Embedded Behavioral Control of Four-legged Robots

David Herrero Pérez and Humberto Martínez Barberá
University of Murcia
Spain

1. Introduction

The problem of creating a four-legged robotics football team is a very difficult and challenging problem. Regardless of the hardware design and manufacture, there are several fields involved, like low level locomotion, perception, location, behavior development, communications, etc., which should be addressed for developing a fully functional team. In practical terms, this means that the software project to develop a robotics soccer team can be very large, which implies that verification, debugging and monitoring tools are needed and play a very important role in software development time (which uses to be a very expensive resource).

This work is focused on the architecture and behavioral programming model we use to develop a team in the Sony Four-Legged League, which is one of the official leagues of the RoboCup. All the code, examples, and tools we present in this work have been developed for the **TeamChaos** team[1], which has participated in the 2004, 2005 and 2006 editions of the RoboCup and several international competitions, and is a follow up of the former **TeamSweden** team, which has participated in the previous editions (1999 to 2003). In this league, all the teams must use the same physical platform, in particular the commercial four-legged robot AIBO developed by Sony. The most recent model, *AIBO ERS-7*, integrates one CPU (64-bits RISC Processor 576MHz), audio interfaces (speaker and stereo microphones), switch sensors, two infrared distance sensors and a CCD camera (350K pixels) as exteroceptive sensors, accelerometers as internal sensors and wireless LAN for communications.

In the 2007 RoboCup Edition, the teams of four-legged league consist on four robots, which have to operate fully autonomously, i.e., there is not external control, neither by humans nor by computers, thus all the processing must be done on board and for practical reasons it has to be performed in real time, which prevents us from using time consuming algorithms. Because of the fact that the hardware platform is standard, we can consider this as a software only league. The field is also standard; there are many color coded objects in the

1 http://www.teamchaos.es

field to make easier the sensing, like ball, goals and beacons, and each team wears a colored uniform. However, in a real soccer field there are not characteristic colored cues, therefore, rules of RoboCup are gradually changed year after year in order to push progress towards the final goal, *"by the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team"*. The official platform and field of this league for the RoboCup 2007 Edition is shown in Fig. 1.

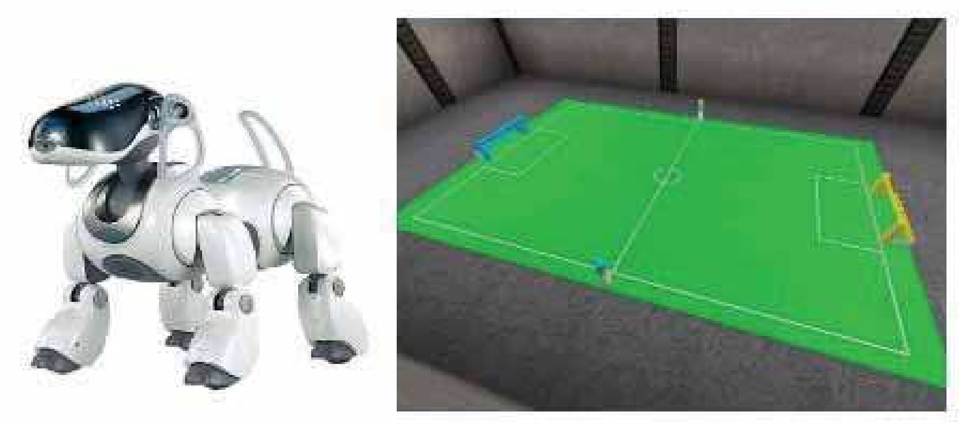

Fig. 1. Official platform (left) and field (right) of four-legged league in 2007 RoboCup Edition

This league proposes a very demanding scenario, with high uncertainty in perceptions and limited processing capabilities. In addition, having a competition implies that at certain periods the software development and tuning presents high activity peaks. These facts condition the way robots have to be programmed. Our approach to this problem is twofold: we define a software architecture into which all the different modules plug, and we use a programming language that can drastically reduce development time for certain modules, in particular all behavior related modules.

We follow the ThinkingCap architecture (Saffiotti et al., 1995), a two-layer architecture which clearly reflects a cognitive separation of modules. From the conceptual point of view, modules are arranged by the nature of their processing tasks. From a software point of view, the interfaces are clear and well defined, so that replacing or improving modules is not a demanding task. Our current instance of the architecture makes extensive use of the behavioral paradigm, and for implementing those behaviors we have opted for the LUA language (Ierusalimschy et al., 1996). LUA is a simple yet powerful embedded language with a quite portable interpreter. We have integrated LUA in the architecture and have developed a set of tools for the on line edition, monitoring and debugging of control programs. This allows us to develop and modify behaviors while testing robots on the playing field at runtime. Moreover, the behaviors can be tested and verified before the execution on the real robot using those software tools, which dramatically reduces the severity and duration of downtime during development time. In order to simplify and reuse behaviors, we organize them in two types depending on the complexity and functionality:

low-level behaviors, which perform specific actions (go to ball, look for ball, kick ball, etc), mostly reactive, and high-level behaviors, which perform high level tasks (defend, attack, pass, penalty shootout, etc), mostly deliberative. These usually imply the use of some form of state, and we have opted for the Hierarchical Finite State Machine (HFSM) paradigm (Hugel et al, 2005) to design them. We have developed a visual editor for HFSMs (based on Hugel's code), which automatically generates LUA code directly executable by our architecture.

This chapter is organized as follows. Section 2 describes the control architecture. Section 3 describes the LUA language and how it has been incorporated into the architecture. Section 4 and 5 describe low-level and high-level behaviors respectively. Section 6 presents some conclusions and future work.

2. Control Architecture

2.1 The ThinkingCap Model

The architecture used by each robot is an instance of the ThinkingCap architecture (Saffiotti et al., 1995). Fig. 2 shows the main elements of this layered architecture; the lower layer provides the interface to the actual hardware, the middle layer maintains a consistent representation of the environment around the robot and provides the reactive robot control, the higher layer maintains the representation of the objects in a world frame and performs decisions considering the global information, and the communication layer provides the interface to share information with the other members of the team.

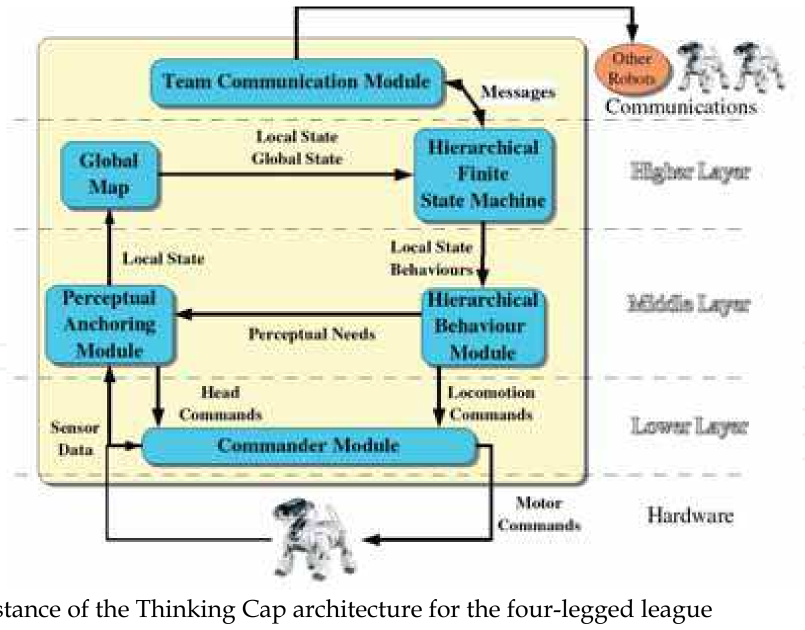

Fig. 2. Instance of the Thinking Cap architecture for the four-legged league

The **lower layer** provides an abstract interface to the sensori-motor functionalities of the robot (Commander Module – CM). This module accepts abstract commands from the upper layer, and implements them in terms of actual motion of the robot effectors. In particular, it receives set-points for the desired linear, lateral and angular velocity, and translates them to an appropriate walking style by controlling the individual leg joints.

The **middle layer** maintains a consistent representation of the space around the robot (Perceptual Anchoring Module - PAM), and implements a set of robust tactical behaviors (Hierarchical Behavior Module – HBM). The PAM acts as a short-term memory of the location of the objects around the robot: at every moment, this module contains an estimate of the position of these objects based on a combination of current and past observations with self-motion information. The PAM module is also in charge of camera control, by selecting the fixation point according to the current perceptual needs (Saffiotti & LeBlanc, 2000). The HBM implements a set of navigation and ball control behaviors.

The **higher layer** maintains a global representation of the field (Global Map - GM) and makes real-time strategic decisions based on the current game state, situation assessment and role selection (Hierarchical Finite State Machine - HFSM). Self-localization in the GM is based on fuzzy logic (Buschka et al., 2000), (Herrero-Pérez et al., 2004). The HFSM implements a behavior selection scheme based on finite state machines (Hugel et al, 2005). In addition, global ball sharing is also based on fuzzy logic (Cánovas et al, 2004).

Radio **communication** is used to exchange position and coordination information with other robots (Team Communication Module - TCM) using customs protocols over UDP/IP.

Intercommunication between the different modules is implemented by data structures interchange. The two more important data structures are those representing the world state, either in a local or global frame. The information stored in these structures can be related to either static objects (nets, landmarks) or dynamic objects (ball, teammates, opponents). The local state represents the objects that have been recently perceived by the robot camera in a robot centric frame. The data structure that represents the local state is called *Local Perceptual Space* or LPS (Saffiotti et al., 1995), which consists on an array of Local Perception Objects or LPO. Each one of these LPOs includes polar positioning information (ρ and θ) and an anchoring value, which somehow represents the reliability of that precise perception and decreases over time (Saffiotti & LeBlanc, 2000). At each control cycle, all the objects that have not been re-perceived are rotated and translated according to the odometry estimation. The global state represents all the objects that can be in the environment in a world frame, in our case the field viewed from the goalkeeper position. The data structure that represents the global state is called *Global State* or GS, whose contents is the result of fusing information from different sources (time aggregation of own camera perceptions and/or other robots global states). The most important information for soccer game play is that related to self-positioning (implemented by filtering the different camera perceptions and odometry estimates) and ball positioning (implemented by filtering the local position of the ball from the different team robots and their absolute position estimations). The first one is important for zonal behaviors (i.e. keeping the goalkeeper in its own area), while the second is important when the robot does not sees the ball or it is too far to get good distance estimation.

From the control point of view, the behaviors use the information contained in the LPS (typically low-level behaviors of the HBM) and the GS (typically high-level behaviors of the HFSM) to perform reactive or strategic decisions, which finally translate into movement commands (forward, lateral and angular velocities or kicking actions) or perceptual commands (object necessities or needs). While the former are more or less standard in any soccer-playing robot (and certainly in any four legged robot), the later are specific to this instance of the ThinkingCap architecture (Saffiotti & LeBlanc, 2000). The needs are stored in an array, and they represent the priority to actively look for an object in the environment. The PAM uses this array to determine at each cycle which object it should look for, typically the one with the highest priority. In case that two or more objects share the highest priority, the one selected is the one less recently seen. In any case, when looking for an object, if any other is perceived it is incorporated into the LPS. In addition to this, behaviors can also share information with teammates (ball booking, role information, etc).

2.2 Integration with OPEN-R

The API for programming and debugging code for the AIBO platform, Sony OPEN-R SDK, is merely an interface to develop software for the Aperios OS, which is the real-time operating system used by the entertainment robots of the company. This API discloses the specifications of the interface between the *system layer* and the *application layer*. The software developed using these specifications is object-oriented and modular, where each module is an OPEN-R object. In practice, OPEN-R objects are threads running concurrently with many inter-threads connections which invoke methods of the OPEN-R objects, i.e., the way to manage the concurrency is event-oriented programming. The software to control the robots consists on multiple objects with various functionalities running concurrently and communicating each other via inter-object communication. The programming language supported by this API is C++, including its functionalities.

The Thinking Cap architecture has been programmed using OPEN-R. The natural way to implement the different modules of the architecture is by using one OPEN-R object for each module of the architecture, being the connections between objects implemented by the event-oriented inter-object communication of Sony OPEN-R SDK. Moreover, this implementation provides effective modularization as well as clean interfaces, making it easy to develop different parts of it. Furthermore, it allows the execution of each module in a computer, using RP-OPEN-R (Remote Processing OPEN-R) which is a tool for compiling and executing OPEN-R objects on x86-based CPUs or AIBO robots. For instance, the low level modules can be executed on-board and the high level modules can be executed off-board, where some debugging tools are available. However, due to the real-time constraints, this distributed implementation generates serious synchronization problems, which cause delays in decisions and the robots cannot react fast enough to dynamic changes in the environment.

Because of the reasons above mentioned, we have favored two particular implementations of the Thinking Cap architecture using OPEN-R objects, which we call distributed and monolithic respectively. The **distributed** version is composed of three OPEN-R objects: low level control (ORLRobot), high level control (ORHRobot) and communications (ORTcm). The low level object contains the functionalities of the robot CM and the PAM module in charge of the head or camera, which allows executing this module independently for debugging. The high level object contains all the behavioral part of the system, the HBM and the HFSM, and the GM, the global representation of the field, which allows executing this module off-board receiving or simulating the data from the low level object. The communications object implements an interface to communicate the robots using customs protocols over UDP/IP. The trade-off between synchronization problems and modularity for debugging is met using this distributed version of only three OPEN-R objects, instead of one object for each module. The **monolithic** version has two OPEN-R objects only: robot control (ORRobot) and communications (ORTcm) objects. The robot control object contains all the functionalities of the robot included in the ORLRobot and ORHRobot modules described above, while the communications object is exactly the same.

In order to maintain both implementations, the different software modules are programmed using standard C++ code, and at compilation time it is decided whether it will be a distributed or a monolithic version. Fig. 3 shows the two possible implementations of the architecture at compilation time, each thread or OPEN-R object contains many software modules of the Thinking Cap architecture. As mentioned above, the distributed implementation allows running modules independently, which facilitates drastically the debugging, and the monolithic implementation avoids the synchronization problems of the inter-object communications of OPEN-R objects, mainly due to concurrency issues.

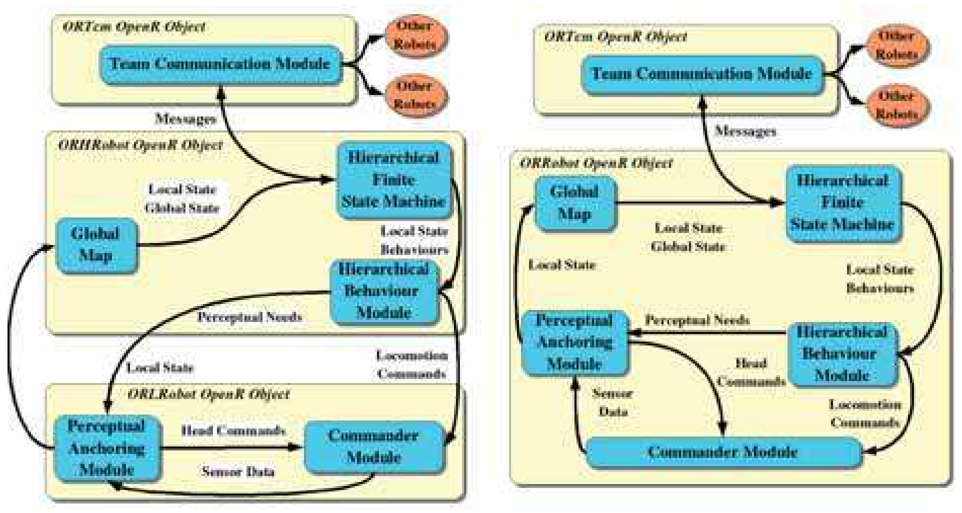

Fig. 3. Implementations of the architecture: distributed (left) or monolithic (right)

3. The LUA Interpreter

3.1 The LUA-based Development Cycle

Because the AIBO exhibits a closed and restricted programming system, the only way to incorporate new software to the robot is by way of OPEN-R objects coded in C++. The typical **on robot** development cycle is as follows: a C++ program is edited and compiled in a desktop computer using a cross compiler for the AIBO CPU. The generated binaries are then written in a memory stick (usually attached to the computer using an USB card reader). The memory stick is then inserted into the robot. Finally the robot is switched on and the program is verified or validated. There are alternatives to this process, some of them involving using FTP to send the new binaries to the robot, but it is always needed to reboot or switch on the robot, which usually takes a couple of minutes). The typical **off robot** development cycle is much simpler, because the C++ programs can be compiled to the desktop computer CPU and then executed there. Unfortunately, this procedure can only be used to check for the completeness of software modules, but not for the verification and validation of actual robot execution (despite the use of a good simulator, which are not freely available). For these situations the real robot is needed, and the debugging is quite tedious. In particular, this is more exacerbated during competitions, in which during a short time, a lot of behaviors and their parameters have to be modified and checked (and even created from scratch).

In this situation, it is close to be mandatory the use of a series of tools that can boost development and debugging time on robot. Therefore, many teams working with the AIBO platform are using different high-level languages in their behaviors designs. For example, some teams use a reduced version of Perl (Upenn team, from University of Pennsylvania) or Python (UNSW team, from University of New South of Wales) interpreters to implement their high-level behaviors and for rapid development using scripts, which are compiled into intermediate opcodes for efficient performance. Other teams have developed specifics languages to engineering the behaviors of multi-autonomous agents in complex and dynamic environments, like the *Extensible Agent Behavior Specification Language* (XABSL) (Loetzsch et al., 2006). All of these approaches have their pros and cons. XABSL is a XML based language, which without the proper editing tool is very difficult to develop. Current XABSL tools only work with GermanTeam[2] architecture and are very specific for that system. On the other hand, Perl and Python are general use languages, but because of the OPEN-R and hardware constraints, only a reduced interpreter can be used. Perl code readability and maintenance is difficult, and learning curve is quite high. Python is a much cleaner language, but its features (base libraries) are far more complex than what is needed for developing behaviors. In addition Python interpreter footprint is large compared to Perl and other options. For these reasons, we have adopted LUA[3] as the programming language of the behaviors of our architecture. LUA (Ierusalimschy et al., 1996) is a free and open-source multi-paradigm programming language, extremely compact and primarily used as a scripting language or an extension to another language, mainly C/C++. It is primarily considered an extension language although it can be categorized as extensible, interpretive, iterative, logic-based, multi-paradigm, object-oriented, reflective, or as a scripting language. The main characteristic is that it utilizes meta-mechanisms instead of implementing various features directly. This means that the core of the language is fairly restrictive because it is embedded, but it can be extended to include other desired features. Because the language does not include these extensions by default, it avoids the overhead for unused functions, streamlining the code to make it optimal for embedding within another program. Nowadays, this language is primarily used in video games because of its versatility.

In order to enable the LUA interpreter to interact with the sensory and motor routines implemented in the OPEN-R modules, the interpreter was extended so that it is able to call C functions exported by the modules, namely a C extension module for LUA. Basically, this

[2] http://www.germanteam.de

[3] http://www.lua.org

extension module is an interface to access data variables in the OPEN-R routines and activate functions that set sequence of motions, behaviors, team messages, etc. The data variables are the structures defined in the control architecture of the robotic team, which is depicted in Fig. 2. Both the HFSM and the HBM are implemented using LUA scripts with this interface, being the difference the input/output data variables and the activation functions. In the HFSM, the data variables are used to coordinate and make decisions considering the team, the global state of the game and the messages of the teammates, and the decision is the activation of a low level behavior in the HBM. In the HBM, the behaviors only depend on the local state around the robot, being the actions the activation of perceptual needs and locomotion commands, including kicks.

With the use of LUA, the **on robot** development cycle is as follows: a LUA source file is edited in the desktop computer, and then it is sent to the running robot (the development tools make some validation and verification checks, and the file is only sent if it contains no errors). From this moment on, successive calls to the edited script will use the new version. This is clearly a much nicer and helpful approach than the C++ one. In addition, although syntax check is performed, the new code can have potential errors. Because of the lack of LUA pointers, in any case, the new code cannot crash or hang the AIBO CPU, which is a quite common situation while debugging C++ code. In order to simplify code management, each behavior (be it low-level or high-level) is coded in a single LUA source code file

3.2 Integration of the LUA Interpreter

Because of LUA runs by interpreting bytecodes for a register-based virtual machine, we have performed different experiments to evaluate its computational cost because this is a critical point in real-time applications, like robotics soccer is. While interpreting LUA bytecodes is extremely fast, the integration of the LUA virtual machine with the legacy C++ code presents some difficulties. Because we use LUA source files that can be edited and replaced at any time, we do not store the bytecodes. The control cycle of our architecture is quite tight given the hardware platform, and typically both a low-level and a high-level behavior are executed every 100 ms. The time consumed by the interpreter should be minimum in order to leave as much CPU as possible for the vision process (PAM). For the integration we have tried and evaluated three different strategies:

- Each time a script is called it is loaded from the file and then interpreted. This is the simplest approach, and the very first to be implemented. While it is simple, the overhead of interpreting two files every control cycle is too high.
- The first time a script is called it is loaded and kept into memory, and then interpreted at every successive call. The advantage of this approach is that the loading time (accessing the memory stick is quite time consuming) is reduced, but when the behavior is modified the system must signal the interpreter to reload it.
- The first time a script is called it is compiled to bytecodes and the kept into memory, and then executed at every successive call. The advantage of this approach, much like the just-in-time compilers (JIT) standard to the Java world, is that both the loading time and the interpretation time are reduced. In this case it is needed to signal the interpreter when a behavior has been modified to reload and recompile it. This is by far the more

complex approach (from an implementation point of view) and the one that has finally being selected.

Fig. 4 shows a plot of the time in microseconds consumed in several calls to a high-level LUA behavior. The curve labeled as original is the simple approach, and the curves labeled as Optimization 1 and Optimization 2 are the two successive improvements. In average the three approaches consume approximately 40 ms, 17 ms and 7 ms respectively. The variance in execution time is due to the execution time of the other concurrent processes, being it larger when the task consumes more CPU. The measured time is the difference between the end and start times of the interpreter process obtained from the real time clock (RTC). As it can be seen, the third approach is by far the less time consuming (by a factor of nine with respect to the simple approach), and is the one that it is currently in use in our system.

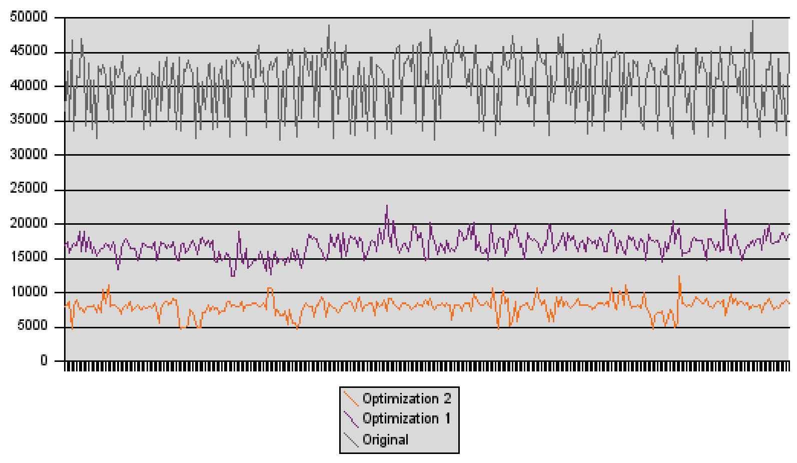

Fig. 4. Execution time (µs) for the three integration approaches of the LUA interpreter

3.3 ThinkingCap Services of the LUA Interpreter

Because behaviors need to access to information provided by other modules of the architecture, we have extended the LUA interpreter to interact with the routines programmed using OPEN-R; in particular, we have implemented a C extension module to call C functions exported by the OPEN-R modules. Basically, the library to interact with the Thinking Cap architecture, named *chaoslib*, is an interface to access data variables in the OPEN-R routines and activate functions that set sequence of motions, behaviors, team messages, etc. We will describe the most important methods of the library for a better understanding of the different examples shown below. The *chaoslib* methods can be grouped into: data access methods, control action methods, coordination methods, and persistence methods.

Data access methods allow accessing all the different information that is generated by the different modules of the architecture, mainly the LPS and the GS. The most important methods are:

- *chaos.lps_getLpo(index).* Returns a table containing the values of the LPO stored at position 'index' of the LPS. A set of convenience constants is available (BALL, NET1, NET2, etc).
- *chaos.gsGetMyPos()*. Returns a table containing the estimation of the location of the robot from the GS.
- *chaos.getBallVel()*. Returns a table containing a vector with the estimation of the velocity of the ball. The goalkeeper typically uses this to decide a defensive action.
- *chaos.getGlobalBall()*. Returns a table containing the estimation of the location of the ball from the GS.
- *chaos.getMates()*. Returns a table containing the estimation of the location of the teammate members.

Control action methods send commands to the other modules of the architecture, like locomotion commands, object necessities and kicking routines activation. The most important methods are:

- *chaos.setVlin(vlin).* Sets the desired robot's linear velocity.
- *chaos.setVrot(vrot)*. Set the desired robot's rotation velocity.
- *chaos.setVlat(vlat).* Set the desired robot's lateral velocity.
- *chaos.setKick(kickid).* Performs a kicking routine with identifier 'kickid'.
- *chaos.setNeeded(index, value).* Sets the necessity of LPO object at LPS position 'index' to a value of 'value'.
- *chaos.trackLandMarks().* Special method that sets necessities for localization specific objects.
- *chaos.setBehavior(behavior).* Executes a behavior with name 'behavior'.

Coordination methods allow the robot to interact with its teammates. These can be related to the current role of the player, implemented with a dynamic role allocation method (Agüero et al., 2006a), and with the booking of the ball, implemented with a distributed mutual exclusion method (Agüero et al., 2006b). The most important methods are:

- *chaos.getRole()*. Returns a table containing the current role of the robot.
- *chaos.haveBookedBall().* Returns a table containing a truth-value representing the ball booking state for the robot.
- *chaos.bookBall()*. Ask the other robots that we want to book the ball.
- *chaos.releaseBookedBall().* Tells the other robots that we no longer want to book the ball.

Persistence methods allow the behavior to obtain information about previous activations. This is due to the fact that consecutive activations of a behavior imply loading bytecodes into the LUA virtual machine. In this way, a behavior can share with itself some form of behavior state, and know information related to its activation. Data storage is implemented in a hash table. Because LUA does not support types, the global hash table has parameters to specify data types. Current supported types are string, integer and float. The most important methods are:

- *chaos.getBehaviorInfo()*. Returns a table containing the information about the current behavior. This information includes a behavior execution timer and a flag indicating if it is the first time the behavior is executed.
- *chaos.setGlobal(key, type, value)*. Stores the variable with name 'key' of type 'type' with the value 'value' into the global hash table.
- *chaos.getGlobal(key)*. Gets the value of the variable with name 'key' from the global hash table.
- *chaos.getCurrentState()*. Gets the current state of the game as set by the external referee.

4. Low-level Behaviors

4.1 The HBM Model and Tools

Behavior-based systems are increasingly used in many robotic applications, including mobile units, manipulators, entertainment robots and humanoids. Behavior-based systems were initially developed on mobile robots, where complex tasks were achieved by combining several elementary control modules, or *behaviors* (Brooks, 1986). In most of these systems, however, arbitration between behaviors was crisp, meaning that only one Behavior was executed at a time, resulting in jerky motion. In other systems, several behaviors are executed concurrently and their outputs are fused together, resulting in smoother motion during switching of behaviors (Cameron et al., 1993), (Saffiotti, 1997), (Saffiotti et al., 1993).

The use of behavior-based system for more complex plants than a wheeled unit needs a framework which is able to handle several DOF (Kim et al., 2001) uses fuzzy rules to control a 6 DOF arm, and (Lever et al., 1994) describes an automated mining excavator that uses concurrent behaviors. However, the complexity of these systems makes the design process very demanding: (Kim et al., 2001) uses 120 rules to perform a Pick Up task, and (Lever et al., 1994) uses a neural network for behavior arbitration, thus giving up the readability of the arbitration rules.

We follow an approach to building behavior-based systems that can be applied to control plants with many DOF. Complexity is addressed by a hierarchical decomposition (Saffiotti & Wasik, 2002) of the overall task into simple behaviors. These behaviors are encoded by small sets of fuzzy rules. In addition, fuzzy meta-rules are used to encode the way behaviors are combined together: this makes it very easy to re-configure the system for different tasks.

We define a set of **basic behaviors**, that is, behaviors that perform elementary types of actions, most of them common to all players independently of their role in the field. In our domain, these include turning toward the ball, going to the ball, or moving to a given position. These behaviors constitute the basic building blocks from which more complex types of actions are obtained by hierarchical composition. The behaviors, which are packaged in the HBM module, are defined by way of fuzzy rules (Saffiotti et al., 1993), (Saffiotti et al., 1995). The input space used by all behaviors is the local state provided by the PAM, which contains the current estimates of the position of all the objects in the field. The output space consists of the velocity set-points which are transmitted to the CMD module.

An additional control variable is used to indicate which kicks are applicable in the given situation.

Thus behaviors are coded by using fuzzy rules of the form:

```
if <predicate>            then <action>
```

where predicate is a formula in fuzzy logic that evaluates a series of properties of the local state. This can be done without the need of an analytical model of the system or an interaction matrix, which may be difficult to obtain for complex plants and tasks, as the RoboCup case is. It is also worth noting that the uncertainty in fuzzy system is taken in account as well.

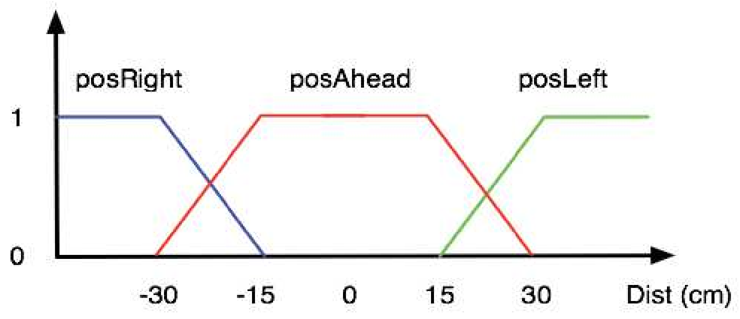

Fig. 5. Membership functions for the *posLeft*, *posAhead* and *posRight* conditions

In order to give a more concrete impression of how behaviors are implemented, we show here the *gkcutb* behavior. For sake of clarity, all the rules shown in this section are given in pseudocode and in a slightly simplified form with respect to the actual rules implemented in the robot. The rules are actually implemented using different LUA methods. The goalkeeper's *gkcutb* behavior uses three rules to control the lateral motion of the robot (action *strafe*) to locate it on the trajectory from the ball to the centre of the net. It also uses three rules to control the orientation of the robot to point the head toward the ball (action *turn*). This behavior does not use the forward motion, and thus it is always set to zero (action *go*). Finally, it controls the type of kick to be applied (action *kick*). If the ball is close enough and the robot is pointing to a safe area it tries to kick it using its both arms and the chest.

```
if posLeft                          then strafe RIGHT
if posAhead                         then strafe NONE
if posRight                         then strafe LEFT
if headedLeft                       then turn RIGHT
if headedAhead                      then turn NONE
if headedRight                      then turn LEFT
if and(freeToKick,  ballClose)      then kick FRONTKICK
always                              go STAY
```

The aim of this behavior is to keep the goalkeeper on the trajectory of the ball to the net. The rule conditions are evaluated from features of the local information, like Ball<ρ, θ> (distance and orientation to the ball), and produce a truth value between 0 and 1. Fig. 5 shows the truth value, or *membership function*, of the first three conditions as a function of the *x* position of the robot with respect to the ball-to-net trajectory. These functions are defined by the designer, and depend on how the robot should try to cut the ball. The consequent of the rules indicate which control variable should be affected, and how: the first three rules involve lateral motion of the robot, while the three following rules involve rotational motion. Each rule affects the corresponding control variable by an amount that depends on the truth value of its condition: smaller or larger adjustments to the robot motion will be generated depending on how much the robot is close to the ball-to-net trajectory. Rules are evaluated and combined according to the standard Mamdani approach.

Behaviors also may incorporate perceptual rules used to communicate the perceptual needs of active behaviors to the perceptual anchoring module (PAM), by using fuzzy rules of the form:

```
if <predicate>                then need <object>
```

whose effect is to assert the need for an *object* at a degree that depends on the truth value of *condition*.

In order to give a more concrete impression of how perceptual behaviors are implemented, we show here the *trackbnet1* behavior. This behavior is useful when the robot tries to store a goal in the opposite's net. In this situation it will always concentrate attention on the ball, but when it is close to the ball, it might want to also get information on the relative position of the net, in order to fine motion to head to the net. This can be accomplished with only the following two rules:

```
always                        need BALL
if ballClose                  then need NET1
```

where *always* is a condition which is always true. In a situation where the ball is at 400mm from the robot, the truth-value of ballClose is 0.7, and these rules assert a value of needed of 1.0 for the anchor BALL and of 0.7 for NET1. Behaviors are dynamically activated and deactivated according to the current task and situation, and several behaviors can be active at the same time. The needed values stored in the *S* state are those asserted by the active behaviors, combined by the max operator. This guarantees that perceptual anchoring only depends on the currently active behaviors, hence on the current task.

We build **complex behaviors** by combining simpler ones using fuzzy meta-rules, which activate concurrent sub-behaviors. This procedure can be iterated to build a full hierarchy of increasingly complex behaviors. The mechanism used to perform behavior composition is called Context-Dependent Blending (Saffiotti, 1997). Thus, under the CDB paradigm flexible arbitration policies can be obtained using fuzzy meta-rules of the form:

```
if <predicate>                then use <behavior>
```

For each such rule, the controller evaluates the truth value, in range [0,1], of <condition> and activates <behavior> at a level that corresponds to this value. Several behaviors may be active at the same time: in this case, their outputs are fused by a weighted combination according to the respective activation levels. Fusion is performed on each control variable independently. The use of fuzzy logic to fuse the outputs of concurrent behaviors provides a smooth transition during switching between behaviors (Saffiotti, 1997), which would be difficult to achieve in architectures based on crisp arbitration like subsumption architecture (Brooks, 1986).

As an example, the following meta-rules implement the *gkclearb* behavior, which uses four rules to decide what action to take: turning until the robot faces the ball, moving the robot to approach the ball location, kicking the ball or moving the robot between the ball and the opponent's net. The behavior also controls the type of kick to be applied; depending on the orientation of the robot, it uses either arms or the head.

```
if not(ballSeen)                            then use faceball
if and(ballSeen, ballClose, freeToKick)     then use dokick
if and(ballSeen, ballClose, not(freeToKick))then use alignbnet1
```

One key characteristic of our behaviors combination technique is that there are well-established techniques to perform fuzzy fusion of the output of behaviors (Saffiotti, 1997). The rule-based approach and hierarchical organization allows us to design, implement and test very complex behaviors, like the goalkeeper behavior, with a limited amount of effort. The goalkeeper behavior involves the use of navigation, manipulation, and perceptual actions in a highly dynamic and unpredictable environment. The full goalkeeper has been decomposed into 12 behaviors, which involve more than 70 fuzzy rules (including those in the basic behaviors) plus 12 perceptual rules. The development of these rules required about 4 weeks of work by one person (Martínez & Saffiotti, 2003).

We have implemented a Java based editor for HBM behaviors and a monitor. The HBM editor (Fig. 6b) is a simple text editor with an integrated LUA interpreter, which is used for detecting syntax errors. If the edited behavior contains no syntax errors, it can be sent to the robot at any time, without the need of stopping or rebooting it. The HBM monitor (Fig. 6b) is a visual tool that shows information very helpful for debugging behaviors. It includes: the table of objects (ball, nets, landmarks) with their anchoring value and relative positions, the current estimated robot position, the current output of the HBM (velocities), the current active behavior, a graphical display of the most recently viewed objects in robot centric coordinates, and a graphical display of the current global robot position with its uncertainty. When debugging single behaviors, we can easily contrast what the robot is actually doing and what it should be doing, and knowing what it actually perceives. A very difficult to debug issue is checking the different preconditions of an action with the real robot because the uncertainty in the perception and localization systems. With this approach we can produce a preliminary version of a behavior with several parameters estimated, then put the robot on the field and execute a single behavior. We then monitor the execution of the behavior, modify the LUA code accordingly and send it to the robot. We repeat this process until we are satisfied with the result. We then can execute the full system allowing for the change of the active low-level behavior by the high-level behaviors.

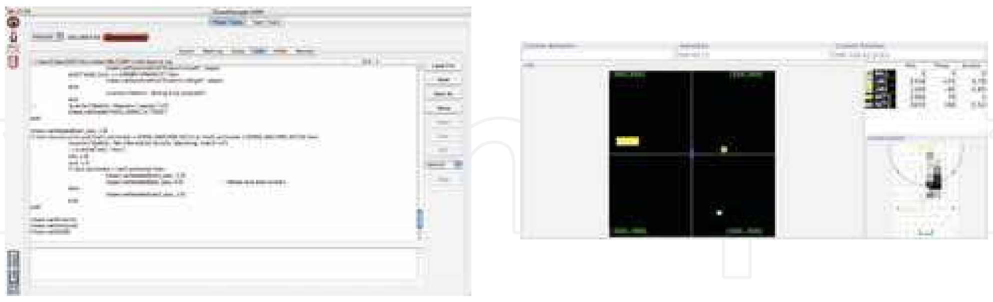
Fig. 6. The HBM tools, code editor (left) and monitor (right)

4.3 Goalkeeper Example

We can use complex behaviors to form even more complex ones, thus creating a *behavior hierarchy*. A simplification of the goalkeeper's strategy is exemplified in Fig. 7. The squared boxes indicate basic behaviors, that is, behaviors that directly control motion and do not call any sub-behaviors.

There is a set of behaviors that are common to all players and are also used by the goalkeeper.

- *lookball*: Turns the robot until it directly sees the ball.
- *go2ball*: Moves the robot to the ball location.
- *dokick*: Moves the robot towards the ball and applies a kick.
- *alignbnet1*: Moves the robot until it is aligned with both the ball and the opponent's net.

While most basic behaviors are coded by means of fuzzy rules (as described in the previous section), there are some cases in which they are not needed. This is the case of the *lookball* behavior. It is intended for finding the ball by means of turning on place. The behavior code is as follows:

```
slowTime       = 400

local ball     = chaos.lps_getLpo(BALL)
local info     = chaos.getBehaviorInfo ()

if info.isNew > 0 then
       sgn = ball.theta / math.abs (ball.theta)
       chaos.setGlobal("BALL_DIRECTION",INTEGER,sgn)
end
sgn = chaos.getGlobal("BALL_DIRECTION")

vrot = 0
if info.timer < slowTime then
       vrot = 50 * sgn
else
       vrot = 75 * sgn
```

```
end

chaos.setNeeded(BALL,1.0)
chaos.setVlin(0)
chaos.setVrot(vrot)
chaos.setVlat(0)
```

For performance reasons, if the robot is not seeing the ball, it is best to turn towards the place where the robot last saw it. This is accomplished using the LUA based state methods, using the global variable BALL_DIRECTION. In addition, during the first four seconds the robot turns slowly (the ball might be close to the robot) and then turns faster (to cover the maximum area per time).

There are some basic behaviors which are specific for the goalkeeper:

- *gktracklms*: Select the least recently seen landmark as a desired perceptual goal. It directly calls the LUA special method *trackLandMarks()*
- *gkkeepout*: Turns the robot slowly moving until it is outside its net.
- *gkkeeparea*. Put the robot facing forward and then moves it to the goalkeeper area.
- *gkkeepbarea*. Put the robot facing backwards and then moves it below the penalty area.
- *gkcutb*. Turn and move sideways in order to intercept the ball trajectory.

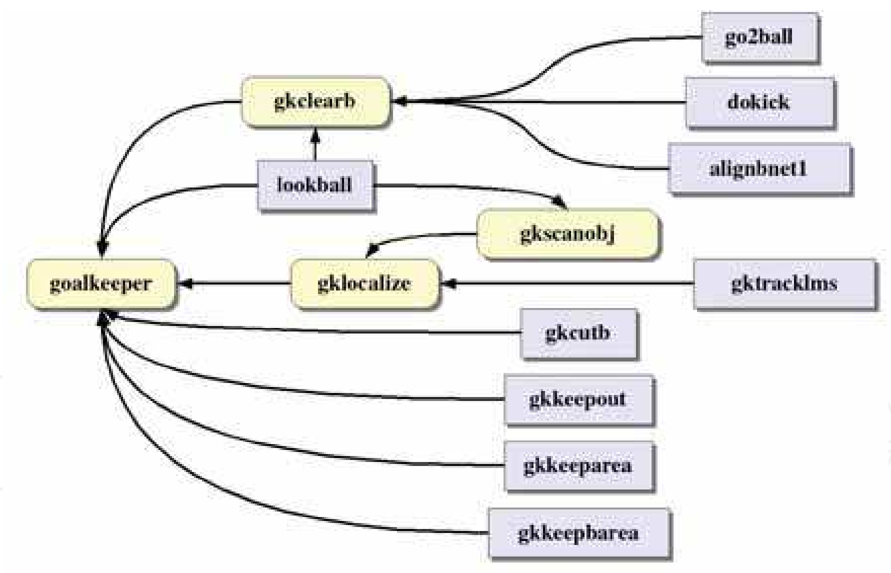

Fig. 7. Behavior hierarchy for the goalkeeper

The other behaviors in the hierarchy are complex behaviors intended to perform the following goalkeeper tasks:

- *goalkeeper*: Top-level goalkeeper behavior.
- *gkscanobj*: Scan the field on place until a given object is found. Look at landmarks and nets cyclically.
- *gklocalize*. Get a better position estimation through looking for the closest landmarks.
- *gkclearb*. Turn and move forward in order to kick the ball out of the goalkeeper's area.

Some of the above behaviors express specific perceptual needs by way of perceptual rules. For instance, most behaviors express a need for the ball position. The *gkkeepout* and *gkkeeparea* behaviors both need to have accurate information about the robot's own location in the field, and hence they express a need for the most probably visible landmarks, including the opponent's net. Moreover, *gkcutball* and *gkclearb* behaviors both need to have accurate information about the ball position in addition to the robot's own accurate location in the field (to avoid self scoring). In general, the overall perceptual needs of the *goalkeeper* behavior depend on which sub-behaviors are activated at every moment, and are used to direct perception.

5. High-level Behaviors

5.1 The HFSM Model and Tools

For specifying and implementing high-level behaviors we make use of the Hierarchical Finite State Machine (HFSM) paradigm (Hugel et al, 2005). In some way or another, the notion of state is usually implied on the execution of a high-level behavior: there is a necessity on knowing what was the state of execution between two successive invocations of the behavior. In the RoboCup, most teams implement some form of state machines, be they ad hoc implementations or the output of formal tools like XABSL (Loetzsch et al., 2006) or Petri Nets (Ziparo and Iocchi, 2006).

An HFSM consists on a set of states, meta-states, which are state machines, and transitions between states and/or meta-states. When the robot is in a state, it executes the corresponding state's code, which is standard LUA code accessing to local perceptions (ball, nets, landmarks, etc.), global information (global ball position, own location, etc.) and shared messages (teammate positions, etc.) from other robots. The states usually invoke low-level behaviors (*faceball*, *go2ball*, etc). The transitions between states and/or meta-states define the conditions to change from one to the other state by the initial and final conditions of the states or meta-states. The meta-states are automata in their self and must carry out all the preconditions for an automaton; they must have an initial state, which is executed first, and cannot contain itself, i.e., an automaton can contain several meta-states (but not itself) and these meta-state can be referenced in different places in our automaton or from other automaton without duplicating code. This simple yet powerful paradigm allows us to specify and reuse machines than can be used inside others, avoiding the repetition of code and allowing for a better code management. For instance, if a typical set of actions is modeled using a meta-state called *Score*, whenever the conditions for scoring are satisfied and no matter in which state we are, we can always call that meta-state. Thus we write code for scoring once, and we can invoke it in different situations. This is much like a subroutine is in programming.

In our implementation, the main HFSM is a meta-state. Meta-states can be referenced from different states without duplicating code. The transitions are implemented defining two conditions to change between states: test code and priority. When the robot is in a state, it checks the test conditions from all transitions from this state. If some of these tests are satisfied, the new state for the robot is the final state of that transition otherwise the robot continues executing the current state. In the case many transitions come out from the same state, the priority associated to the transition is used to decide the final state. In practice, transition code is checked considering the priority, and when the conditions of a transition are satisfied, the robot's state is changed. When the transitions are checked, not only the transitions from the state (inside the meta-state) are checked but also the transitions from the meta-state. In fact, transitions from the meta-state are first checked, that is, transitions from meta-states have more priority than transitions from states.

The HFSM mechanism is also used for role assignment and execution (Agüero at al., 2006a), so that field players can play different roles in different game conditions. For example, if a defender goes to the ball it can change its role to attacker and another robot should change to its own role to defender. This can be easily achieved defining several meta-states and sharing information between the robots in order to know when to change. Fig. 8 shows an example of the HFSM of field players using three roles: attacker, defender and supporter. Each one of these roles is implemented as a meta-state, and the transitions reflect the conditions for switching from a given role to another.

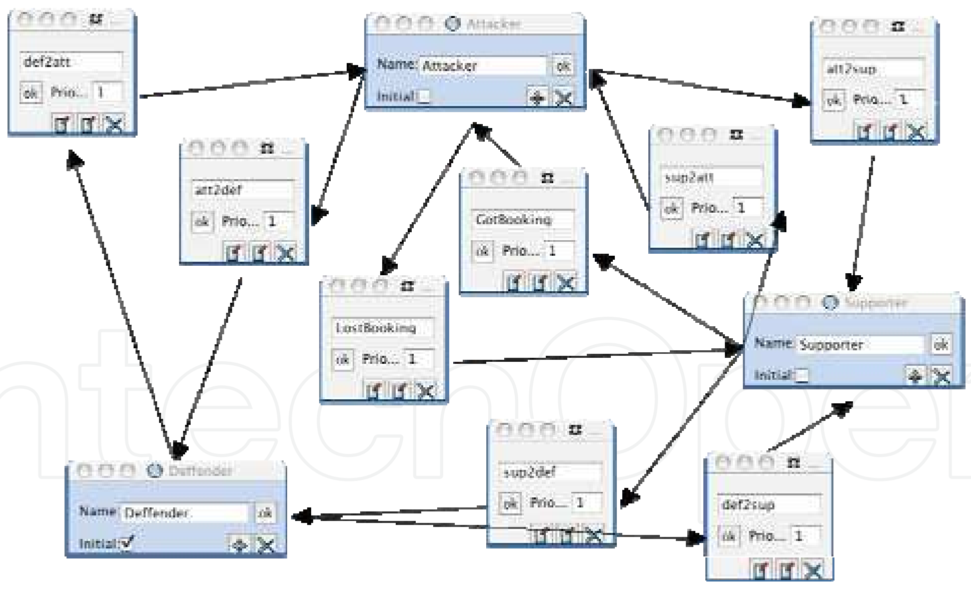

Fig. 8. A sample HFSM with meta-states and transitions

A very important feature of the HFSM model is that a visual tool can be easily produced, so that code development is greatly sped up. We have implemented a Java based visual editor for HFSM which is based on a early implementation of the tool by the Université de Versailles (Hugel et al, 2005). We have redesigned the GUI and in addition to generating C++ code, we also generate LUA code. The development process with the HFSM tool is as follows. The user starts by creating a set of states and meta-states with their corresponding transitions. Then the corresponding LUA code for the states and transitions is edited. Finally, the user can generate LUA code using the corresponding menu and then transfer it to the robot. The generated LUA code is included in a single LUA file. This process can be repeated over time without the need of stopping or rebooting the robot. The GUI includes a tree view of the whole HFSM (to get an overview of all the states, meta-states and transitions) and a visual view of the selected meta-state (Fig. 9a). When the code of a state or transition is edited, a programming window is open with a syntax-coloring editor and the list of available behaviors that can be invoked (Fig. 9b).

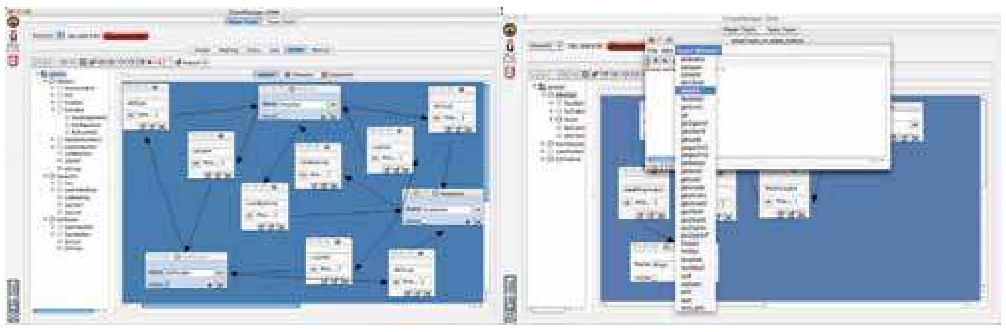
Fig. 9. The HFSM editor, for meta-state edition (left) and code edition (right)

5.2 Attacker Example

In order to show how to apply HFSM to robotics soccer players, we present and describe a simple attacker, quite similar to the one currently used for competitions, being the major difference the lack of role negotiation (which is implemented in a higher level state machine, as shown in Fig. 8). The *Attacker* HFSM (shown in Fig. 10) is composed of three states: *FaceBall*, *GoToBall*, and *Score*. This HFSM should be activated only when the ball is in direct view of the robot (this is something that a higher level state machine should be take care for). The rationale behind the Attacker is to turn towards the ball (by calling the *faceball* behavior from the *FaceBall* state), then move towards the ball (by calling the *go2ball* behavior from the *GoToBall* state), and finally pushing the ball towards the net. This last action is complex enough to be divided into some stages, and thus the state *Score* is in fact a meta-state.

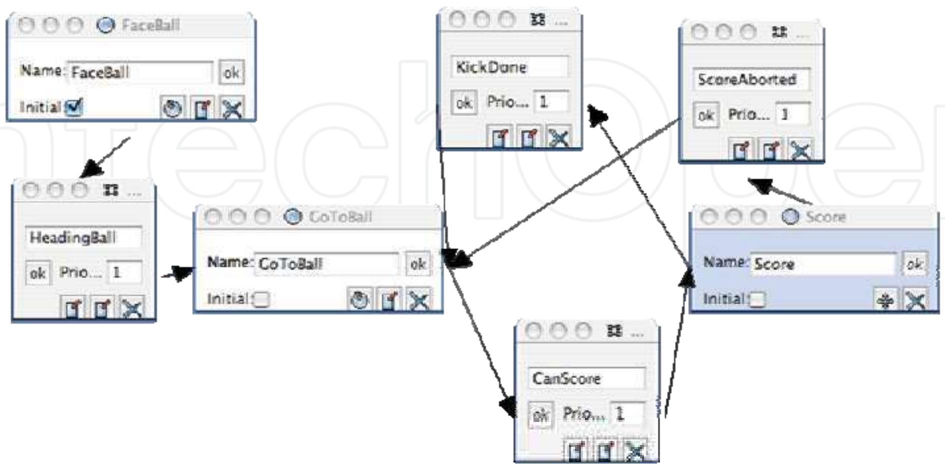

Fig. 10. The *Attacker* HFSM

The *Score* meta-state (shown in Fig. 11) is composed of three states: *ApproachBall*, *Align*, and *DoKick*. This meta-state should be activated only when the robot is close enough to the ball (approximately 50 cm). The rationale behind the Score is to move in a more precise way towards the ball (by calling the *go2ball* behavior from the *ApproachBall* state) while taking care than the robot, the ball and the net are more or less aligned (by calling the *alibnet1* behavior from the *Align* state), and finally produce a kicking movement that pushes the ball into the net (by calling the *dokick* behavior from the *DoKick* state).

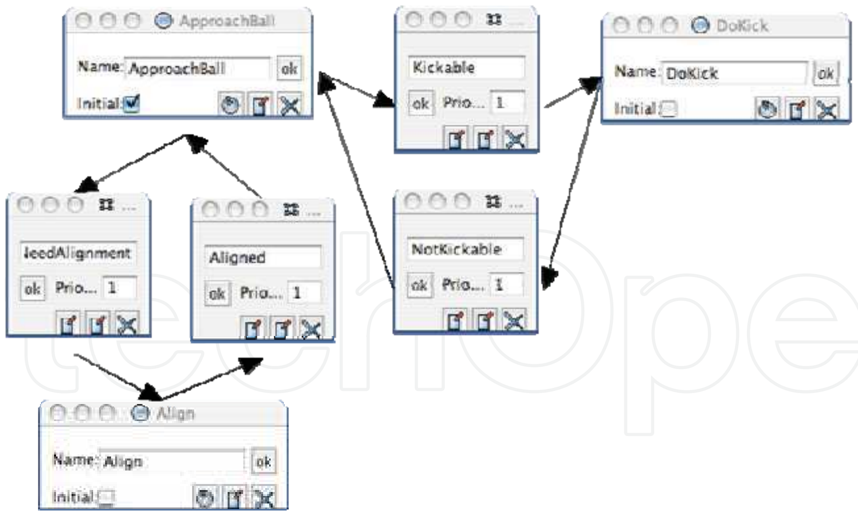

Fig. 11. The *Score* meta-state

In general, the LUA code embedded in the previously described states is very simple, and consists in a call to the corresponding behavior, because each state has been designed to correspond to a single behavior. For instance, the *DoKick* state contains the code:

```
chaos.setBehavior("dokick")
```

There are some states that need a little more complex code. For example, the *GoToBall* state not only calls the corresponding single behavior *go2ball* for robot movement, but also takes care of calling the appropriate localization related tasks, to avoid that in long displacements the robot misses the landmarks. This is the main difference between the state *GoToBall* and *ApproachBall*: the later does not perform localization tasks to concentrate the visual focus of the robot on the ball. The *GoToBall* state LUA code is as follows:

```
local gs       = chaos.gsGetMyPos()
local ball     = chaos.lps_getLpo(BALL)
local net1     = chaos.lps_getLpo(NET1)
local net2     = chaos.lps_getLpo(NET2)

if (gs.quality < 0.6) then
       chaos.trackLandMarks()
end
if (net1.anchored < 0.5) and (net2.anchored < 0.5) then
       chaos.setNeeded (NET1, 1.0)
end

chaos.setBehavior("go2ball")
```

The code performs three activities. If the robot is not properly localized (its localization quality goes below 0.6) a special method *trackLandMarks()* is called, which sets necessities for the most relevant landmarks or nets. In addition, if one of the nets has not been perceived for some time, it sets the necessity for the opponent's net, which might be helpful when attacking. Finally, it always invokes the *go2ball* behavior.

Besides having LUA code in the states, there is also LUA code in the transitions, which basically check for the preconditions of activation of the corresponding state. These usually imply simple tests of either the local or global state (LPS or GS). For instance, the *Kickable* transition code from state *ApproachBall* to *DoKick* (Fig. 11) is as follows:

```
local ball = chaos.lps_getLpo(BALL)

if ball.rho < 600 then
       return true
else
       return false
end
```

This code tests if the distance of the robot to the ball is less than 60cm, in which case the ball can be kicked.

6. Conclusions

This work has presented the architecture and behavioral programming model used to develop a team of the Sony Four-Legged League, which is one of the official leagues of the RoboCup. This league is a very demanding scenario, with high uncertainty in perceptions and limited processing power. In addition, having a competition implies that some dates the software development and tuning presents high activity peaks. These facts condition the way robots have to be programmed. Thus, our main goal has been improving productivity as much as possible while being able to correctly develop all the required behaviors.

The approach consists on adopting a programming architecture that reflects a cognitive separation of modules and allows for an efficient management of modules code. Because the standard programming mode of the AIBO robots makes use of OPEN-R and C++, the on robot behavior development control cycle is very unproductive, and this is typical task that sooner or later must be done (typically in dates close to or during the competition). We have then opted to use an embedded language to make much easier the behavior development task and have selected the LUA language for its many features, being the more important benefits its reduced footprint, its clear language and its execution speed. We have shown different examples of behaviors coded with the language and the main methods of the custom library to access the architecture from LUA.

In order to organize behaviors in the architecture, we divide them into two types, low-level and high-level behaviors, which are implemented using the HBM and the HFSM model respectively. The HBM model allows us to define behaviors by way of fuzzy rules and fuzzy meta-rules in order to cater with uncertainty in both perceptions and actions. The HFSM model allows us to define behaviors by way of state machines in order to sequence high-level tasks. The combination of these two models allows for the combination of the conceptual expressiveness of state machines and the robustness of fuzzy controllers, with the added benefit of being programmed in LUA, which allows for a very productive development cycle.

Although all the work presented has been directed towards robotics soccer, it is important to note that the methodology and tools presented can be used in other scenarios in which the on robot behavior development is an important or crucial task. Two future lines open on this work: porting all the software and technology to a humanoid robot, and incorporating the methodology in the development of a prototype unmanned boat.

7. Acknowledgement

This work has been supported by CICYT project DPI2004-07993-C03-02. The development of the different software pieces could not have been possible without the contribution of many individuals. Without being extensive, we would like to acknowledge: Alessandro Saffioti (University of Örebro) for his implementation of the ThinkingCap architecture and the HBM, in which the one described is based on, Vincent Hugel (Université de Versailles) for his early implementation of the HFSM in which ours is based on. Last but not least, we would like to thank some members of the TeamChaos (Vicente Matellán, Miguel Cazorla, Francisco Bas, Carlos Agüero and Francisco Martín) for implementing, testing or debugging different pieces of software.

8. References

Agüero, C.; Matellán, V.; Cañas, J.M. & Gómez, V. (2006a) Switch! Dynamic Role Exchange among Cooperative Robots. *Proceedings of 2nd International Workshop on Multi-Agent Robotics Systems*, Setúbal, Portugal

Agüero, C.; Martín, F.; Matellán, V. & Martínez-Barberá, H. (2006b) Communications and Simple Coordination of Robots in TeamChaos. *Proceedings of 7th Workshop on Physical Agents (WAF-06)*, Gran Canaria, Spain

Books, R.A. (1986). A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, March 1986, 14-23.

Buschka, P.; Saffiotti, A. & Wasik, Z. (2000). Fuzzy Landmark-Based Localization for a Legged Robot. *Proceedings of International Conference on Intelligent Robotic Systems (IROS)*, pp. 1205-1210, Takamatsu, Japan.

Cameron, J.M.; MacKenzie, D.C.; Ward, K.R.; Arkin, R.C. & Book, W.J. (1993) Reactive Control for Mobile Manipulation, *Proceedings of International Conference on Robotics and Automation*, pp. 228-235, USA.

Cánovas, J.P.; LeBlanc, K. & Saffiotti, A. (2004). Robust Multi-Robot Object Localization Using Fuzzy Logic. *Proceedings of RoboCup 2004*, pp. 247-261, Lisbon, Portugal.

Herrero-Pérez, D.; Martínez-Barberá, H. & Saffiotti, A. (2004). Fuzzy Self-Localization using Natural Features in the Four-Legged League. *Proceedings of RoboCup 2004*, pp. 110-121, Lisbon, Portugal.

Hugel, V.; Amouroux, G.; Costis, T.; Bonnin, P. & Blazevic, P. (2005). Specifications and Design of Graphical Interface for Hirarchical Finite State Machines. *Proceedings of RoboCup 2005*, pp. 648-655, Osaka, Japan.

Kim, C.S.; Seo, W.H.; Han, S.H. & Khatib, O. (2001). Fuzzy logic control of a robot manipulator based on visual servoing. *Proceedings of International Symposium on Industrial Electronics*, pp. 1597-1602.

Ierusalimschy, R.; de Figueiredo, L. H. & Celes, W. (1996). LUA-an extensible extension language. *Software: Practice & Experience,* Vol. 26, No. 6, pp. 635-652.

Lever, P.J.A.; Wang, F-Y. & Chen, D. (1994). A fuzzy control system for an automated mining excavator. *Proceedings of International Conference on Robotics and Automation*, pp. 3284-3289.

Loetzsch, M.; Risler, M. & Jüngel, M. (2006). XABSL - A Pragmatic Approach to Behavior Engineering. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5124-5129.

Martínez-Barberá, H. & Saffiotti, A. (2003). On the strategies of a 4-legged goal-keeper for RoboCup. *Proceedings of the 6th International Conference on Climbing and Walking Robots (WLAWAR-03)*, pp. 745-752, Catania, Italy.

Saffiotti, A.; Ruspini, E.H. & Konolige, K. (1993) Blending reactivity and goal-directedness in a fuzzy controller. *Proceedings of the 2nd IEEE International Conference on Fuzzy Systems*, pp. 134-139.

Saffiotti, A.; Konolige, K. & Ruspini, E.H. (1995). A Multivalued-Logic Approach to Integrating Planning and Control. *Artificial Intelligence*, Vol. 76, No. 1-2, pp. 481-526.

Saffiotti, A. (1997). Fuzzy logic in autonomous robots: behavior coordination. *Proc. of the 6th IEEE International Conference on Fuzzy Systems*, pp. 573-578.

Saffiotti, A. & K. LeBlanc (2000). Active perceptual anchoring of robot behavior in a dynamic environment. *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3796-3802, San Francisco, USA.

Saffiotti, A. & Wasik, Z. (2002). Using hierarchical fuzzy behaviors in the RoboCup domain. In Zhou, Maravall, and Ruan, editors, Autonomous Robotic Systems, pp. 235-262.

Ziparo, V.A. & Iocchi, L. (2006) Petri net plans. *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pp. 267–290, Turku, Finland.