4

Humanoid soccer player design

Francisco Martín, Carlos Agüero, José María Cañas and Eduardo Perdices
Rey Juan Carlos University
Spain

1. Introduction

The focus of robotic research continues to shift from industrial environments, in which robots must perform a repetitive task in a very controlled environment, to mobile service robots operating in a wide variety of environments, often in human-habited ones. There are robots in museums (Thrun et al, 1999), domestic robots that clean our houses, robots that present news, play music or even are our pets. These new applications for robots make arise a lot of problems which must be solved in order to increase their autonomy. These problems are, but are not limited to, navigation, localisation, behavior generation and human-machine interaction. These problems are focuses on the autonomous robots research.

In many cases, research is motivated by accomplishment of a difficult task. In Artificial Intelligence research, for example, a milestone was to win to the chess world champion. This milestone was achieved when *deep blue* won to Kasparov in 1997. In robotics there are several competitions which present a problem and must be solved by robots. For example, Grand Challenge propose a robotic vehicle to cross hundred of kilometers autonomously. This competition has also a urban version named Urban Challenge.

Fig. 1. Standard Platform League at RoboCup.

Our work is related to RoboCup. This is an international initiative to promote research on the field of Robotics and Artificial Intelligence. This initiative proposes a very complex problem, a soccer match, in which several techniques related to these field can be tested, evaluated and compared. The long term goal of the RoboCup project is, by 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.

This work is focused on the *Standard Platform League*. In this league, all the teams use the same robot and changes in hardware are not allowed. This is the key factor that makes that the efforts concentrate on the software aspects rather than in the hardware. This is why this league is known as The *Software League*. Until 2007, the chosen robot to play in this league was Aibo robot. But since 2008 there is a new platform called Nao (figure 1). Nao is a biped humanoid robot, this is the main difference with respect Aibo that is a quadruped robot. This fact has had a big impact in the way the robot moves and its stability while moving. Also, the sizes of both robots is not the same. Aibo is 15 cm tall while Nao is about 55 cm tall. That causes the big difference on the way of perception. In addition to it, both robots use a single camera to perceive. In Aibo the perception was 2D because the camera was very near the floor. Robot Nao perceives in 3D because the camera is at a higher position and that enables the robot to calculate the position of the elements that are located on the floor with one single camera.

Many problems have to be solved before having a fully featured soccer player. First of all, the robot has to get information from the environment, mainly using the camera. It must detect the ball, goals, lines and the other robots. Having this information, the robot has to self-localise and decide the next action: move, kick, search another object, etc. The robot must perform all these tasks very fast in order to be reactive enough to be competitive in a soccer match. It makes no sense within this environment to have a good localisation method if that takes several seconds to compute the robot position or to decide the next movement in few seconds based on the old percerpetion. The estimated sense-think-act process must take less than 200 millisecond to be truly eficient. This is a tough requirement for any behavior architecture that wishes to be applied to solve the problem.

With this work we are proposing a behavior based architecture that meets with the requirements needed to develop a soccer player. Every behavior is obtained from a combination of reusable components that execute iteratively. Every component has a specific function and it is able to activate, deactivate o modulate other components. This approach will meet the vivacity, reactivy and robustness needed in this environment. In this chapter we will show how we have developed a soccer player behavior using this architecture and all the experiments carried out to verify these properties.

This paper is organised as follows: First, we will present in section 2 all relevant previous works which are also focused in robot behavior generation and following behaviors. In section 3, we will present the Nao and the programming framework provided to develop the robot applications. This framework is the ground of our software. In section 4, the behavior based architecture and their properties will be described. Next, in section 5, we will describe how we have developed a robotic soccer player using this architecture. In

section 6, we will introduce the experiment carried out to test the proposed approach and also the robotic soccer player. Finally, section 7 will be the conclusion.

2. Related work

In this section, we will describe the previous works which try to solve the robot behavior generation and the following behaviors. First of all, the classic approaches to generate robot behaviors will be described. These approaches have been already successfully tested in wheeled robots. After that, we will present other approaches related to the RoboCup domain. To end up, we will describe a following behavior that uses an approach closely related to the one used in this work.

There are many approaches that try to solve the behavior generation problem. One of the first successful works on mobile robotics is Xavier (Simmons et al, 1997). The architecture used in these works is made out of four layers: obstacle avoidance, navigation, path planning and task planning. The behavior arises from the combination of these separate layers, with an specific task and priority each. The main difference with regard to our work is this separation. In our work, there are no layers with any specific task, but the tasks are broken into components in different layers.

Another approach is (Stoytchev & Arkin, 2000), where a hybrid architecture, which behavior is divided into three components, was proposed: deliberative planning, reactive control and motivation drives. Deliberative planning made the navigation tasks. Reactive control provided with the necessary sensorimotor control integration for response reactively to the events in its surroundings. The deliberative planning component had a reactive behavior that arises from a combination of schema-based motor control agents responding to the external stimulus. Motivation drives were responsible of monitoring the robot behavior. This work has in common with ours the idea of behavior decomposition into smaller behavioral units. This behavior unit was explained in detail in (Arkin, 2008).

In (Calvo et al, 2005) a follow person behavior was developed by using an architecture called JDE (Cañas & Matellán, 2007). This reactive behavior arises from the activation/deactivation of components called schemes. This approach has several similarities with the one presented in this work.

In the RoboCup domain, a hierarchical behavior-based architecture was presented in (Lenser et al, 2002). This architecture was divided in several levels. The upper levels set goals that the bottom level had to achieve using information generated by a set of virtual sensors, which were an abstraction of the actual sensors.

Saffiotti (Saffiotti & Zbigniew, 2003) presented another approach in this domain: the *ThinkingCap* architecture. This architecture was based in a fuzzy approach, extended in (Gómez & Martínez, 1997). The perceptual and global modelling components manage information in a fuzzy way and they were used for generating the next actions. This architecture was tested in the four legged league RoboCup domain and it was extended in (Herrero & Martínez, 2008) to the Standar Platform League, where the behaviors were

developed using a LUA interpreter. This work is important to the work presented in this paper because this was the previous architecture used in our RoboCup team.

Many researches have been done over the Standar Platform League. The B-Human Team (Röfer et al, 2008) divides their architecture in four levels: perception, object modelling, behavior control and motion control. The execution starts in the upper level which perceives the environment and finishes at the low level which sends motion commands to actuators. The behavior level was composed by several basic behavior implemented as finite state machines. Only one basic behavior could be activated at same time. These finite state machine was written in XABSL language (Loetzsch et al, 2006), that was interpreted at runtime and let change and reload the behavior during the robot operation. A different approach was presented by Cerberus Team (Akin et al, 2008), where the behavior generation is done using a four layer planner model, that operates in discrete time steps, but exhibits continuous behaviors. The topmost layer provides a unified interface to the planner object. The second layer stores the different roles that a robot can play. The third layer provides behaviors called "Actions", used by the roles. Finally, the fourth layer contains basic skills, built upon the actions of the third layer.
The behavior generation decomposition in layers is widely used to solve the soccer player problem. In (Chown et al 2008) a layered architecture is also used, but including coordination among the robots. They developed a decentralized dynamic role switching system that obtains the desired behavior using different layers: strategies (the topmost layer), formations, roles and sub-roles. The first two layers are related to the coordination and the other two layers are related to the local actions that the robot must take.

3. Nao and NaoQi framework

The behavior based architecture proposed in this work has been tested using the Nao robot. The applications that run in this robot must be implemented in software. The hardware cannot be improved and all the work must be focused in improving the software. The robot manufacturer provides an easy way to access to the hardware and also to several high level functions, useful to implement the applications.

Our soccer robot application uses some of the functionality provided by this underlying software. This software is called NaoQi[1] and provides a framework to develop applications in C++ and Python.

NaoQi is a distributed object framework which allows to several distributed binaries be executed, all of them containing several software modules which communicate among them. Robot functionality is encapsulated in software modules, so we can communicate to specific modules in order to access sensors and actuators.

[1] http://www.aldebaran-robotics.com/

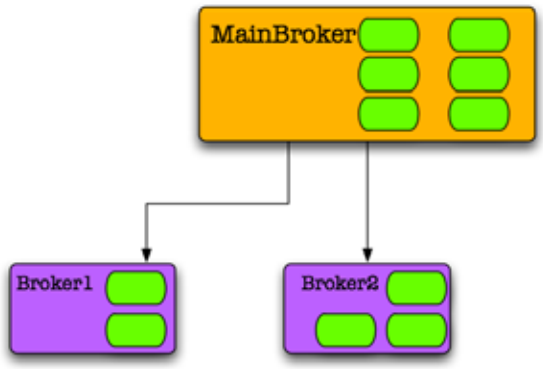

Fig. 2. Brokers tree.

Every binary, also called *broker*, runs independently and is attached to an address and port. Every broker is able to run both in the robot (cross compiled) and the computer. Then we are able to develop a complete application composed by several brokers, some running in a computer and some in the robot, that communicate among them. This is useful because high cost processing tasks can be done in a high performance computer instead of in the robot, which is computationally limited.

The broker's functionality is performed by modules. Each broker may have one or more modules. Actually, brokers only provide some services to the modules in order to accomplish their tasks. Brokers deliver call messages among the modules, subscription to data and so on. They also provide a way to solve module names in order to avoid specifying the address and port of the module.

Fig. 3. Modules within MainBroker.

A set of brokers are hierarchically structured as a tree, as we can see in figure 2. The most important broker is the *MainBroker*. This broker contains modules to access to robot sensors and actuators and other modules provide some interesting functionality (figure 3). We will describe some of the modules intensively used in this work:

- The main information source our application is the camera. The images are fetched by *ALVideoDevice* module. This module uses the Video4Linux driver and makes the images available for any module that create a proxy to it, as we can observe in figure 4. This proxy can be obtained locally or remotelly. If locally, only a reference

to the data image is obtained, but if remotelly all the image data must be encapsulated in a SOAP message and sent over the network.
To access the image, we can use the normal mode or the direct raw mode. Figure 5 will help to explain the difference. Video4Linux driver maintains in kernel space a buffer where it stores the information taken from the camera. It is a round robin buffer with a limited capacity. NaoQi unmaps one image information from Kernel space to driver space and locks it. The difference in the modes comes here. In normal mode, the image transformations (resolution and color space) are applied, storing the result and unlocking the image information. This result will be accessed, locally or remotelly, by the module interested in this data. In direct raw mode, the locked image information is available (only locally and in native color space, YUV422) to be accessed by the module interested in this data. This module should manually unlock the data before the driver in kernel space wants to use this buffer position (around 25 ms). Fetching time varying depending on the desired color space, resolution and access mode, as we can see in figure 6.

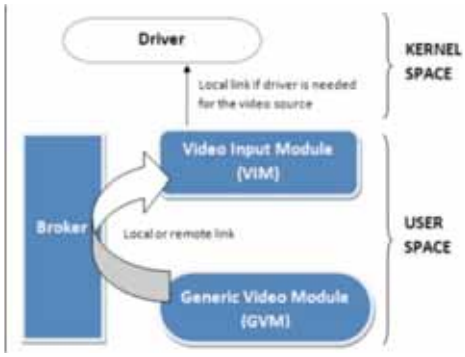

Fig. 4. NaoQi vision architecture overview.

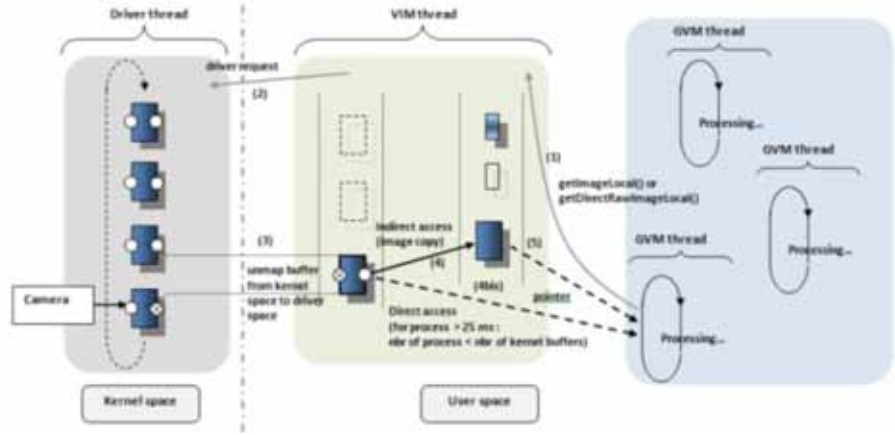

Fig. 5. Access to an image in the NaoQi framework.

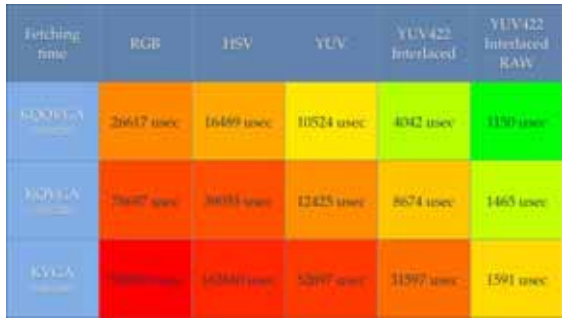

Fig. 6. Access time to the image depending on resolution and space color. Last column is *direct raw* mode.

- In order to move the robot, NaoQi provides the *ALMotion* module. This module is responsible for the actuators of the robot. This module's API let us move a single joint, a set of joints or the entire body. The movements can be very simple (p.e. set a joint angle with a selected speed) or very complex (walk a selected distance). We use these high level movement calls to make the robot walk, turn o walk sideways. As a simple example, the `walkStraight` function is:

  ```
  void walkStraight (float distance, int pNumSamplesPerStep)
  ```

  This function makes the robot walk straight a `distance`. If a module, in any broker, wants to make the robot walk, it has to create a proxy to the *ALMotion* module. Then, it can use this proxy to call any function of the *ALMotion* module.

  The movement generation to make the robot walk is a critical task that NaoQi performs. The operations to obtain each joint position are critical. If these real time operations miss the deadlines, the robot may lost the stability and fall down.

- NaoQi provides a thread-safe module for information sharing among modules, called *ALMemory*. By its API, modules write data in this module, which are read by any module. NaoQi also provides a way to subscribe and unsubscribe to any data in *ALMemory* when it changes or periodically, selecting a class method as a callback to manage the reception. Besides this, *ALMemory* also contains all the information related to the sensors and actuators in the system, and other information. This module can be used as a *blackboard* where any data produced by any module is published, and any module that needs a data reads from *ALMemory* in order to obtain it.

As we said before, each module has an API with the functionality that it provides. Brokers also provide useful information about their modules and their APIs via *web services*. If you use a browser to connect to any broker, it shows all the modules it contains, and the API of each one.

When a programmer develops an application composed by several modules, she decides to implement it as a dynamic library or as a binary (broker). In the dynamic library (like a plug-in) way, the modules that it contains can be loaded by the *MainBroker* as its own modules. Using this mechanism the execution speeds up, from point of the view of communication among modules. As the main disadvantage, if any of the modules crashes, then *MainBroker* also crashes, and the robot falls to the floor. To develop an application as a separate broker makes the execution safer. If the module crashes, only this module is affected.

The use of NaoQi framework is not mandatory, but it is recommended. NaoQi offers high and medium level APIs which provide all the methods needed to use all the robot's functionality. The movement methods provided by NaoQi send low level commands to a microcontroller allocated in the robot's chest. This microcontroller is called DCM and is in charge of controlling the robot's actuators. Some developers prefer (and the development framework allows it) not to use NaoQi methods and use directly low level DCM functionality instead. This is much laborious, but it takes absolute control of robot and allows to develop an own walking engine, for example.

Nao robot is a fully programmable humanoid robot. It is equipped with a x86 AMD Geode 500 Mhz CPU, 1 GB flash memory, 256 MB SDRAM, two speakers, two cameras (non stereo), Wi-fi connectivity and Ethernet port. It has 25 degrees of freedom. The operating system is Linux 2.6 with some real time patches. The robot is eqquiped with a microcontroller ARM 7 allocated in its chest to controll the robot’s motors and sensors, called DCM.

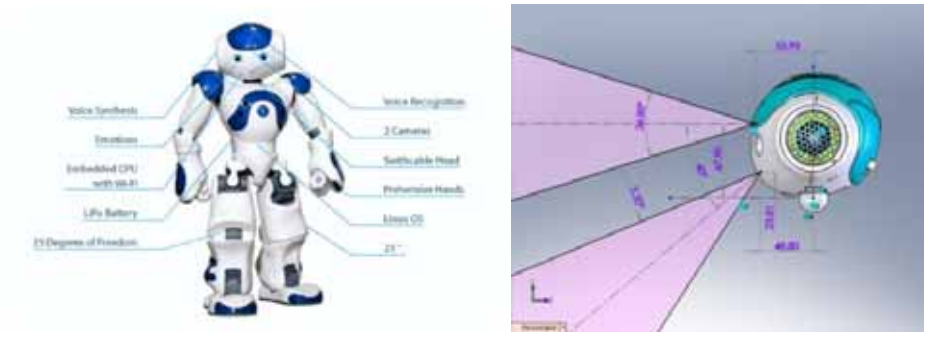

Fig. 7. Aldebaran Robotics’ Nao Robot.

These features impose some restrictions to our behavior based architecture design. The microprocessor is not very powerful and the memory is very limited. These restrictions must be taken into account to run complex localization or sophisticathed image processing algorithms. Moreover, the processing time and memory must be shared with the OS itself (an GNU/Linux embedded distribution) and all the software that is running in the robot, including the services that let us access to sensors and motors, which we mentioned before. Only the OS and all this software consume about 67% of the total memory available and 25% of the processing time.

The robot hardware design also imposes some restrictions. The main restriction is related to the two cameras in the robot. These cameras are not stereo, as we can observe in the right side of the figure 7. Actually, the bottom camera was included in the last version of the robot after RoboCup 2008, when the robot designer took into account that it was difficult track the ball with the upper camera (the only present at that time) when the distance to the ball was less than one meter. Because of this non stereo camera characteristic, we can't estimate elements position in 3D using two images of the element, but supposing some other characteristics as the heigh position, the element size, etc.

Besides of that, the two cameras can't be used at same time. We are restricted to use only one camera at the time, and the switching time is not negligible (about 70 ms). All these restrictions have to taken into account when designing our software.

The software developed on top of NaoQi can be tested both in real robot and simulator. We use Webots (figure 8) (MSR is also available) to test the software as the first step before testing it in the real robot. This let us to speed up the development and to take care of the real robot, whose hardware is fragile.

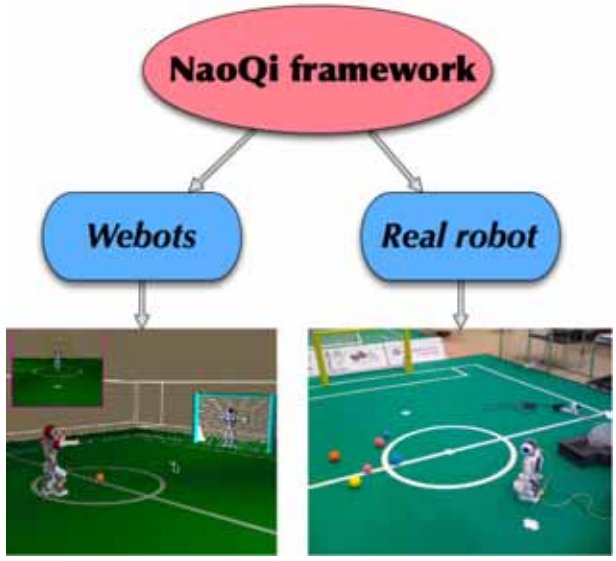

Fig. 8. Simulated and real robot.

4. Behavior based architecture for robot applications

The framework we presented in the last section provides useful functionality to develop a software architecture that makes a robot perform any task. We can decompose the functionality in modules that communicate among them. This framework also hides almost all the complexity of movement generation and makes easy to access sensors (ultrasound, camera, bumpers…) and actuators (motors, color lights, speaker…).

It is possible to develop basic behaviors using only this framework, but it is not enough for our needs. We need an architecture that let us to activate and deactivate components, which is more related to the cognitive organization of a behavior based system. This is the first step to have a wide variety of simple applications available. It's hard to develop complex applications using NaoQi only.

In this section we will describe the design concepts of the robot architecture we propose in this chapter. We will address aspects such as how we interact with NaoQi software layer, which of its functionality we use and which not, what are the elements of our architecture, how they are organized and timing related aspects.

The main element in the proposed architecture is the *component*. This is the basic unit of functionality. In any time, each component can be active or inactive. This property is set using the start/stop interface, as we can observe in figure 6. When it is active, it is running and performing a task. When inactive, it is stopped and it does not consume computation resources. A component also accepts modulations to its actuation and provides information of the task it is performing.

For example, lets suppose a component whose function is perceive the distance to an object using the ultrasound sensors situated in the robot chest. The only task of this component is to detect, using the sensor information, if a obstacle is in front of the robot, on its left, on its right or there is not obstacle in a distance less than D mm. If we would like to use this functionality, we have to activate this component using its start/stop interface (figure 9). We may modulate the D distance and ask whenever we want what is this component output (front, left, right or none). When this is information is no longer needed, we may deactivate this component to stop calculating the obstacle position, saving valuable resources.

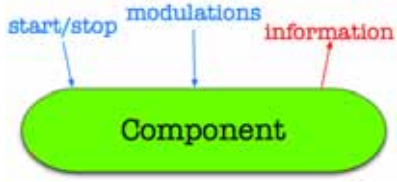

Fig. 9. Component inputs and outputs.

A component, when active, can activate another components to achieve its goal, and these components can also activate another ones. This is a key idea in our architecture. This let to decompose funtionality in several components that work together. An application is a set of components which some of them are activated and another ones are deactivated. The subset of the components that are activated and the activation relations are called *activation tree*. In figure 10 there is an example af an *activation tree*. When component A, the root component, is activated, it activates component B and E. Component B activates C and D. Component A needs all these components actived to achieve its goal. This estructure may change when a component is modulated and decides to stop a component and activate another more adequate one. In this example, component A does not need to know that B has activated C and D. The way component B performs its task is up to it. Component A is only interested in the component B and E execution results.

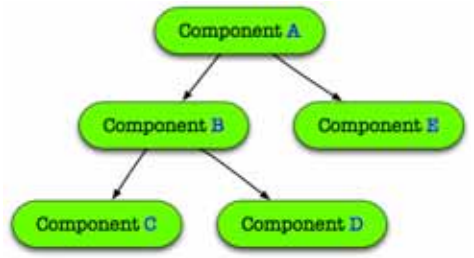

Fig. 10. *Activation tree* composed by several components.

Two differents components are able to activate the same child component, as we can observe in figure 11. This property lets two components to get the same information from a component. Any of them may modulate it, and the changes affect to the result obtained in both component.

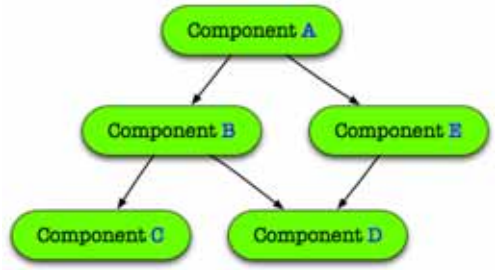

Fig. 11. *Activation tree* where B and D activates D component.

The *activation tree* is no fixed during the robot operation. Actually, it changes dinamically depending on many factors: main task, environment element position, interaction with robots or humans, changes in the environment, error or falls… The robot must adapt to the changes in these factors by modulating the lower level components or activating and deactivating components, changing in this way the static view of the tree.

The main idea of our approach is to decompose the robot functionality in these components, which cooperate among them to make arise more complex behaviors. As we said before, component can be active or inactive. When it is active, a `step()` function is called iteratively to perform the component task.

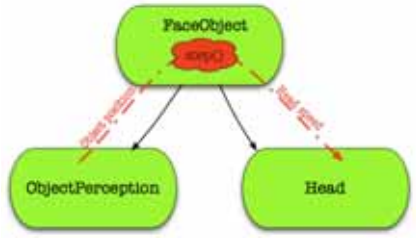

Fig. 12. Activation tree with two low level components and a high level components that modulates them.

As an example, in figure 12 we show an activation tree composed by 3 components. `ObjectPerception` is a low level component that determines the position of an interesting object in the image taken by the robot's camera. `Head` is a low level component that moves the head. These components functionality is used by a higher level component called `FaceObject`. This component activates both low level components, that execute iteratively. Each time `FaceObject` component performs its `step()` function, it asks to `FaceObject` for the object position and modulates `Head` movement to obtain the global behavior: facing the object.

Components can be very simple or very complex. For example, the `ObjectPerception` component of the example is a perceptive iterative component. It does't modulate or activate another component. It only extract information from an image. The `ObjectPerception` component is a iterative controller, that activate and modulate another components. Another components may activate and deactivate components dinamically dependining on some stimulus. They are implemented as *finite state machine*. In each state there is set of active components, and this set is eventually different to the one in other state. Transitions among states reflect the need to adapt to the new conditions the robot must face to.

Using this guideline, we have implemented our architecture in a single NaoQi module. The components are implemented as *Singleton* C++ classes and they communicate among them by method calls. It speeds up the communications with respect the SOAP message passing approach.

When NaoQi module is created, it starts a thread which continuosly call to `step()` method of the root component (the higher level component) in the *activation tree*. Each `step()` method of every component at level *n* has the same structure:

1. Calls to `step()` method of components in *n-1* level in its branch that it wants to be active to get information.
2. Performs some processing to achieve its goal. This could include calls to components methods in level *n-1* to obtain information and calls to lower level components methods in level *n-1* to modulate their actuation.
3. Calls to `step()` methods of component in *n-1* level in its branch that it wants to be active to modulate them.

Each module runs iteratively at a configured frequency. It has not sense that all the components execute at the same frequency. Some informations are needed to be refreshed very fast, and some decisions are not needed to be taken such fast. Some components may need to be configured at the maximun frame rate, but another modules may not need such high rate. When a `step()` method is called, it checks if the elapsed time since last execution is equal or higher to the established according to its frequency. In that case, it executes 1, 2 and 3 parts of the structure the have just described. If the elapsed time is lower, it only executes 1 and 3 parts. Tipically, higher level components are set up with lower frequency than lower level ones, as we can observe in figure 13.

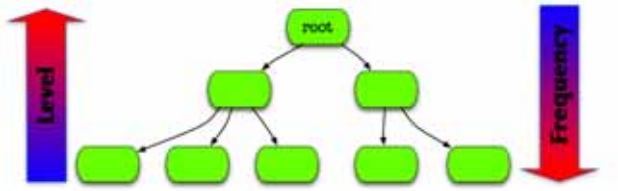

Fig. 13. Activation tree with a root component in the higher level. As higher is the level, lower is the frequency.

Using this approach, we can modulate every module frequency, and be aware of situations where the system has a high load. If a module does not meet with its (soft) deadline, it only makes the next component to executed a little bit late, but its execution is not discarted (*graceful degradation*).

In next section we will describe some of the components developed using this approach for our soccer player application, clarifying some aspects not fully described .

5. Soccer player design

The concepts presented in last section summarizes the key ideas of this architecture design. We have presented the *component* element, how these components can be activated in a activation tree and how they execute. This architecture is focused to develop robot applications using a behavioral approach. In this section we will present how, using this architecture, we solve the problem previously introduced in the section 1: play soccer.
A soccer player implementation is defined by the set of activation trees and how the components modulate another ones. These components are related to perception and actuations and are part of the basis of this architecture. High level components make use of these lower level components to achieve higher level components. So, the changes between soccer player implementations depends on these higher level components. We will review in next sections how particular components to make a robot play soccer are designed and implemented.

5.1 Soccer player perception

At RoboCup competition, the environment is designed to be perceived using vision and all the elements have a particular color and shape. Nao is equipped with two (non-stereo) cameras because they are the richest sensors available in robotics. This particular robot has also ultrasound sensors to detect obstacles in front of it, but a image processing could also detect the obstacle and, additionally, recognize whether it is a robot (and what teams it belong) or another element. This is why we have based the robot perception in vision.

The perception is carried out by the `Perception` component. This component obtains the image from one of the two cameras, process it and makes this information available to any component interested on it using the API it implements. Furthermore, it may calculate the 3D position of some elements in the environment. Finally, we have developed a novel approach to detect the goals, calculating at same time an estimation of the robot pose in 3D.

The relevant elements in the environment are the ball, the green carpet, the blue net, the yellow net, the lines and the other robots. The ilumination is not controlled, but it is supposed to be addequate and stable. The element detection is made attending to its color, shape, dimensions and position with respect the detected border of the carpet (to detect if it is in the field).

We want to use *direct raw* mode because the fetching time varying depending on the desired color space, resolution and access mode, as we can see in figure 14.

Fig. 14. Relevant elements in the environment.

Perception component can be modulated by other components that uses it to set different aspects related to the perception:

- Camera selection. Only bottom or upper camera is active at same time.
- Set the stimulus of interest.

We have designed this module to detect only one stimulus at the same time. There are four types of stimulus: ball in the image, goals in the image, ball in ground coordinates and goal in robot coordinates. This is usefull to avoid unncecessary processing when any of the elements are not usefull.

5.1.1 Ball and goal in image

These stimulus detection is performed in the `step()` method of this component. Once the image obtained is filtered attending only to the color of the element we want to detect. To speed up this process we use a lookup table. In the next step, the resulting pixels on the filtering step are grouped in blobs that indicate connected pixels with the same color. In the last step, we apply some conditions to each blob. We test the size, the density, the center mass position with respect the horizont, etc. The horizon is the line that indicates the upper border of the green carpet. Ball is always under horizon, and nets have a maximun and minimum distance to it. All this process for ball and net is shown in figure 15.

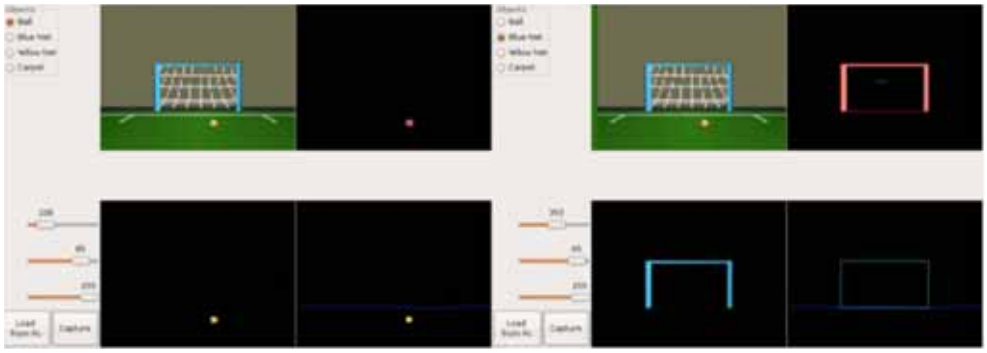

Fig. 15. Element detection process.

The element detected is coded as a tuple {[-1,1],[-1,1]}, indicating the normalized position {X,Y} of the object in the image. When `step()` method finishes, any component can ask for this information using method such us `getBallX()`,`getBlueNetY()`, etc.

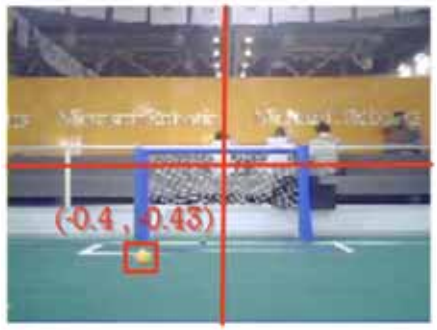

Fig. 16. Tuple containing the ball position.

5.1.2 Ball in ground coordinates

In last subsection we describe how the element information is calculated. This information is 2D and is related to the image space. Sometimes it is not enough to achieve some task. For example, if the robot wants to be aligned in order to kick the ball, it is desired to have the ball position available in the robot space reference, as we can see in figure 17.

Obtain the element position in 3D is not an easy task, and it is more difficult in the case of an humanoid robot that walks and perceive an element with a single camera. We have placed the robot axes in the floor, centered under the chest, as we can see in figure 17. The 3D position $\{O_X, O_Y, O_Z=0\}$ of the observed element O (red lines in figure 11) is with respect the robot axes (blue lines in figure 17).

To calculate the 3D position, we start from the 2D position of the center of the detected element related to the image space in one camera. Using the *pinhole* model, we can calculate the a 3D point situated in the line that joints the center of the camera and the element position in the camera space.

Once obtained this point we represent this point and the center of the camera in the robot space axes. We use NaoQi functions to help to obtain the transformation from robot space to camera space. Using *Denavit and Hartenberg* method (Denavit, 1955), we obtain the *(4x4)* matrix that correspond to that transform (composed by rotations and translations).

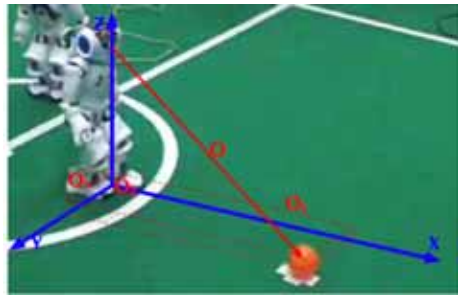

Fig. 17. Element 3D position and the robot axes.

Each time this component is asked for the 3D position of an image element, it has to calculate this transformation matrix (each time the joint angles from foots to camera are differents) and apply to to the 2D element position in the camera frame calculated in the last `step()` iteration.

5.1.3 Goal in robot coordinates

Once the goal has been properly detected in the image, spatial information can be obtained from the that goal using geometric 3D computations. Let Pix1, Pix2, Pix3 and Pix4 be the pixels of the goal vertices in the image. The position and orientation of the goal relative to the camera can be inferred, that is, the 3D points P1, P2, P3 and P4 corresponding to the goal vertices. Because the absolute positions of both goals are known (AP1,AP2,AP3,AP4) that information can be reversed to compute the camera position relative to the goal, and so, the absolute location of the camera (and the robot) in the field. In order to perform such 3D geometric computation the robot camera must be calibrated.

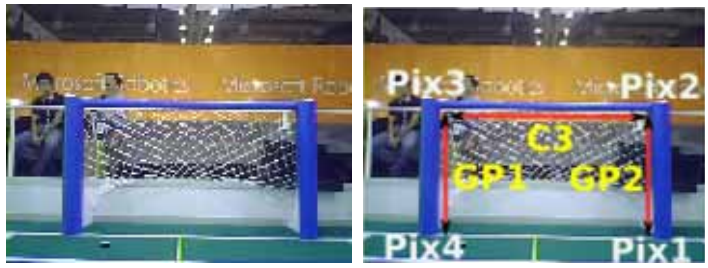

Fig. 18. Goal detection.

Two different 3D coordinates are used: the absolute field based reference system and the system tied to the robot itself, to its camera. Our algorithm deals with line segments. It works in the absolute reference system and finds the absolute camera position computing some restrictions coming from the pixels where the goal appears in the image.

There are three line segments in the goal detected in the image: two goalposts and the crossbar. Taking into consideration only one of the posts (for instance GP1 at figure 18) the way in which it appears in the image imposes some restrictions to the camera location. As we will explain later, a 3D thorus contains all the camera locations from which that goalpost is seen with that length in pixels (figure 19). It also includes the two corresponding goalpost vertices. A new 3D thorus is computed considering the second goalpost (for instance GP2 at figure 18), and a third one considering the crossbar. The real camera location belongs to the three thorus, so it can be computed as the intersection of them.

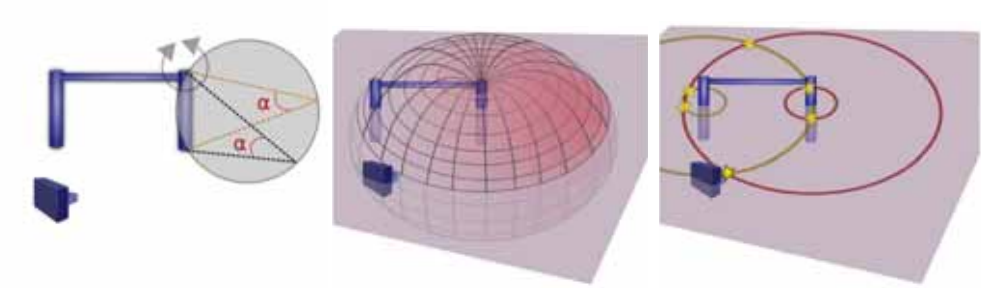

Fig. 19. Camera 3D position estimation using a 3D thorus built from the perception.

Nevertheless the analytical solution to the intersection of three 3D thorus is not simple. A numerical algorithm could be used. Instead of that, we assume that the height of the camera above the floor is known. The thorus coming from the crossbar is not needed anymore and it is replaced by a horizontal plane, at h meters above the ground. Then, the intersection between three thorus becomes the intersection between two parallel thorus and a plane. The thorus coming from the left goalpost becomes a circle in that horizontal plane, centered at the goalpost intersection with the plane. The thorus coming from the right goalpost also becomes a circle. The intersection of both circles gives the camera location. Usually, due to simmetry, two different solutions are valid. Only the position inside the field is selected.

To compute the thorus coming from one post, we take its two vertices in the image. Using projective geometry and the intrisinc parameters of the camera, a 3D projection ray can be computed that traverses the focus of the camera and the top vertex pixel. The same can be computed for the bottom vertex. The angle α between these two rays in 3D is calculated using the dot product.

Let's now consider one post at its absolute coordinates and a vertical plane that contains it. Inside that plane only the points in a given circle see the post segment with an angle α . The thorus is generated rotating such circle around the axis of the goalpost. Such thorus contains all the camera 3D locations from which that post is seen with a angle α, regardless its orientation. In other words, all the camera positions from which that post is seen with such pixel length.

Fig. 20. Estimation of the robot position

5.2 Basic movements

Robot actuation is not trivial in a legged robot. It is even more complicated in biped robots. The movement is carried out by moving the projection of center of mass in the floor (zero moment point, ZMP) to be in the support foot. This involves the coordination of almost all the joints in the robot. In fact, it is common even use the arms to improve the balance.

It is hard to develop complete walking mechanism. This means to generate all the joint positions in every moment, which is not mathematically trivial. It also involves real time aspects because if a joint command is sent late, even few milliseconds, the result is fatal, and the robot may fall to floor. All this work is critical for any task that the robot performs, but it has not very valuable, from the scientific point of view. Sometimes there is not chance, and this work has to be done. For example, we had to calculate every joint position each 8 milliseconds to make walk the quadruped AiBo robot because there were not any library or function to make it walk. Luckily, NaoQi provides some high level functions to make the robot move. There are function to walk (straight or side), turn or move in many ways an only joint. It is not mandatory to use it, and everyone can develop his own walking mechanism, but it is difficult to improve the results that NaoQi provides.

We have chosen to use NaoQi high level functionality to move the robot. We do not use these function in every component that wants to move the robot in any way. This would incur in conflicts and it is not desiderable mix high and low level functions. For these reasons, we have developed some components to manage the robot movement, providing and standard and addequate interface for all the component that wants to perform any actuation. This interface is implemented by the `Body`, `Head` and `FixMove` components.

5.2.1 `Body` component

The `Body` component manages the robot walk. Its modulation consists in two parameters: straight velocity (v) and rotation velocity (w). Each parameters accepts values in the [-1,1]. If v is 1, the robot walks forward straight; if v is -1, the robot walks backward straight; if v is 0, robot doesn't move straight. If w is 1, the robot turn left; if w is -1, the robot turn right; if w is 0, robot doesn't turn. Unfortunately, this movements can't be combined and only one of them is active at the same time.

Actually, `Body` doesn't this work directly but it activates and modulates two lower level components: `GoStraight` and `Turn`, as we can see in figure 21. When v is different to 0, it

deactivates `Turn` component if it was active, modulates and activates `GoStraight` component. When *w* is different to 0, it deactivates `GoStraight` and activates `Turn`.

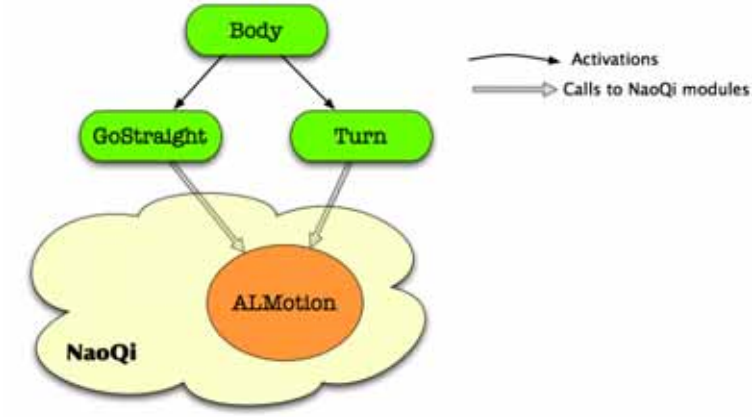

Fig. 21. `Body` component and its lower level components, which comumnicate with NaoQi to move the robot.

5.2.2 `Head` component

`Body` component makes move all the robot but the robot head. Robot head is involved in the perception and attention process and can be controlled independiently from the rest of the robot. The robot head is controlled by the `Head` component. This component, when active, can be modulated in velocity and position to control the pan and tilt movement. While the head control in position is quite simple (it sends motion commands to `ALMotion` to set the joint to the desired angle), the control in velocity is more sophisticated. We developed a *PID controller* to adjust the movement speed. The modulation parameter for this type of control, in range [-1,1] in each pan and tilt, is taken as the input of this controller. The value -1 means the maximun value in one turn sense, 1 in the other sense, and 0 means to stop the head in this axe.

5.2.3 Fixed Movement behavior

The last component involved in actuation is the `FixMove` component. Sometimes it is required to perform a fixed complex movement composed by several joint positions in determined times. For example, when we want that robot kicks the ball we have to made a coordinate movement that involves all the body joints and takes several seconds to complete. These movements are coded in several files, one for each fixed movement, that describe the joints involved in the movement, the positions and when these positions should applied. Lets look an example of this file:

```
Movement_name
name_joint_1 name _joint_2 name _joint_3 ... name _joint_n
angle_1_joint_1 angle_2_joint_1 angle_3_joint_1 ... angle_m1_joint_1
angle_1_joint_2 angle_2_joint_2 angle_3_joint_2 ... angle_m2_joint_2
angle_1_joint_3 angle_2_joint_3 angle_3_joint_3 ... angle_m3_joint_3
...
angle_1_joint_n angle_2_joint_n angle_3_joint_n ... angle_mn_joint_n
time_1_joint_1 time_2_joint_1 time_3_joint_1 ... time_m1_joint_1
time_1_joint_2 time_2_joint_2 time_3_joint_2 ... time_m2_joint_2
time_1_joint_3 time_2_joint_3 time_3_joint_3 ... time_m3_joint_3
...
time_1_joint_n time_2_joint_n time_3_joint_n ... time_mn_joint_n
```

In addition to the desired fixed movement, we can modulate two parameters that indicates a walking displacement in straight and side senses. This is useful to align the robot with the ball when kicking the ball. If this values are not zero, a walk preceed the execution of the fixed movement.

As we have just introduced, we use this component for kicking the ball and for standing up when the robot is pushed and falls to the floor.

5.3 Face Ball behavior

`FaceBall` component tries to center the ball in the image taken from the camera. To achieve this goal, when active, this component activates both `Perception` and `Head` components, as we see in figure 22.

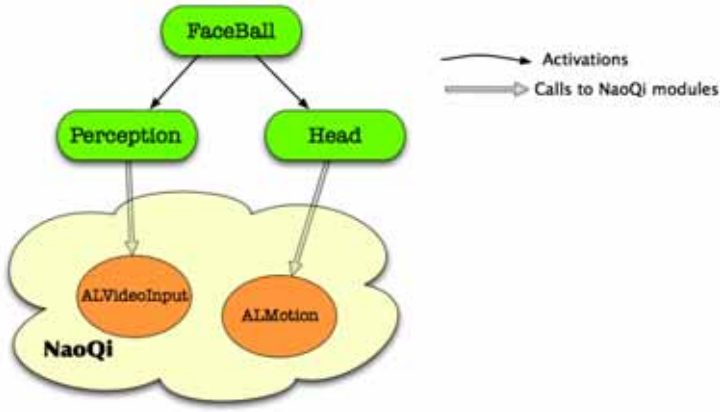

Fig. 22. `FaceBall` component.

This component activates `Perception` and `Head` while it is active. It modulates `Perception` to detect the ball. In its `step()` function, it simply takes the output of the perception component and uses this value as the input of the `Head` component. These values are in the [-1,1] range. When the ball is centered, the X and Y value of the ball are 0, so the head is stopped. If the ball is in the extreme right, the X value, 1, will be the modulation of the pan velocity, turning the head to the left. Here is the code of the `step()` funtion of `FaceBall`.

```
void
FaceBall::step(void)
{
        perception->step();

        if (isTime2Run())
        {
                head->setPan( perception->getBallX());
                head->setTilt( perception->getBallY() );
        }

        head->step();
}
```

5.4 Follow Ball behavior

The main function of `FollowBall` component is going to the ball when it is detected by the robot. This component activates FaceBall and Body components. The modulation of the Body component is the position of the robot head, that is tracking the ball. Simplifying, the code of the step function of this component is something like this:

```
void
FollowBall::step(void)
{
        faceball->step();

        if (isTime2Run())
        {
                float panAngle = toDegrees(headYaw);
                float tiltAngle = toDegrees(headPitch);

                if(panAngle > 35) body->setVel(0, panAngle/fabs(panAngle));
                else body->setVel(1, 0);

        }

        body->step();
}
```

Fig. 23. `FollowBall` component.

5.5 Search Ball behavior

The main function of `SearchBall` component is search the ball when it is not detected by the robot. This component introduces the concept of finite state machine in a component.

When this component is active, it can be in two states: *HeadSearch* or *BodySearch*. It starts from *HeadSearch* state and it only moves the head to search the ball. When it has scanned all the space in front of the robot it transitates to *BodySearch* state and the robot starts to turn while it is scanning with the head. In any state, `SearchBall` component modulates `Perception` component in order to periodically change the active camera.

Depending on the state, the activation tree is different, as we can see in figure 24. At start, the active state is *HeadSearch*. In this state only `Head` component is active. During this state, `Head` component is modulated directly from `SearchBall` component. It starts moving the head up and it continues moving the head to scan the space in front of the robot. When this scan is completed, it transitates to *BodySearch* state. In this state, `Body` component is also activated in order to make turn the robot in one direction.

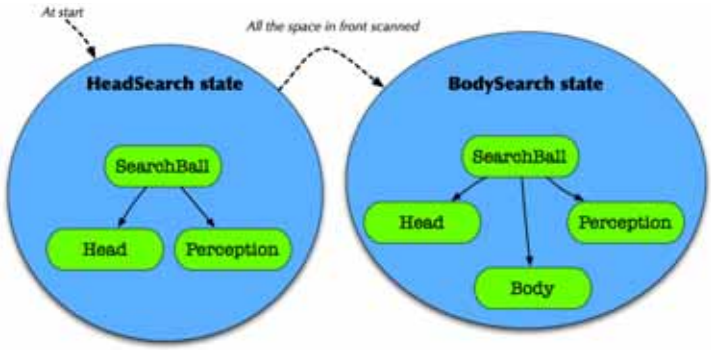

Fig. 24. The two states that this component can be and the activation tree in each state.

This component does not use the `Perception` component to get information about the ball presence. This component only manages the robot movement and the camera selection. Any other component has to activate `SearchBall` and `Perception` components, and stop `SearchBall` once the ball is found. Next we will see which component do this work.

5.6 Search Net behavior

This behavior is implemented by the `SearchNet` component is used to search the net where the robot must kick the ball to. It activates `Head` and `Perception` components. Its work is divided in two states: *Scanning* and *Recovering*.

When the *Scanning* state starts, the head position is stored (it is supposed to be tracking the ball) and the robot modulates `Perception` component to detect the nets instead of the ball. It has not sense continuing doing processing to detect the ball if now it is not the interesting element, saving processing resources.

While *Scanning* state, this component also modulates Head component to move the head along the horizont, searching the ball. When this component is active, the robot is stopped, and we can suppose where is the horinzont. If the scanning is complete, or the net is detected, this component tansitates to the *Recovering* state.

In the *Recovering* state the `Perception` component is modulated to detect the ball, and the head moves to the position stored when *Scanning* state started.

5.7 Field Player behavior

The `Player` component is the root component of the forward player behavior. Its functionality is decomposed in five states: *LookForBall, Approach, SeekNet, Fallen and Kick*. These five states encode all the behavior that makes the robot play soccer.

In *LookForBall* state, `Player` component activates SearchBall and Perception components, as is shown in figure 25. For clarity reasons, in this figure we don't display all the activation tree but the components that `Player` component directly activates.

When the `Perception` components indicates that the ball is detected, this component transitates to *Approach* state. It deactivates SearchBall State, and the ball is supposed to be in the active camera. In this state is activated the FollowBall component in order to make the robot walk to the ball.

It is common that the robot initially detects the ball with the upper camera, and it starts the approach to the ball using this camera. When the ball is nearer than one meter, it can't follow it with the upper camera bacause of the neck limitations and the ball is lost. It transitates to the *LookForBall* state again and starts searching the ball, changing the camera. When the ball is detected with the lower camera, it continues the approaching with the right camera.

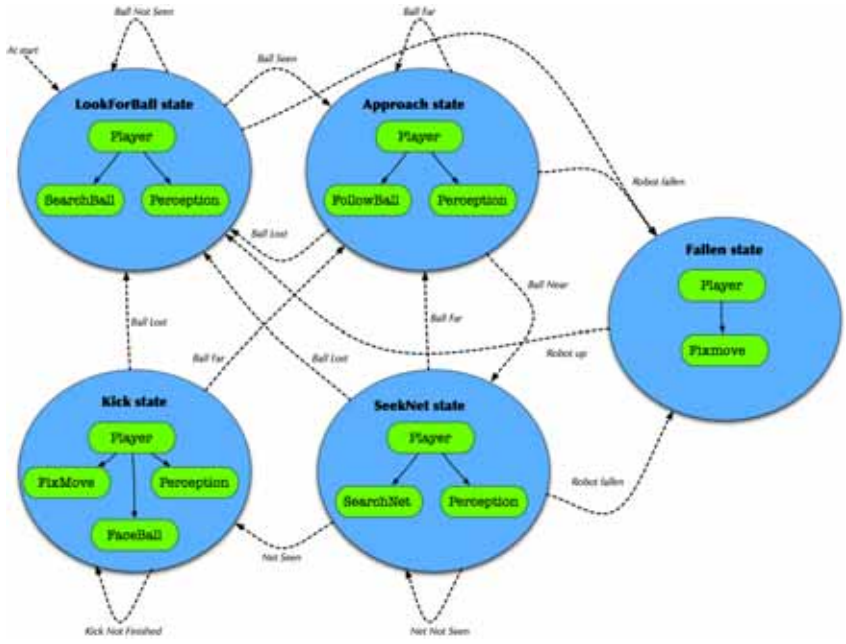

Fig. 25. `Player` component finite state machine with its corresponding activation tree.

When the ball is close to the robot, the robot is ready to kick the ball, but it has to detect the net first in order to select the addecuate movement to score. For this reason, the `Player` component transitates to *SeekNet* state, activating `SearchNet` component.

Once detected the net, the robot must kick the ball in the right direction according to the net position. Player component makes this decission in the *Kick* state. Before activating `FixMove` component, `Player` component ask to `Perception` component the 3D ball position. With this information, it can calculate the displacement needed by the selected kick to perform this kick correctly. Once activated the `FixMove` component, the robot performs the kick.

In this state, this component also activates `FaceBall` component to track the ball while the robot is kicking the ball.

The last state is *Fallen*. This component goes to transitates to this state when the robot falls to the floor. It activates `Fixmove` component and modulates it with the right movement to make it getting up.

These are the components needed for the forward player behavior. In next sections we will explain the tools developed to tune, configurate and debug these components, and also the components developed to make a complete soccer player.

6. Tools

In previous section we described the software that the robot runs onboard. This is the software needed to make the robot perform any task. Actually, this is the only software that is working while the robot is playing soccer in an official match, but some work on calibrating and debugging must be done before a game starts.

We have developed a set of tools useful to do all the previous work needed to make the robot play. *Manager* application contains all the tools used to manage the robot. This application runs in a PC connected to the robot by an ethernet cable or using wireless communications.

To make possible the communication between the robot and the computer we have used the SOAP protocol that NaoQi provides. This simplify the development process because we do not have to implement a socket based communication or anything similar. We only obtain a proxy to the NaoQi module that contains our software, and we make calls to methods to send and receive all the management information.

Next, we will describe the main tools developed inside the *Manager*. These tool let to debug any component, tune the vision filters and the robot movements.

6.1 Component debugging tool

Inside the *Manager* we can debug any component individually. We can activate a component, modulate it and change its frequency. It is also possible to take measures related to the CPU consumption.

In figure 26 we show the GUI of this tool and how the `Turn` component is debugged. We can activate independiently using a checkbox. We can also configurate the frequency, in this case it is set to run at 5 Hz. We use the slider to modulate this component setting its input in the [-1,1] range. In the figure the modulation is 0, so the robot is stopped. Finally, we can obtain the mean, maximum and minimum CPU consumption time in each iteration.

Fig. 26. Component debugging tool and how the `Turn` component is debugged.

6.2 Perception tool

The environment conditions are not similar in every place the robot must work. Even in the same place, the conditions are not similar along the day. For this reason to calibrate the camera characteristics and the color definitions is essential to face these changes in the ligt conditions.

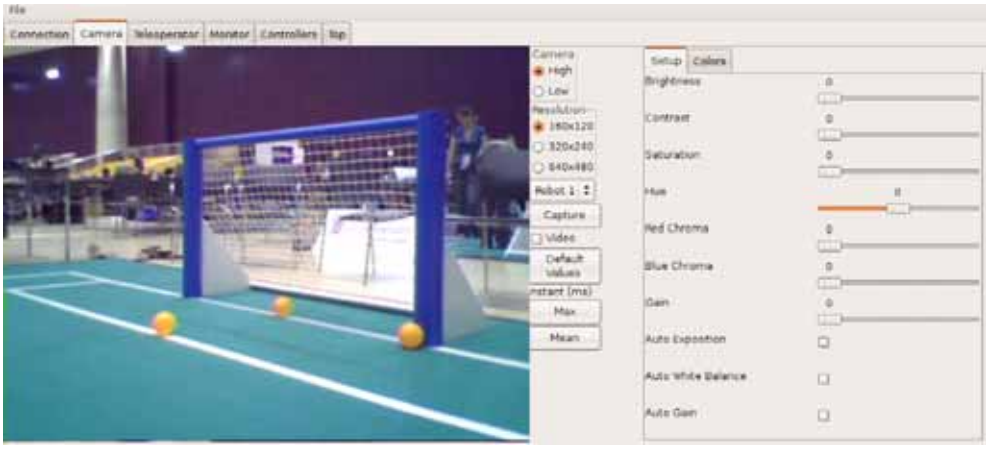

Fig. 27. Camera values tunning.

The *Manager* contains some tools to do this calibratarion. Figure 27 shows the tool used to calibrate the camera values (brightness, contrast, …). Each relevant element to the robot has

a different color, as we explained in section 5.1.1. These colors and the recognization values are set using the tools shown previously in figure 15.

6.3 Fixed Movement tool

In some situations the robot must perform a fixed movement. To kick the ball or to get up, the robot needs to follow a sequence of movements that involves many joints. It is difficult to create these movements without a specific tool.

We have implemented a tool to create sequences of movemenet. For each sequence we have to specify the joints involved, the angles that each joint has to be set and the time these angles are set. This was explained when we presented the component that performs these movements, `Fixmove` (section 5.3), where we shown an example of the file that stores the sequence. The goal of this tool, shown in figure 28, is to create these sequence files.

Using this tool we can create the sequence step by step. En each step we define the duration and we change the joint values needed in this step. We can turn off a single joint and move it manually to the desired position, and then get that position. This makes easy to create movement using this process for each step and for each joint.

We have created three types of kicks using this tool. Each kick can be done simetrically with both legs, then we really have six kicks. Also, we have created two movements to make the robot get up from the floor.

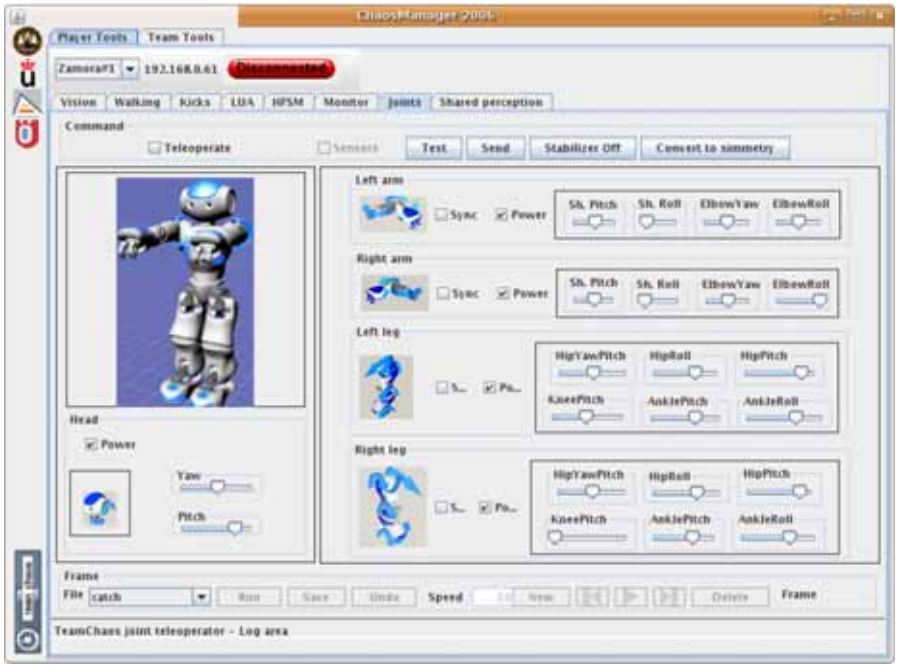

Fig. 28. Fixed movement sequence generation tool.

7. Experiments

In this behavior we have presented our behavior based architecture and a complete soccer player application using it. In this chapter we will show the experiments carried out during and after its development.

7.1 First behavior architecture attempt

Not always the first steps are the right ones. In this architecture design, the proposed solution wasn't the first approximation we took. At initial we tried to exploit all the benefits that NaoQi provides. This software element lets to decompose our application functionality in modules which cooperate among them to achieve a goal. Each module perfoms some processing task and sends data to other modules. This would lets to implement our architecture in a natural way using this approximation. NaoQi has a funtionality to start and stop calling iteratively a method, using a callback to a periodic clock event. This solves the execution cycle to call `step()` method iteratively. Communications among modules are solved by the SOAP messages mechanism that NaoQi provides. We also could use ALMemory as a blackboard where all the information from sensorial components and all the modulations to actuation modules are registered and taken. Even callbacks can be set up in each module to be called each time an interesting value in this blackboard changes. In fact, this was the first approach we took to design our architecture. Unfortunately, and intensive use of these mechanisms had a big impact in NaoQi performance and some real time critical tasks were severely affected. One of these real time critical tasks is movement generation. When the performace in this task was poor, the movement was affected and the robot fallen to floor.

7.2 General behavior creation process using the proposed architecture

The final design tried to use as less NaoQi mechanisms as possible. We use NaoQi to access sensors and actuators, but all the communication via SOAP messages are reduced to the minimun possible. ALMemory as a blackboard was discarded and callbacks to data are used only to the essential information generated by NaoQi that are useful to our tasks. Although we have taken this decission, some of the NaoQi functionality is still essential to us. We use NaoQi for accessing to the camera images or for walking generation. We are not interesed in developing our own locomotion mechanism because this is being improved continuosly by the robot manufacturer and we want to concentrate in hich level topics.

With the changes from the initial to the final version we obtained better performance and we avoid to affect NaoQi locomotion mechanism. Althought our software wants to consume too much processing time, only our software will be affected. This is a redical improvement with respect the initial version.

The robot soccer behavior is itself an experiment to test the proposed behavioral architecture. Using this architecture, we have developed a complete behavior able to cope with the requirements that a RoboCup match imposes. We have shown some of the characteristics of this robot architecture during this process:

- `FaceBall`, `SearchNet` and `SearchBall` components reuses the component `Head`. In this way we have shown how a component can be reused only changing its modulation.
- Only `Perception`, `GoStraight` and `Turn` components face to the complexity of the robot hardware, which let in the other levels to ignore this complexity.
- Component activations and deactivations, for example in section 5.5, let to have diferent behaviors on the robot.

7.3 Forward soccer player test

The final experiment is the real application of this behavior. We have tested it at RoboCup 2009 in Graz. Before the test we had to use the tools described in section 6 to calibrate the colors of the relevant elements in the environment. Once tuned, the robot is ready to work. This sequence has been extracted from a video which full version may be visualized at http://www.teamchaos.es/index.php/URJC#RoboCup-2009.

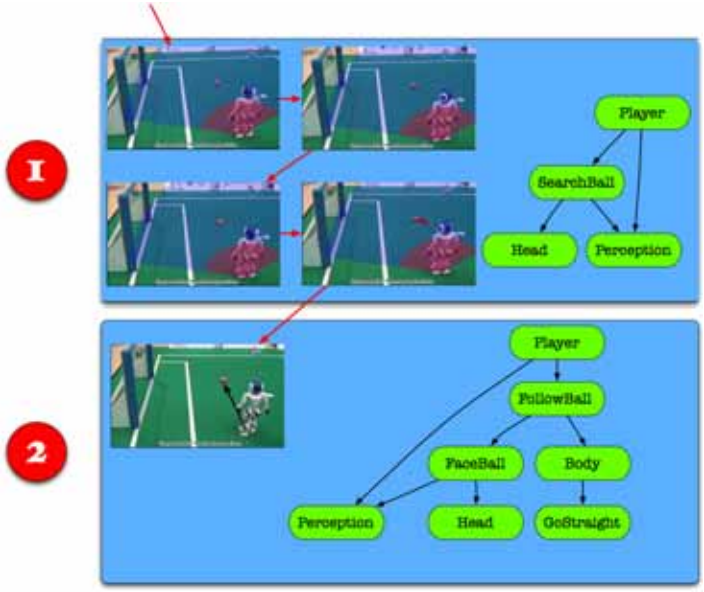

Fig. 29. Ball searching sequence.

Figure 29 shows a piece of an experiment of the soccer player behavior. In this experiment the robot starts with total uncertainty about the ball. Initially, the Player component is in LookForBall state and it has activated the SearchBall component to look for the ball. SearchBall component uses first the bottom camera. The shadow area respresents the camera coverage, where the red area represents the bottom camera coverage and the blue area the upper camera coverage. In the two first images the robot is scanning the nearest space with the bottom camera and it doesn't find any ball. Once completed the near scan, SearchBall component modulates Perception component in order to change the camera, covering the blue marked area. Player component is continously asking Perception

component for the ball presence, and when the ball is detected in the image (fourth image in the sequence), SearchBall component is deactivated and FollowBall component is activates, approaching to the ball (last image in the sequence). Take note that in this example, the upper camera is now active.

FollowBall component activates FaceBall component to center the ball in the image while the robot is approaching to the ball. FollowBall activates Body to approach the ball. As the neck angle is less than a fixed value, i.e 35 degrees (the ball is in front of the robot), Body activates GoStraight component in order to make the robot walk straight.

The approaching to the ball, as we said before is made using FaceBall component and Body component. Note that in any moment no distance to the ball is take into account. Only the head pan is used by the body component to approach the ball.

In figure 30, while the robot the robot is approaching to the ball, it has to turn to correct the walk direction. In this situation, the head pan angle is higher than a fixed value (35 degrees, for example) indicating that the ball is not in front of the robot. Immediately, after this condition is true, FollowBall modulates to Body so the angular speed is not null and forward speed is zero. Then, Body component deactivates GoStraight component and activates Turn Components, which makes the robot turn in the desired direction.

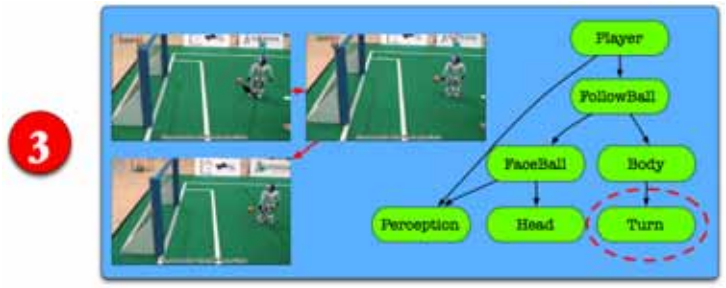

Fig. 30. Ball appraching modulation to make the robot turn.

The robot reached the ball when it is walking to the ball, the bottom camera is active, the head tilt is higher than a threshold, and the head pan is low. This situation is shown in the first image in the figure 31. In that moment, the robot has to decide which kick it has to execute. For this reason, the net has to be detected. In the last image, the conditions to kick the ball are held and the player component deactivates FollowBall component and activates the SearchNet component. The SearchNet component has as output a value that indicates if the scan is complete. The Player component queries in each iteration if the scan is complete. Once completed, depending on the net position (or if it has been detected) a kick is selected. In the second image of the same figure, the blue net is detected at the right of the robot. For this test we have created 3 types of kicks: front, diagonal and lateral. Actually, we have 6 kicks available because each one can be done by both legs. In this situation the robot selects a lateral kick with the right leg to kick the ball.

Before kick the ball, the robot must be aligned in order to situate itself in the right position to do an effective kick. For this purpouse, the player component ask to the Perception module the ball position in 3D with respect the robot. This is the only time the ball position is estimated. The player components activates Fixmove component with the selected kick and a lateral and straight alignement. As we can see in third and fourth image, the robot moves on its left and back to do the kick.

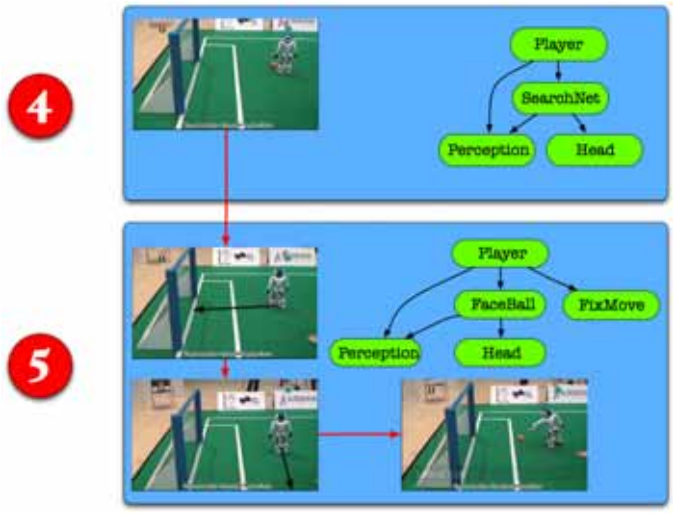

Fig. 31. Search net behavior and kick.

While the kick is performing and after the kick, FaceBall component is activeted to continue traking the ball. This spped up the recovering after the kick and sometimes it is not needed to transitate to the searching ball state, but the approaching to the ball state.

This experiment has been carried out at the RoboCup 2009 in Graz. This behavior was tested in the real competition environment, where the robot operation showed robust to the noise produced by other robots and persons.

7.4 Any Ball challenge

In RoboCup 2009 competitions we also took part in the Any Ball Challenge. The goal was to kick ball differents to the orange official one. To achieve this goal we changed the Perception component to detects the non-green objects under the orizont that seemed like balls. In the figure 32 (and in the video we mentioned before) we can see how the robot was able to kick differents balls.

Fig. 32. Any Ball Challenge. The robot must detect and kick hetereogeneuos balls.

7.5 Camera switching experiment

In the experiment described in section 7.2, between the instant 3 and 4 sequence, a camera switch has made. For clarity, lets use another texperiment to explain this change with another sequence. In figure 33, the robot is approaching to the ball using the upper camera. The Player component has activated FollowBall component and the robot is walking straight to the ball using the upper camera.

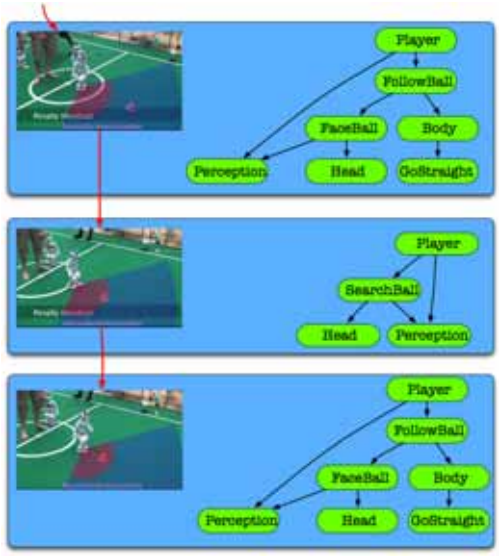

Fig. 33. Camera switching.

When ball enters in the red area and the tilt head has the maximun value, ball dissapears from the image taken by the upper camera. Then, the Player components deactivates FollowBall components (and their activated components in cascade) and activates SearchBall component. SearchBall component always starts with the bottom camera activated and moves the head up. In few steps (maybe 1 or two are enough) ball is detected again. Player component deactivates SearchBall component and activates FollowBall component, that starts with the bottom camera selected. The camera change is made.

8. Conclusions

In this chapter we have proposed a robotic behavior based architecture. With this architecture we can create robotics behaviors. The behavior arises from a cooperative execution of iterative processing units called components. These units are hierarchically organized, where a component may activate and modulate another components. In every moment, there are active components an latent components that are waiting for be activated. This hierarchy is called *activation tree*, and dinamically changes during the robot operation. The components whose ouput is not needed are deactivated in order to save the limited resources of the robot.

We can use this behavior architecture for create any robotic behavior. In this chapter we have shown how the behaviors are implemented within the architecture. We have created a forward player behavior to play soccer in Standar Platform League at RoboCup. This is a dynamic environment where the conditions are very hard. Robots must react very fast to the stimulus in order to play soccer in this league. This is an excellent test to the behaviors created within our architecture.

We have developed a set of component to get a forward soccer player behavior. These components are latent until a component activate it to use it. These component have a standar modulation interface, perfect to be reused by several component without any modification in the source code or to support multiple different interfaces.

Perception is done by the `Perception` component. This component uses the camera to get different stimulus form the environment. The stimulus that it detect are the ball and the net in image coordinates, the ball in ground coordinates and the goal in camera coordinates. This component only perceives one stimulus at the same time, saving the limited resources.

Locomotion is done by the `Body` component, that uses `Turn` or `GoStraight` components alternatively to make the robot walk to the ball. This component is modulated by introducing a lineal or rotational speed. We found this interface is more appropiate that the one provided by the NaoQi high level locomotion API to create our behaviors.

The head movement is not managed from the Body components because it is involved in the perception process and we think that it is better to have a sepparate component for this element. This component is the `Head` component, and moves the robot's neck in pan and tilt frames.

Robot also moves by performing a sequence of basic joint-level movements. This functionality is obtained from the `FixMove` component execution. This is useful to create kicking sequences or getting up sequences.

These components use the NaoQi API directly and are usually in the lower level of the activation tree. They are iterative components and no decision are taken in each step. There are other more complex components. These components activate other components and may vary dinamically the set of components that it activates.

The `FaceBall` component activates `Perception` and `Head` component in order to center the ball in the image. This component is very important, because when it is activated, we can assume that the ball position is where the ball is pointing at. This is used by the `FollowBall` component. This components uses `Body` component to make the robot move ahead when the neck pan angle is small, and turning when it is big. There is not need to know the real distance or angle to the ball, only the neck pan angle.

When the ball is not detected in the image, the `SearchBall` component activates and modulates `Head` and `Body` components to detect the ball. In the same way, the SearchNet component component uses the `Head` component to look for the goals, in order to calculate the right kick to use in order to score.

The higher level component is the `Player` component. This component is implemented as a finite state machine and activates the previously described components in order to obtain the forward player behavior.

This behavior, created with this architecture, has been tested in the RoboCup environment, but it is not limited to it. We want to use this architecture to create robot behaviors to solve another problems out of this environment.

9. References

Thrun, S.; Bennewitz, M.; Burgard, W.; Cremers, A. B.; Dellaert, F.; Fox, D.; Hahnel, D.; Rosenberg, C. R.; Roy, N.; Schulte, J; Schulz, D. (1999). *MINERVA: A Tour-Guide Robot that Learns*. Kunstliche Intelligenz, pp. 14-26. Germany

Reid, S. ; Goodwin, R.; Haigh, K.; Koenig, S.; O'Sullivan, J.; Veloso, M. (1997*). Xavier: Experience with a Layered Robot Architecture*. Agents '97, 1997.

Stoytchev, A.; Arkin, R. (2000). *Combining Deliberation, Reactivity, and Motivation in the Context of a Behavior-Based Robot Architecture*. In Proceedings 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation. 290-295. Banff, Alberta, Canada. 2000.

Arkin, R. (1989). *Motor Schema Based Mobile Robot Navigation.* The International Journal of Robotics Research, Vol. 8, No. 4, 92-112 (1989).

Saffiotti, A. ; Wasik, Z. (2003). *Using hierarchical fuzzy behaviors in the RoboCup domain*. Autonomous robotic systems: soft computing and hard computing methodologies and applications. pp. 235-262. Physica-Verlag GmbH. Heidelberg, Germany, 2003.

Lenser, S.; Bruce, J.; Veloso, M. (2002). *A Modular Hierarchical Behavior-Based Architecture*, Lecture Notes in Computer Science. RoboCup 2001: Robot Soccer World Cup V. pp. 79-99. Springer Berlin / Heidelberg, 2002.

Röfer, T.; Burkhard, H. ; von Stryk, O. ; Schwiegelshohn, U.; Laue, T.; Weber, M.; Juengel, M.; Gohring D.; Hoffmann, J.; Altmeyer, B.; Krause, T.; Spranger, M.; Brunn, R.; Dassler, M.; Kunz, M.; Oberlies, T.; Risler, M.; Hebbela, M.; Nistico, W.; Czarnetzkia, S.; Kerkhof, T.; Meyer, M.; Rohde, C.; Schmitz, B.; Wachter, M.; Wegner, T.; Zarges. C. (2008). *B-Human. Team Description and code release 2008. Robocup 2008*. Technical report, Germany, 2008.

Calvo, R.; Cañas, J.M.; García-Pérez, L. (2005*). Person following behavior generated with JDE schema hierarchy*. ICINCO 2nd Int. Conf. on Informatics in Control, Automation and Robotics. Barcelona (Spain), sep 14-17, 2005. INSTICC Press, pp 463-466, 2005. ISBN: 972-8865-30-9.

Cañaas, J. M.; and Matellán, V. (2007). *From bio-inspired vs. psycho-inspired to etho-inspired robots*. Robotics and Autonomous Systems, Volume 55, pp 841-850, 2007. ISSN 0921-8890.

Gómez, A.; Martínez, H.; (1997). *Fuzzy Logic Based Intelligent Agents for Reactive Navigation in Autonomous Systems*. Fitth International Conference on Fuzzy Theory and Technology, Raleigh (USA), 1997

Loetzsch, M.; Risler, M.; Jungel, M. (2006*). XABSL - A pragmatic approach to behavior engineering*. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006), pages 5124-5129, Beijing, October 2006.

Denavit, J. (1955). *Hartenberg RS. A kinematic notation for lower-pair mechanisms based on matrices*. Transactions of ASME 1955;77: 215–221 Journal of Applied Mechanics, 2006.

Herrero, D. ; Martínez, H. (2008). *Embedded Behavioral Control of Four-legged Robots*. RoboCup Symposium 2008. Suzhou (China), 2008.

Akin, H.L.; Meriçli, Ç.; Meriçli, T.; Gökçe, B.; Özkucur, E.; Kavakhoglu, C.; Yildiz, O.T. (2008). *Cerberus'08 Team Report*. Technical Report. Turkey, 2008.

Chown, E.; Fishman, J.; Strom, J.; Slavov, G.; Hermans T.; Dunn, N.; Lawrence, A.; Morrison, J.; Krob, E. (2008). *The Northern Bites 2008 Standard Platform Robot Team*. Technical Report. USA, 2008.