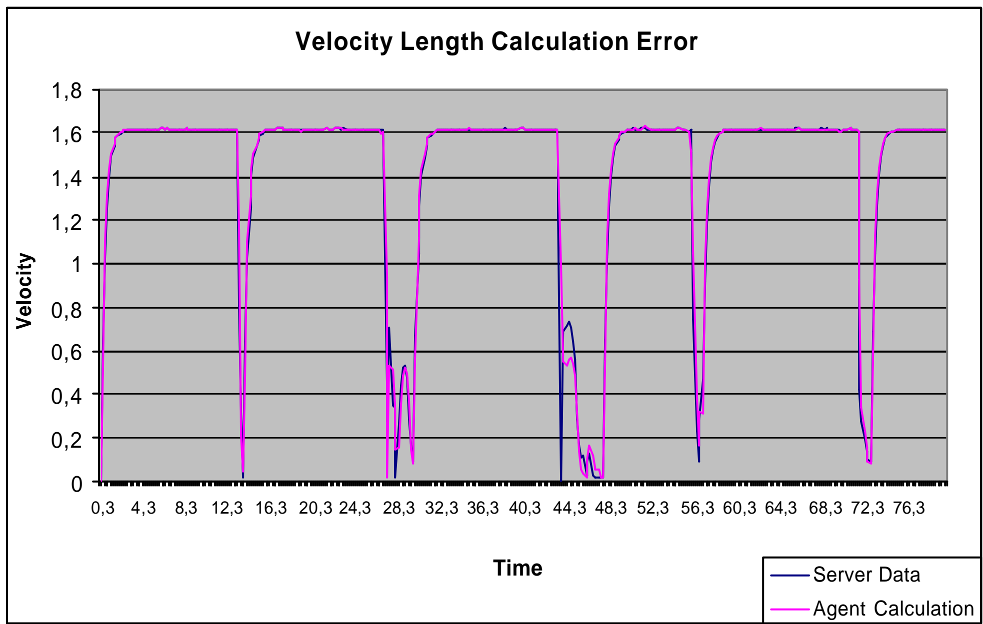


Figure 18 – Error on the calculation of the length of the velocity vector.

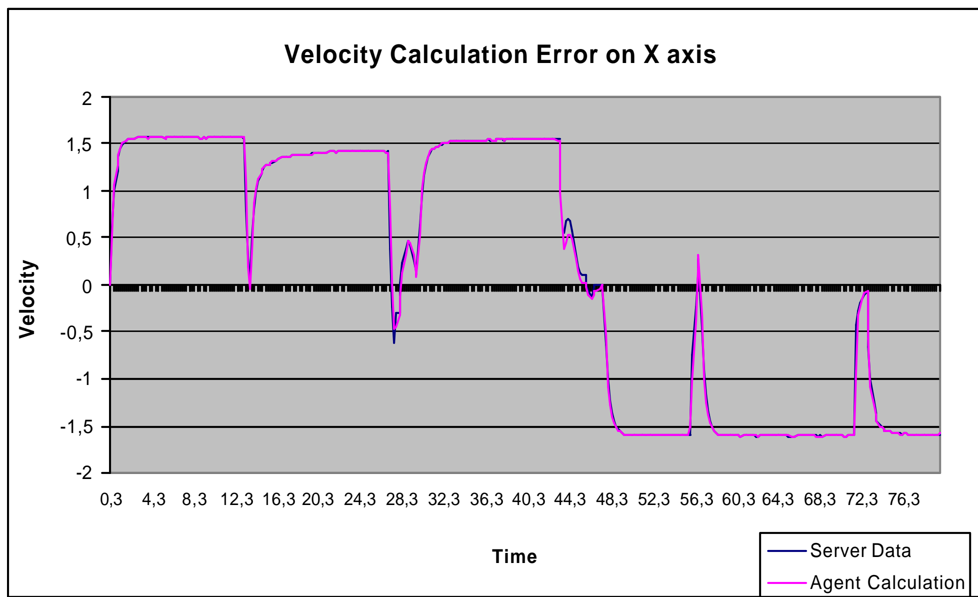


Figure 19 – Error on the calculation of the x component of the velocity

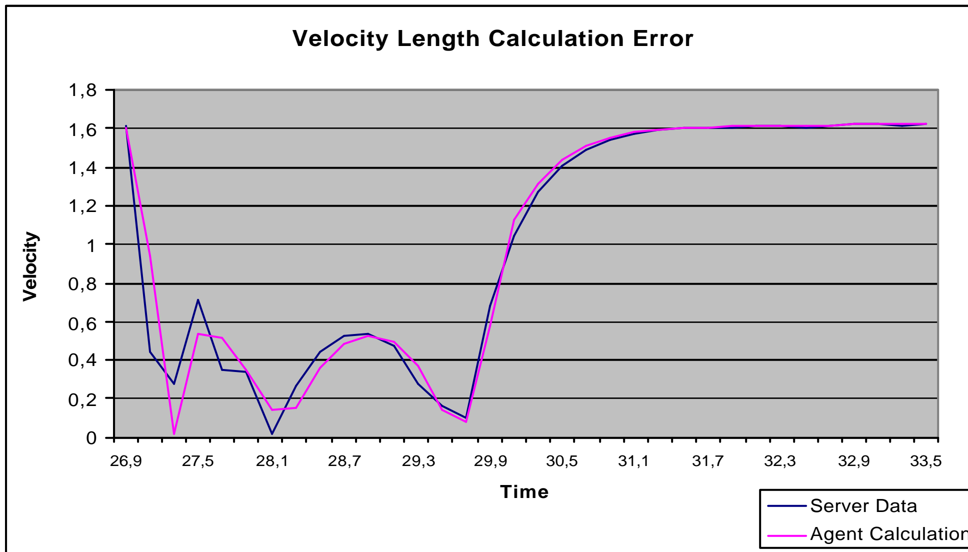


Figure 20 – Error on the calculation of the length of the velocity vector between time 26.9s and 33.5s.

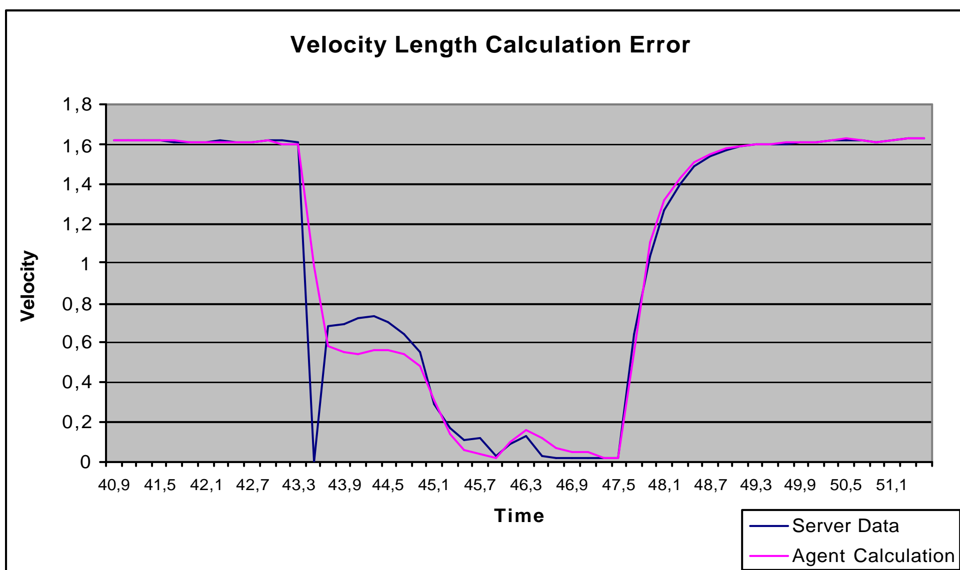


Figure 21 – Error on the calculation of the length of the velocity vector between time 26.9s and 33.5s.

6.5 Skills

Move

Localization: FCPagentRoot/skills/move/Move.cpp

The move skill takes three arguments: a pointer to the world object, a vector containing the point to where the agent wants to move and the velocity that it wants to arrive there. To calculate the force that should be applied at a given moment in order to move the agent stars by getting the braking distance to reach a given velocity, given its own current position and velocity. Then it can estimate the distance that it must be done by accelerating (*distanceToPoint – brakeDistance*). If that distance is made in more than forty cycles it accelerates, else it brakes[9].

```
direction = GetMoveDirection();

// gets the braking distance to reach velocity V
GetBreakingDistance(myPosition, myVelocity, &brakeDistance);

power = GetPowerToBeApplied(); // depends on the distance to the end point

distToBrakePoint = distanceTo

cycles = GetNumCyclesToMakeDistanceDApplingForceF(dist, power);

if (time < 40)
   force = direction * power;      // accelerate
else
   force = direction * (-power);   // break
```

Algorithm 5 – Move algorithm.

Kick

Localization: FCPagentRoot/skills/kick/Kick.cpp

As already stated a kick action takes two arguments – the kick power and the kick angle. The FC Portugal kick skill receives four arguments: a pointer to the world object, a vector giving the point to where the ball should be kicked, a boolean informing if maximum power should be used and another boolean informing if the ball should be kicked by the air.

The power that an agent applies when kicking, is directly proportional to the distance to the final point of the kick plus an extra force. The extra force varies depending whether the agent wants to kick the ball by air or not.

[9] 40 cycles = 20 cycles to start the action + 20 cycles to execute the action. The position where the agent will be at the time the action is executed must be estimated by the agent.

```
If (airBall) {
   kickAngle = 30.0;
   extraForce = 15.0;
}Else{
   kickAngle = 0.0;
   extraForce = 5.0;
}

If ( (distanceToGoal > 25.0) or (maxPower) )
   kickPower = 100.0;       // maximum power
Else
   kickPower = distanceToGoal * 4 + extraForce;
```

Algorithm 6 – Kick Algorithm.

Dribble

Localization: FCPagentRoot/skills/dribble/Dribble.cpp

Dribbling is the skill that allows an agent to run with the ball near its body. This is achieved by giving small kicks to the ball and running in order to catch it again. The starts by moving the agent to a position that enables it to kick the ball in the goal point (*finalPosition*) direction. Arriving there the agent tries to kick the ball as farther as possible (*maxDistance*). If there is an opponent agent that is able to still the ball, the FCPagent starts to reduce the kick distance successively until none of the opponent players can catch the ball first.

```
If (PlayerInPositionToKick( FinalPosition )){

   maxDistance = GetDistanceToFinalPosition();

   if (maxDistance > 7.0)
      maxDistance = 7.0;

   While ( maxDistance > 0.0 )
   {
      If (OpponentThatCatchesTheBallFirstExists())
         maxDistance -= 1;
      Else
         Break;
   }

   kickPoint = GetKickPoint(finalPosition, distance);
   Kick( kickPoint );

}Else
   MoveToPositionToKick( finalPosition );
```

Algorithm 7 – Dribble algorithm.

Interception

Localization: FCPagentRoot/interception/Interception.cpp

```
If (ballDistance < 0.5) // too close the ball
{
   interceptionPoint = ballPosition;
   interceptionTime = 0.0;
   return;
}

runningDistance = 0.0

While ( runningDistance < ballDistance )
{
   ballPosition = CalculateNextBallPosition(); // ball movement formulas
   ballDistance = GetBallDistance(ballPosition);
   runningDistance = t * playerMaxSpeed;
   time += 0.2;    // 20 server cycles
}

Time -= 0.2   // interception reached 20 cycles before
```

Algorithm 8 – Interception Algorithm.

The interception skill gives the agent the ability to catch the ball. At the moment just the quicker interception is calculated, that is, the interception that enables the agent to catch the ball in the less time possible. It receives, as arguments, a pointer to the world and the number and the team of the player to whom the interception will be calculated. The algorithm is quite simple; it calculates the position of the ball in each 20 cycles (interval between sensations), calculates the distance that the player is able to run at maximum speed and the distance from the player to the ball. The agent catches the ball when the ball distance is smaller than the distance that is able to run in a given time.

6.6 Actions

Pass

Not done yet.

Shoot

Not done yet.

Forward

Not done yet.

6.7 Future work

Geometry

To extend the relevance of the geometry package it should also be implemented a Ray class, a Line class and a Rectangle class, which could be used to produce calculations like: the distance from a point to a line, the interception point of two lines (or rays), the angle between

two lines (or rays), whether a point is in or out of a rectangle area, etc. These classes should be implemented taking in attention the final goal, that is, the calculations that each agent has to perform. This means that, for example, if one wants to know an interception point between two trajectories, the size of the objects must also be used, opposing to calculations that reduce the agent to its centre point.

Physics

The estimation of the velocity should be improved some how, since it plays a very important role in all the components of the agent.

7 Bibliography

Riley, Patrick (2003). *SPADES for Parallel Agent Discrete Event Simulation*. http://spades-sim.sourceforge.net/

Chen, Mao *et al* (2002). *RoboCup Soccer Server*. http://sserver.sourceforge.net/

Smith, Russell (2003). *Open Dynamics Engine v0.039 User Guide*. http://opende.sourceforge.net/

RoboCup Soccer Server 3D Maintenance Group (2003). *The RoboCup 3D Soccer Simulator*. http://sserver.sourceforge.net/

Kitano, Hiroaki *et al* (1995). RoboCup: The Robot World Cup Initiative. In *Proc. of IJCAI-95 Workshop on Entertainment and AI/Alife*.

L.P. Reis, N. Lau, E.C. Oliveira (2001). Situation Based Strategic Positioning for Coordinating a Team of Homogeneous Agents. In: *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, (Markus Hannebauer, Jan Wendler, Enrico Pagello. (Ed)), LNCS 2103, pp. 175-197, Springer Verlag, Berlin.

L.P. Reis and N. Lau (2001). FC Portugal Team Description: RoboCup 2000 Simulation League Champion. In: *RoboCup-2000: Robot Soccer World Cup IV*. (Peter Stone, Tucker Balch and Gerhard Kraetzschmar. (Ed)), LNAI 2019, pp. 29-40, Springer Verlag, Berlin.