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/

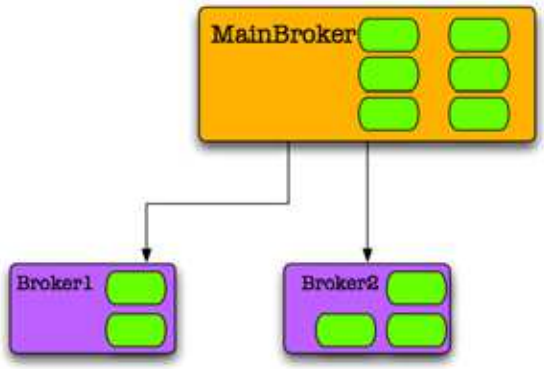

Fig. 2. Brokers tree.

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.

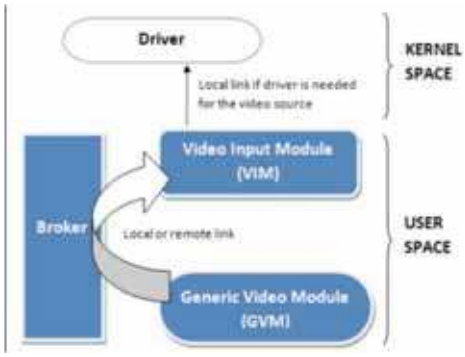

Fig. 4. NaoQi vision architecture overview.

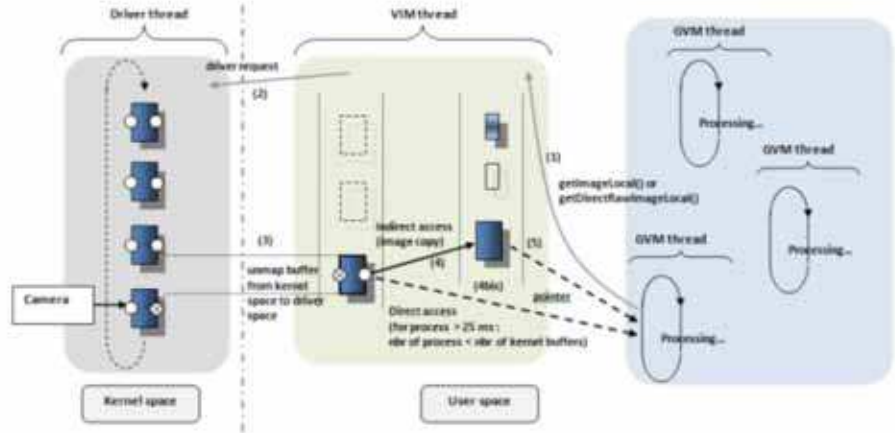

Fig. 5. Access to an image in the NaoQi framework.

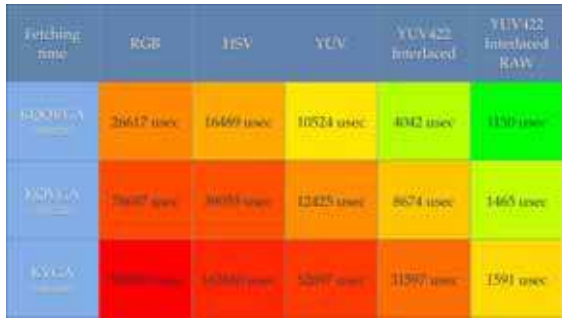

Fig. 6. Access time to the image depending on resolution and space color. Last column is *direct raw* mode.

- 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.

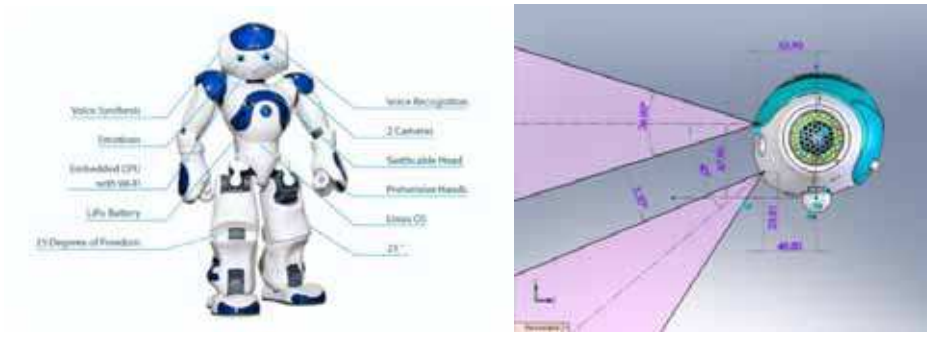

Fig. 7. Aldebaran Robotics’ Nao Robot.

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.

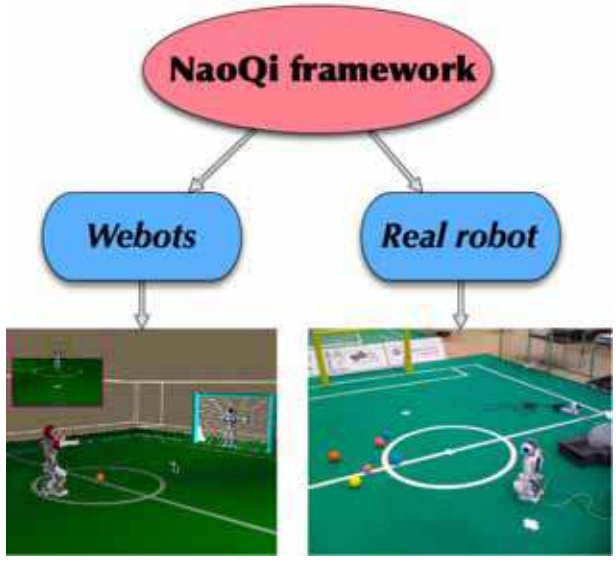

Fig. 8. Simulated and real robot.

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.

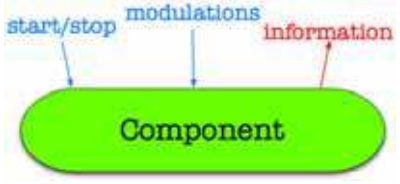

Fig. 9. Component inputs and outputs.

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.

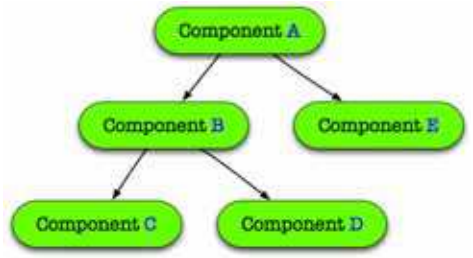

Fig. 10. *Activation tree* composed by several components.

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.

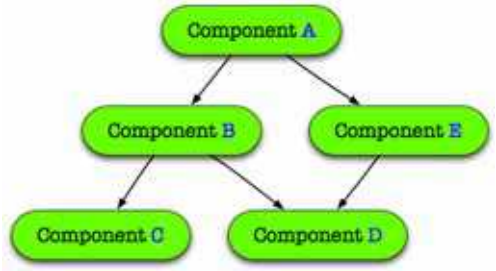

Fig. 11. *Activation tree* where B and D activates D component.

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.

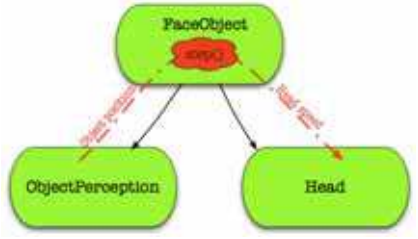

Fig. 12. Activation tree with two low level components and a high level components that modulates them.

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.

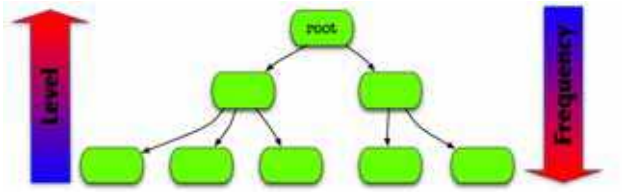

Fig. 13. Activation tree with a root component in the higher level. As higher is the level, lower is the frequency.

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.

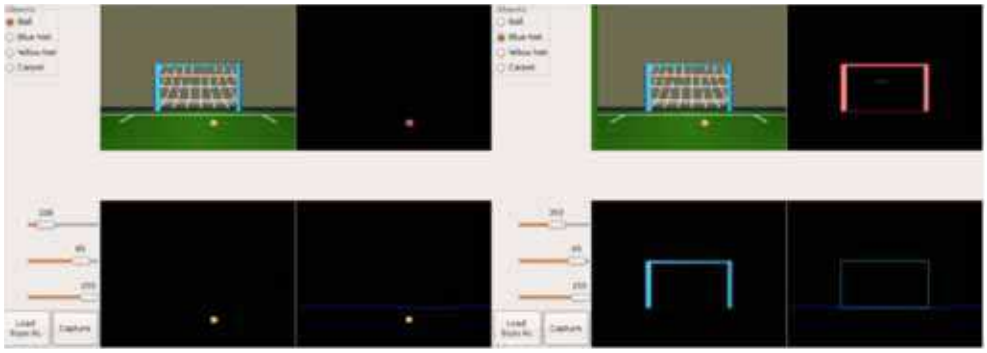

Fig. 15. Element detection process.

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.

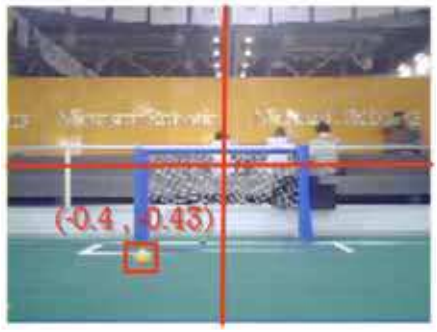

Fig. 16. Tuple containing the ball position.

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).

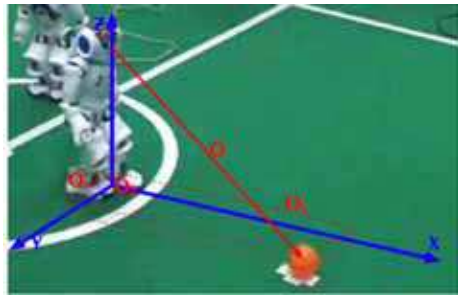

Fig. 17. Element 3D position and the robot axes.

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.

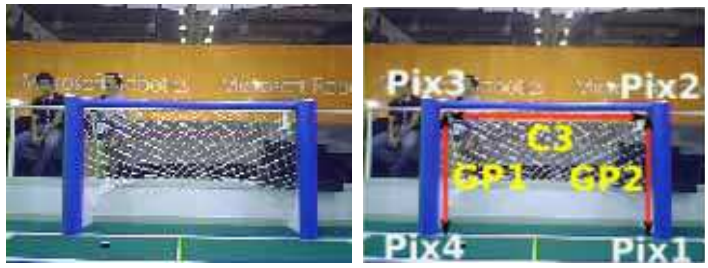

Fig. 18. Goal detection.

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.

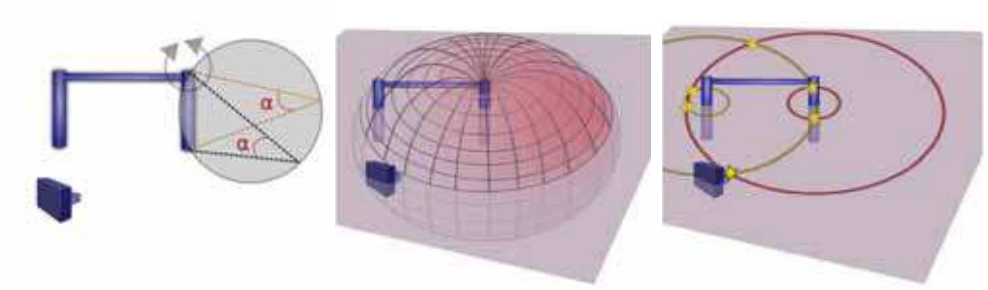

Fig. 19. Camera 3D position estimation using a 3D thorus built from the perception.

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.