Follow ball behavior for an humanoid soccer player

Francisco Martín, Carlos E. Agüero and José María Cañas
{fmartin,caguero,jmplaza}@gsyc.es.

***Abstract*—This paper describes a simple behavior based approach that makes an humanoid robot search and go to a ball. This behavior is composed by several execution units called *components*, which are activated and modulated by another ones in an activation tree. The main goals of this behavior are: Reactive control which hides robot complexity, component reutilisation, communications among pairs and component activation/deactivation/modulation by a parent component. This behavior has been tested in the RoboCup domain, in simulator and in real robot, and it is the initial behavior for a complete soccer player.**

***Index Terms*—Autonomous robots, soccer, behavior architectures**

I. 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 [4], 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 accomplish 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.

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 in the *Standard Platform League*. In this league all the teams use the same robot. This is a key factor because it makes to effort in the software aspect rather than in the hardware. This is why many people call this league a *software league*. Up to 2007 the robot chosen 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, and this is the main difference with respect Aibo, that is a quadruped robot. This fact has a big impact in the way the robot moves and its stability while moving. The size of both robots are not the same. Aibo was 15 cm tall and Nao is about 55 cm tall. This is a big difference in the way the perception is made. In Aibo the perception is 2D because the camera was very close to the floor. Robot Nao may perceive in 3D because the camera is at a higher position.

Fig. 1. Nao robot playing soccer

Nao robot is a very attractive comercial platform, where changes in hardware are not possible. Manufacturer provides a complete API to move the robot and access the sensors. Despite of this, many problems must to be solved before having a fully featured soccer player. First of all, the robot have to obtain information from the environment using mainly the cameras. It has to detect the ball, goals, lines and the other robots. With this information the robot has to self-localise and decide the next action: move, kick, search another object, etc. All these processes are controlled by a behavior architecture, that organizes how and when the information is perceived, and how to generate the actions to react to any situation.

During RoboCup matches the most basic behavior is to perceive the ball and walk close to it. This behavior is very basic, but all teams are trying to achieve this goal in this new platform. Every team has experience in these type of behaviors. They have solved this problem for quadruped robots in the Four Legged League, but its constraints and requisites are different for this robot. In this work we propose

Robotics Lab. Universidad Rey Juan Carlos.
This work has been partially founded by the Ministerio de Ciencia y Tecnología, in the COCOGROM project: DPI2007-66556-C03-01, by the Comunidad de Madrid in the RoboCity 2030 project: S-0505/DPI/0176.

a FollowBall behavior that tries to achieve this goal. This preliminary work will let us to start designing a behavior based architecture approach to generate a soccer player behavior by solving this problem. The goal is to develop a complete architecture where behavior can be easily generated by reusing components. Furthermore, behaviors must be reactive in order to be adequate to make a robot play soccer in the Standard Platform League at RoboCup, where games are very dynamic.

This paper is organised as follows: First of all, in section II we will present relevant previous works which are also focused in robot behavior generation and following behaviors. In section III we will present the Nao and the programming framework provided to develop the robot applications. In the IV we will describe the architecture where the components are organized and their properties. An example of these components and a simple functional hierarchy is shown in section V. In section VII we will present the conclusion of this work.

II. Related work

In this section we will describe previous works that tries to solve robot behavior generation and following behaviors. First of all, we will describe classic approaches to generate robot behaviors. These approaches were successfully tested in wheeled robots. Next, we will present more related approaches to RoboCup domain. Finally, we will describe a following behavior that uses a very related approach to the one used in this work.

There are a lot of approaches which tries to solve the behavior generation problem. One of the first successful works on mobile robotics is Xavier [1]. The architecture used in this work was made up of four layers: obstacle avoidance, navigation, path planning and task planning. The behavior arises from the combination of these separate layers, each one with an specific task and priority. The main difference with respect our work is this separation. In our work there are not layers with a specific task, but the tasks are decomposed into components in different layers.

Another approach is [2], where it was proposed an hybrid architecture where behavior were divided into three components: deliberative planning, reactive control and motivation drives. Deliberative planning did navigation tasks. Reactive control provided 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 that respond to the external stimuli. Motivation drives were responsible for monitoring the robot behavior. This work had in common with our in the idea of behavior decomposition into smaller behavioral units.This behavior unit was explained in detail in [3].

In the RoboCup domain, Saffiotti [5] presented the *ThinkingCap* architecture. This architecture was based in a fuzzy approach, extended in [10]. The perceptual and global modelling components manage information in a fuzzy way and were used to generate the next actions. This architecture was tested in the four legged league RoboCup domain. This work is important for the work presented in this paper because this was the previous architecture used in our RoboCup team. Using this approach we developed a follow ball behavior. This behavior was made using LUA programming language and reads ball position from local perception data maintained by a perception module and set velocity values to the locomotion module.

A hierarchical behavior-based architecture was presented in [6]. This architecture was divided in 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.

Another successful approach [7] divides their architecture in four levels: perception, object modelling, behavior control and motion control. The execution starts in the upper level perceiving the environment and finishes at low level sending 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 [11], that was interpreted at runtime and let change and reload the behavior during the robot operation.

Finally, in [8] a follow person behavior was developed by using an architecture called JDE [9]. This reactive behavior arises from the activation/deactivation of components called schemes. This approach has several similitudes with the one presented in this work.

III. Nao and NaoQi framework

The behavior developed in this work has been tested using the Nao robot. Nao 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.

Behavior design must be implemented in software. In this section we will describe the underlying software mapping the robot hardware. FollowBall behavior has been programmed on top of it. This underlying software is called NaoQi, and provides a framework to develop applications in C++ and Python.

NaoQi is a distributed object framework which allows several distributed binaries, each containing several software modules to communicate together. Robot functionality is encapsulated in software modules, so we can communicate to specific modules in order to access sensors and actuators.

Each binary, also called *broker*, runs independently and is attached to an address and port. Every broker is able to run in the robot (cross compiled) or in the computer. In this way we can develop a complete application that is composed by several brokers running in a computer and others 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 the robot, that 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 modules to accomplish their tasks.

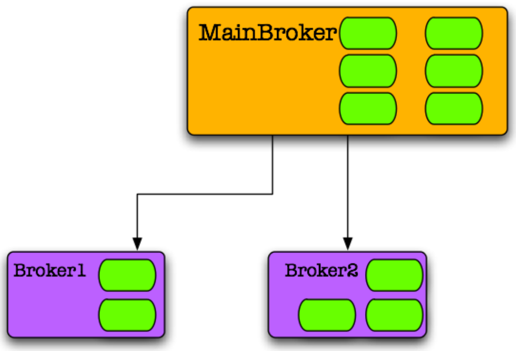

Fig. 2. MainBroker (orange) and others brokers(violet). Each one with their modules (green)

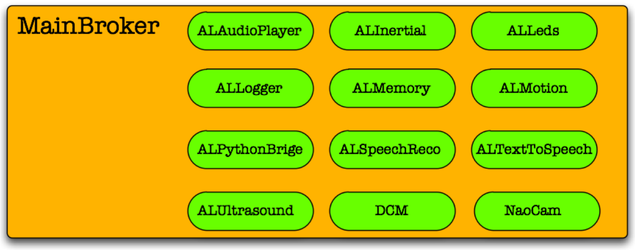

Fig. 3. MainBroker and some modules it contains

Brokers deliver call messages among modules, subscription to data and so on. They also provide a way to resolve module names in order to avoid specifying module's address and port.

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 that provide some interesting functionality (figure 3). We will describe some of the modules intensively used in this work:

- NaoQi provides a thread-safe module for sharing information 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. In this work we have used this module as a *blackboard* where any data produced by any module is published, and any module that needs a data reads from *ALMemory* to obtain it.
- 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 main information source for this work is the camera. In NaoQi the images are fetched by *NaoCam* module. This module uses the Video4Linux driver and makes the images available for any module. If a module wants to obtain images, it has to create and configure a video proxy with an image size, a color space (HSV, RGB, YUV, YUV422) and a frame rate. When it is configured, the module that want an image, only has to call to `getImageRemote()` (when running off board) or `getImageLocal()` (running on board) to get the image. These functions return a structure with the image properties (timestamp, width, height, etc.) and the image data itself.

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

When a programmer develops an application composed by several modules, he decides to implement it as a dynamic library or as a binary (broker). In the dynamic library (plugin) way, the modules it contains can be loaded by the *MainBroker* as its own module. This is faster from point of the view of communication among modules . On the other hand, if any of the modules crashes, then *MainBroker* 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) do not use NaoQi methods and use directly low level DCM functionality. This is much laborious, but it takes absolute control of robot and allows to develop an own walking engine, for example.

The work exposed in this paper uses extensively NaoQi Framework. In fact, the functionality developed is made by a set of *components*, which are actually NaoQi modules with a defined common interface. In the next section we will describe in detail these elements.

IV. Behavioral Architecture

Our approach is based on dividing the behavior in several components. These components can be activated and performs an iterative task at a controlled frequency. They may send commands to actuators, process data from sensors, or activate/deactivate and modulate other components. The components are organised in a hierarchy in order to make a complete behavior. High level component activate low level components, which run concurrently. The components use a common shared memory space to read its inputs and write its outputs. The upper level component connects the output with the inputs of the modules it activate. This way a low level component could be reused by another high level components which could decide to connect the low level components in a different way.

The components developed in this work are functional elements which perform a task. In our particular goal, approach to ball, there are several components that run together to achive this goal: a component that detects the ball, a component that makes the robot turn, another that moves the head in order to center the perceived ball in the camera image, etc.

These components are developed as separated NaoQi modules. Their interfaces are available to other modules and they are able to use all the functionality that NaoQi provides.

Components can be activated or deactivated when its functionality is needed. Two or more components can be activated if they do not use the same actuator. A component can obtain information from sensors, call to motion methods or activate/deactivate other components. When a component activates/deactivates other components, it also modulates their execution.

NaoQi has not a `main()` function where we may implement our control cycle. To control the components execution we have used a synchronisation mechanism provided by NaoQi. In section III we described the module *ALMemory* which stores information about the robot and the modules. As we said before, we can subscribe and unsubscribe to changes in the data stored in *ALMemory*, and specify the maximum frequency we want to read a data when it changes. There is a variable called `Motion/Synchro` that contains the time (in milliseconds) since the robot started up. Each component in our architecture subscribes to this data when it is activated, specifying the frequency *freq* it wants to read `Motion/Synchro`. We have associated the callback method `step()` to attend this events. The method `step()` will be the module control cycle. In this way, `step()` will be executed each $\frac{1}{freq}$ milliseconds. When component is deactivated, it simply unsubscribe to `Motion/Synchro` to finish executing.

Communications among modules use extensively the module *ALMemory*. Each module subscribes to some variables and writes in another variables. When a component activates two or more components, it is in charge of modulating them, often connecting directly one component's output with other component's inputs.

Figure 4 shows an example of module activation and modulation. In any moment any module activates `Module_A` and modulates its execution frequency by writing in the variable `Module_A/freq`, stored in `ALMemory`, a value, in this case 500. This means that `Module_A` will excute `step` method one time each 500 milliseconds. When `Module_A` is activated, it reads the variable `Module_A/freq` to adjust itself its frequency. In any time, `Module_A` activates `Module_C` and `Module_B` and modulates their frequency by setting `Module_C/freq` and `Module_B/freq` respectively. `Module_A` is a module that reads data from a sensor, processes this information and writes the result in `Module_C/outvar`. `Module_B` reads `Module_B/invar` value and depending on this value, sends commands to an actuator. `Module_A` establishes a direct communication from `Module_C/outvar` to `Module_B/invar` in order to make these modules to cooperate.

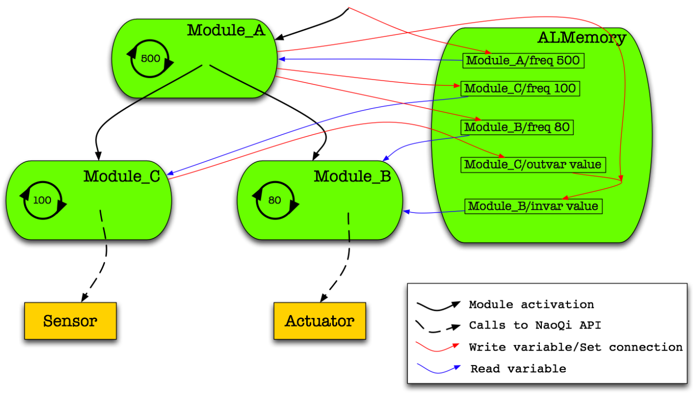

Fig. 4. Example of module activation and modulation

V. FollowBall behavior design

Using the component philosophy presented in the previous section, we have developed a behavior that makes the robot look for an orange ball. This is a basic behavior for playing robot soccer and involves perception and actuation.

To achieve this goal we have developed a set of components. First of all, a perception component called `BallPerception` that obtains an image from camera and processes it to detect the ball and go for it. The actuation is divided in three components: `Head` component moves the head, `Turn` component makes the robot turn left or right and `GoStraight` makes the robot walk straight. These two last components can not be active at same time because they manage the same actuators (all the body excepts the head).

A. `BallPerception` component

This component is in charge of perceiving the ball and obtain its position in an image. As the other components, it has to read the `BallPerception/freq` variable from `ALMemory` and sets its execution frequency. This value in this method is critical because the frequency of the overall system depends on the time spent in processing an image, and it is inversely proportional to the behavior reactiveness.

The image is obtained in HSV colour space and processed by a simple threshold filter to detect the orange pixels. The gravity center of the orange pixels determines the ball position. The ball position (x, y) in the image is normalised in the $([-1, 1], [-1, 1])$ range (as shown in figure 5). It is written in the `BallPerception/x` and `BallPerception/y` variables. Also the amount of orange pixels is stored in

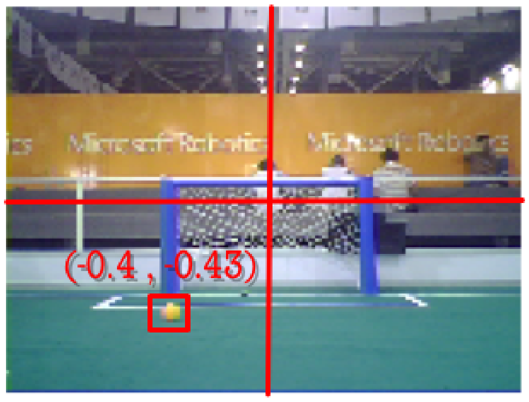

Fig. 5. BallPerception module processing

BallPerception/n to take into account when the orange ball is not present in the image and its size.

B. Head component

This component moves the yaw and pitch motors in the robot's neck. It reads two variables form *ALMemory*: BallPerception/pan and BallPerception/tilt, both in the $[-1, 1]$ range. This value is used as a error value by two PID components that calculate the speed of the actuator. A value near 0 means the motors must stop, and a value near 1 means full speed.

This module frequency depends on the reactivity wanted for this module, which is directly related with the frequency that its input variables are updated. For example, if the information source for this module is the BallPerception module output, both modules should have similar frequency.

C. Turn component

This component makes the robot turn in a direction hiding the complexity of sending motion command regularly to ALMotion module (which is the component provided by NaoQi to manage the robot movements).

This component reads one variable Turn/rotation, that is in $[-1, 1]$ range. This value is, in the same way of Head component, used as a error value by one PID component. An error value near -1 makes the robot turn on the left, 1 makes it turn on the right and 0 stops the robot. Intermediate values modulates the turn speed.

D. GoStraight component

This component makes the robot walk straight in the philosophy of the previous component. It hides the ALMotion module complexity and let us use it in the way we set the displacement as a velocity.

This component reads variable GoStraight/translation in $[-1, 1]$ range. Once again, this value is used as a error value by one PID component to calculate the speed. An error value near -1 makes the robot go back, 1 makes it go ahead and 0 stops the robot. Intermediate values modulates the speed.

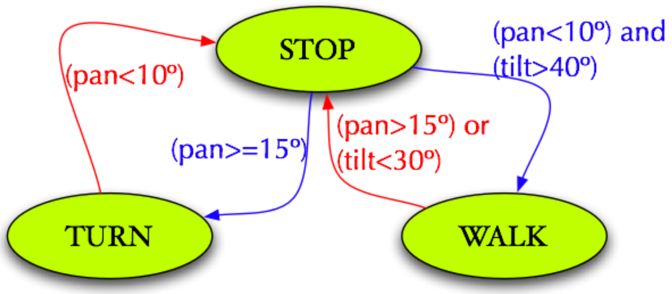

Fig. 6. Finite state machine that implements FollowBall component

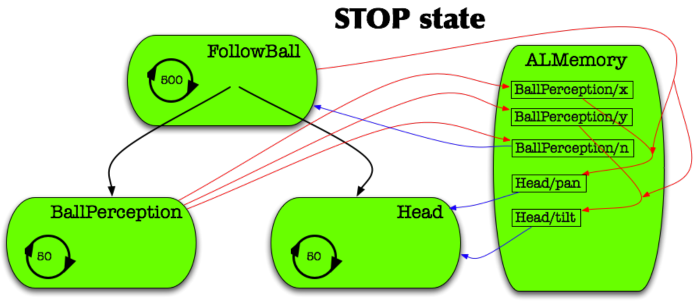

Fig. 7. Activation and connection scheme in *STOP* state

E. FollowBall component

The FollowBall behavior is developed as a component is the same way of the previous ones. It is implemented as a finite state machine, whose state is evaluated each execution step. This finite state machine is shown in figure 6. There are three states: *STOP*, *TURN* and *WALK*. In each state we have some components activated. When a transition from an state to other occurs, we active or deactivate components. In all states Head and BallPerception components are active. This makes the robot center the ball in the image every time. The conditions for state change may depend on the head position. If the neck pan is near 0, the ball is in front of the robot and it is in *WALK* state. If the ball is in the image and the neck pan is high (left or right), we must face the ball. This is made by switching to *TURN* state. Robot stops, in the *STOP* state, when the ball in front on him and it is near.

- In *STOP* state we have activated only the Head and BallPerception components . Any other components previously activated whenentering into *STOP* state is first deactivated. In this and the other states, the connection scheme among these two components are similar. In figure 7 the activation and connection scheme are shown. In this representation we do not show the variables related to frequency for clarity. The BallPerception/x variable is directly connected to Head/pan variable and the BallPerception/y variable is directly connected to Head/tilt variable. Note that the frequency (represented in the oriented circles in each component) is different depending on the reactiveness needed in each component.
- In *TURN* state, represented in figure 8, we have added Turn component to our set of activated components. In

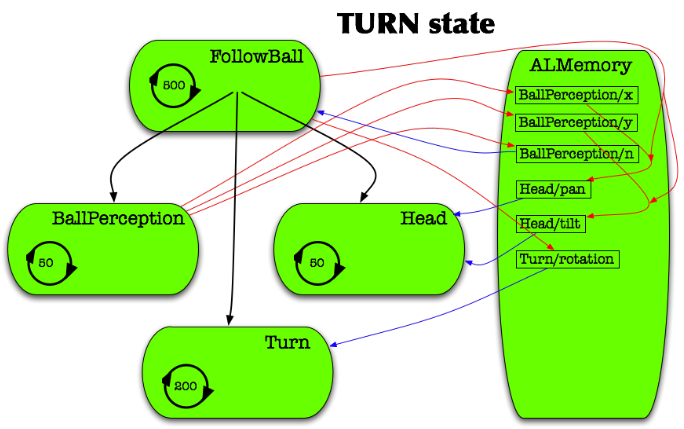

Fig. 8. Activation and connection scheme in *TURN* state

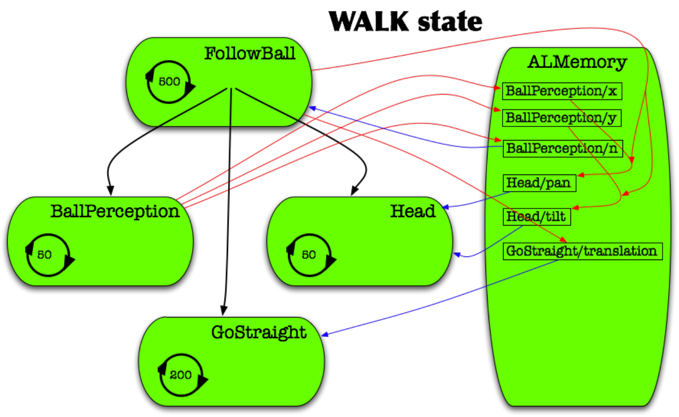

Fig. 9. Activation and connection scheme in *WALK* state

this case we modulate the `Turn` component by writing $pan/MAX\_PANANGLE$ in `Turn/rotation` variable.

- In *WALK* state, represented in figure 9 we activate `GoStraight` component and modulated by writing $tilt/MAX\_TILTANGLE$ in `GoStraight/translation` variable.

So, the conditions for state change depend on the head position. If the neck pan is near 0, the ball is in front of the robot and it is in *WALK* state. In this state we activate `GoStraight` component and it is modulated by writing the neck tilt value on `GoStraight/translation` variable. When the ball is very near, we decide to stop.

If the ball is in the image and the neck pan is high, we must turn in order to align the Nao body to the head bearing. This is made by going to *TURN* state. In this state we activate the `TURN` component and modulate it with the neck pan value writing in `Turn/rotation`. When neck tilt value is high, the robot ball is supposed to be very near from robot, and it stops.

VI. Experiments

To validate the approach presented in this paper we have tested the FollowBall behavior both in the simulator and in the real Nao robot. Experiments in simulator were useful to validate the behavior components and its interaction. We believe this test is useful to detect errors in components interaction, but the definitive test must be carried out in the real robot.

NaoQi works on top of real Nao hardware , on top of Webots[1] or on top of Microsoft Robotics Studio[2](MSR). We use Webots because it is more complete than the other option. MSR is a simulator that displays a simulated robot which only reads the joint positions from `ALMemory`. It does not provides images from the robot cameras neither sensor information. On the other hand, Webots provides the synthetic images taken from the simulated robot cameras and other sensory information.

In this case, a Webots project is provided to use Nao robot by manufacturer. When Webots starts up, a MainBroker starts in the local computer. This MainBroker takes the images and sensor information from the simulator, and the motion commands are translated to the simulated robot. The only aspects not supported in webots are the bumpers and the leds. The same code developed for the robot works in simulator with no changes.

Module	Frequeny
BallPerception	10 Hz
Head	10 Hz
Turn	5 Hz
GoStraight	5 Hz
FollowBall	2 Hz

TABLE I
MODULE ITERATION FREQUECES USED IN EXPERIMENTS.

The frequency set up for the components is shown in table I. We have not a geometric 2D (or 3D) local space perception of ball that updates relative ball coordinates with each robot movement in order to walk to it. Instead of a local perception space, we directly use head pan and tilt to decide the turn and walk activation. This is why the ball tracking with the head must be as fast as possible. The time spent in calculating the ball position in the image is 80-95 milliseconds in real robot. This is why the `BallPerception` and `Head` module has the same frequency, and it is the fastest possible. Decisions about walking does not need such frequency, and it is set to 5 Hz.

The figure 10 shows a typical sequence in the simulator in which the FollowBall behavior is performed. In all the images the robot tracks the ball with its head. When the head pan angle is high the robot is in *TURN* state, turning in order to face the ball, as we can see in the snapshot labeled as 1. The snapshot 2 of the same figure, the head pan angle is almost 0 and the active state is *WALK*. This makes the robot walk straight towards the ball.

In the figure 11 the real robot performs the FollowBall behavior. As in the previous test in simulator, in every moment the robot tracks the ball with its head. It starts in *STOP* state and transits to the *TURN* state to face the ball. When head pan angle is low, the robots walks in order to reach the ball.

[1]http://www.cyberbotics.com/
[2]http://msdn.microsoft.com/en-us/robotics/default.aspx

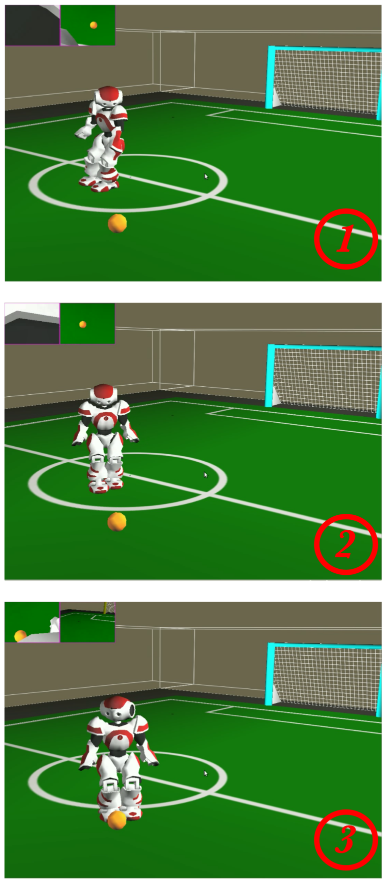

Fig. 10. FollowBall beharior in Webots simulator

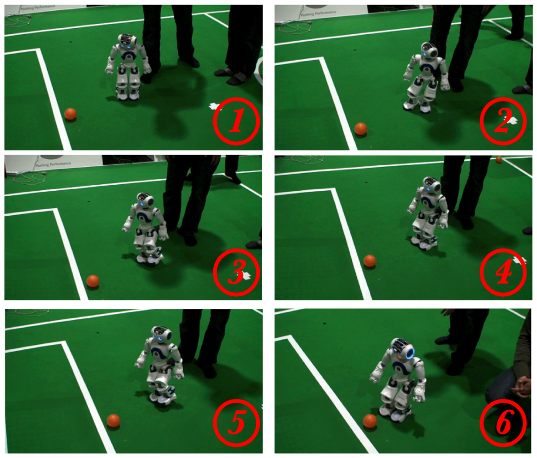

Fig. 11. FollowBall behavior in the real robot

One of the main differences between real and simulated robot is the limited computational time available to do all the processing. Much of this time is spent in fetching the camera image. For this reason, our module is compiled as a MainBroker plugin in the real robot: This let us to speed up the access to camera images because memory access mechanism is used instead of SOAP mechanism. Using this mechanism our module consumes about 94% of the available CPU time and 97% of the total amount of memory. NaoQi consumes about 30% of the available CPU time and 64% of the total amount of memory when idle.

In these tests we have used the same code both for simulator and robot. In every moment we can test in the simulator the code developed for the real robot, but usually the code developed in the simulator does not work so well in the real robot, and further development is needed. Besides that, testing the code in the simulator saves a lot of time and avoids to make some mistakes easy to detect in the simulator, but hard to detect in the real robot.

As we have shown in these tests, the approach presented in this paper is effective to achieve the goal proposed on it. This work has been also tested in a real competition, in the German Open Competition that took place in Hannover, April 2009.

VII. Conclusions

This work let us got familiar with the new framework for Nao programming named NaoQi, in order to learn how to use the robot locomotion and perception. Nao is a new hardware platform with promising capabilities. Its functionality, characteristics and low price (for this type of robots) make it a promising humanoid platform for the next years.

In this paper a FollowBall behavior composed by several components has been presented developed inside a behavior based software architecture. In this behavior we have implemented a perceptive component (`FollowBall`) to detect the ball, 3 actuation components (`Head`, `Turn` and `GoStraight`) and a component that using a finite state machine selects the component to be active in each moment and modulates it (`FollowBall).`

These components can be activated by another components, in a component activation hierarchy. When a component is activated, it performs an iterative simple task at a configured frequency. Components that activate another components also modulate its behavior and connect the results produced by a component with another component input. We have shown how the component performs its task. It can be very simple or it can implements finite state machine. Its task can be reading from a sensor, sending command to an actuator or activating and modulating another components.

We have extensively used the NaoQi framework to read information from sensors and use the robot actuators. We also have used the `ALMemory` module as a blackboard where the components can communicate among them, receive modulation from another components or set modulation to another ones.

Using this approach we have developed a FollowBall behavior that uses some component in different ways, showing how a module can be reused for various different tasks. This behavior is reactive to the changes in the ball position.

There are several videos[3] that show this behavior in the simulator and in the real robot, proving the robustness of this approach when facing to real world conditions. The participation in German Open Competition 2009 (Hannover) demonstrates the viability of this approach, in which the follow

[3]http://www.teamchaos.es/index.php/URJC#FollowBall

ball behavior was completed by other states to search and kick the ball.

As future lines, we are planning to develop new components and reuse the ones presented in this work to obtain a search ball behavior and a kick behavior. Perception will be redesigned to perceive more elements in the field and more efficiently. Perception will also be redesigned to perceive the relevant landmarks to develop a self-localization method in order to implement more complex position dependent component.

REFERENCES

[1] Reid Simmons, R. Goodwin, K. Haigh, S. Koenig, Joseph O'Sullivan, and Maria Manuela Veloso, *Xavier: Experience with a Layered Robot Architecture,* Agents '97, 1997.

[2] Alexander Stoytchev and Ronald C. Arkin, *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.

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

[4] Thrun, Sebastian and Bennewitz, Maren and Burgard, Wolfram and Cremers, Armin B. and Dellaert, Frank and Fox, Dieter and Hahnel, Dirk and Rosenberg, Charles R. and Roy, Nicholas and Schulte, Jamieson and Schulz, Dirk, *MINERVA: A Tour-Guide Robot that Learns*. KI - Kunstliche Intelligenz, pp. 14-26. Germany, 1999.

[5] Saffiotti, Alessandro and Wasik, Zbigniew, *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.

[6] Scott Lenser and James Bruce and Manuela Veloso, *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.

[7] T. Rofer and H. Burkhard and O. von Stryk and U. Schwiegelshohn and T. Laue and M. Weber and M. Juengel and D. Gohring and J. Hoffmann, B. Altmeyer and T. Krause and M. Spranger and R. Brunn and M. Dassler and M. Kunz and T. Oberlies and M. Risler and M. Hebbela and W. Nistico and S. Czarnetzkia and T. Kerkhof and M. Meyer and C. Rohde and B. Schmitz and M. Wachter and T. Wegner and C. Zarges. *German team: Robocup 2005*. Technical report, Germany, 2005.

[8] R.Calvo, J.M.Cañas, L.García-Pérez. *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.

[9] Cañas, J. M. and Matellán, V. *From bio-inspired vs. psycho-inspired to etho-inspired robots* Robotics and Autonomous Systems, Volume 55, pp 841-850, 2007. ISSN 0921-8890.

[10] Antonio Gómez Skarmeta, y Humberto Martínez Barberá, *Fuzzy Logic Based Intelligent Agents for Reactive Navigation in Autonomous Systems*, Fitth International Conference on Fuzzy Theory and Technology, Raleigh (USA), 1997

[11] M. Loetzsch, M. Risler, and M. Jungel. *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.