Behavior-based **I**terative **C**omponent **A**rchitecture for robotic applications with the Nao humanoid

Francisco Martín José M. Cañas Carlos Agüero Eduardo Perdices

Robotics Group U.Rey Juan Carlos francisco.rico@urjc.es

Robotics Group U.Rey Juan Carlos jmplaza@gsyc.es

Robotics Group U.Rey Juan Carlos caguero@gsyc.es

Robotics Group U.Rey Juan Carlos edupergar@gmail.com

Abstract

Software architectures are essential for robotic applications development. They organize perception and actuation capabilities in order to achieve the goals the robots are developed for. In this paper we present the second major release of our software architecture, named BICA, that aims to be applied in a wide range of applications using the Nao humanoid robot as the hardware platform. This architecture has been designed using state-of-the-art concepts to be reliable, extensible and efficient and the second release improves some of the shortcomings observed along the experience with the initial design. In order to prove these features, this architecture has been tested in different domains, mainly the Robocup Standard Platforms League, which is very demanding, competitive and dynamic. Around this software architecture we have developed an useful set of tools to design, setup and debug the perceptive habilities and the behaviors the robot performs.

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 [2], 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, localization, behavior generation and human-machine interaction.

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.

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. Until 2007, the official robot to play in this league was quadruped Aibo robot, but since 2008 is the

Figure 1: Robot Nao playing soccer.

Nao humanoid (Figure 1). This change 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 big differences on perception. 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.

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-localize 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 localization method if that takes several seconds to compute the robot position or to decide the next movement in few seconds based on the old perception. The estimated sense-think-act process must take less than 200 millisecond to be truly efficient. 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 or modulate other components. This approach will meet the vivacity, reactivity and robustness needed in this environment. This architecture is inherited from the one we presented in [1], but redesigned in order to improve efficiency and reliability. We will present the problems we had to face to using the previous approach. These problems made us to redesign the entire architecture.

In section 2 we will present relevant previous works which are also focused in robot behavior generation. In section 3 we will present the Nao and the programming framework provided to develop the robot applications. In section 4, the behavior based architecture and their properties will be described. Next, in section 5, we will show two useful tools in the architecure. In section 6 we will describe some experiments carried out to validate the architecture and some actuation and perception developed components. Finally, section **??** will summarize the conclusions.

2 Related works

There are many approaches that try to solve the behavior generation problem. One of the first successful works on mobile robotics is Xavier [3]. 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 [4], where a hybrid architecture, which behavior is divided into three components, was proposed: delibera-

tive 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 [5].

The JDE architecture [9] has several similarities with the one presented in this work, including the activation/deactivation of reactive components called schemas.

In the RoboCup domain, a hierarchical behavior-based architecture was presented in [7]. 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 [6] presented another approach in this domain: the ThinkingCap architecture. This architecture was based in a fuzzy approach, extended in [10]. The perceptual and global modeling 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 [13] to the Standar Platform League, where the behaviors were developed using a LUA interpreter.

Much research has been done over the Standar Platform League. The B-Human Team [8] divides their architecture in four levels: perception, object modeling, 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 the 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.

A different approach was presented by Cerberus Team [**?**], 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 [15] 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 robot manufacturer provides an easy way to access to the hardware and also to several high level functions, useful to implement the applications. This software is called NaoQi and provides a framework to develop applications in C++ and Python. Our soccer robot application uses some of the functionality provided by this underlying software.

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.

Every binary, also called broker, runs inde-

pendently and is attached to an address and port. Every broker is able to run both in the robot (cross compiled) and in the computer. A complete application may be 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.

NaoQi is voracious, consuming a lot of memory and computing resources. Intensive use of memory, communication or synchronization mechanism provided by NaoQi affect to the robots movement. That is why we try to use NaoQi as less as we can. In fact we use NaoQi for motion and camera access mainly. We use intensively only two NaoQi modules:

- **ALMotion** This module provides a set of methods to move the robot motors. There are methods which set a joint angle, methods which let us to set robot chains (head, foots, and arms) to a desired cartesian position, and methods which provides a high level functionality, such as walking movement.
- **ALVideoDevice** This module lets us to acquire an image for the camera.

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 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. On the other hand, NaoQi motion mechanism is better than the majority of the motion mechanisms developed until now.

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. The robot is equipped with a microcontroller ARM 7 allocated in its chest to control the robot's motors and sensors, called DCM.

These hardware features impose some restrictions to our behavior based software 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 sophisticated 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 software developed on top of NaoQi can be tested both in real robot and simulator. We use Webots (Figure 2) (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.

4 BICA: Behavior-based architecture for robot applications

It is possible to develop basic behaviors using only the Naoqi framework, but it is not enough for our needs and developing complex applications using NaoQi only is hard. 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

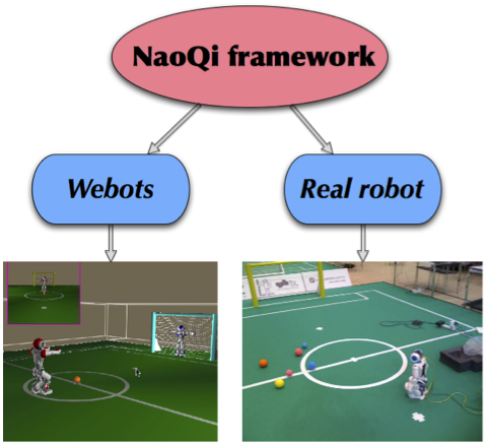

Figure 2: Webots simulator and real robot.

variety of simple applications available.

In this section we will describe the design concepts of the robot architecture we propose in this paper, named BICA. The main element in BICA is the component, it 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. When it is active, it is running and a `step()` function is iteratively called to perform the component 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.

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 functionality 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 3 there is an example of an activation tree. When component A, the root component, is activated, it activates component B and E. Component B activates C and

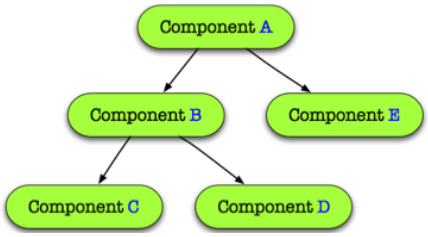

Figure 3: Activation tree composed by several components.

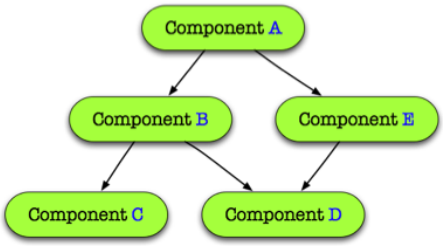

Figure 4: Activation tree where two components activate the same component.

D. Component A needs all these components activated to achieve its goal. This structure 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.

Two differents components are able to activate the same child component, as we can observe in Figure 4. 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 components.

The activation tree is no fixed during the robot operation. Actually, it changes dynamically depending on many factors: main task, environment element position, interaction with robots or humans, changes in the environment, error or falls. The robot must

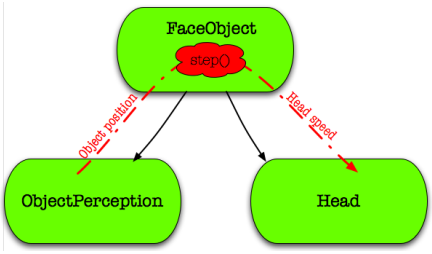

Figure 5: Activation tree with two low level components and a high level components that modulates them.

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.

As an example, Figure 5 shows 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 not modulate or activate any other component, it only extracts information from an image. The FaceObject component is an iterative controller, that activates and modulates other components. One component may activate and deactivate other components dynamically depending on the situation. They are implemented as finite state machines. In each situation a set of components is active, 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 to the SOAP message passing approach.

Activations and deactivations are made implicit in the components code. There is not an activate method, but each component that wants to activate other component, calls to its `step()` method. When NaoQi module is created, it starts a thread which continuously 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.

The `step()` code of the last example looks like this:

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

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

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 maximum 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) items of the structure the have just described. If the elapsed time is lower, it only executes (1) and (3) items.

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 discarded (graceful degradation).

5 Tools

Several tools have been built in BICA, which ease and speed up the development of robotic applications, their debugging and fine tunning.

5.1 VICODE: VIsual COmponent DEsigner

The robot applications are organized as a collection of connected components, perceptive ones and actuation ones. Developing them in BICA is quite easy, but can be tedious and tricky for complex components. Some actuation components may be succesfully programmed as reactive controllers or simple PID feedback controllers. Many times the complexity of the components fits well in finite state machines (FSM). Using FSMs powerful components can be programmed which unfold complex behaviors. But developing complex behaviors based on FSMs is complicated and prone to errors. Because of this we have developed an useful tool, named VICODE (VIsual COmponent DEsigner), that automatically generates C++ code from a visual description of the finite state machine.

We use VICODE for the development of complex components, and even for the basic ones, as the code generation is faster and more reliable using it than writting the code manually in C++.

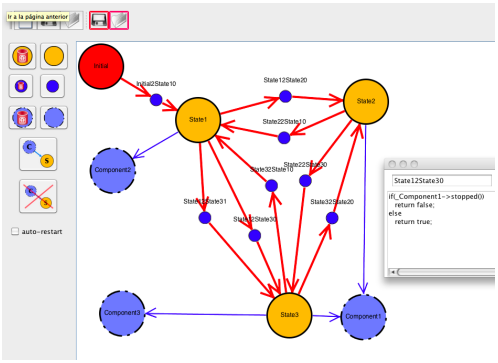

Figure 6: VICODE tool for component generation.

This tool (Figure 6) let us to design an iterative finite state machine setting its states and transitions. Each state has a source code attached to be run at each iteration of the FSM being in such state. At the same time it has a source code to check possible transitions from it to other states when certain perceptive conditions are met. Furthermore, we can visually establish which components are used in each state, and whether it is a modulation or a requirement link.

VICODE generates the component C++ code. This includes state machine code, the headers file with the component API, and calls to the `step()` method of the components that it uses or modulates. VICODE lets us to edit the states and transitions code. This code is even refreshed if the code is externally edited to avoid inconsistencies. Transitions are defined as functions that return true or false if the transition has to be taken. This information to take the decisions can be provided by other components or by a timer (used for time-based transitions).

5.2 JManager

VICODE is included in the JManager tool as a tab. JManager is an external application which centralizes all the debugging and monitorization tools developed for the BICA architecture. This tool lets to set up the components (e.g. color of the stimulus), activate and modulate them. Each component may have an

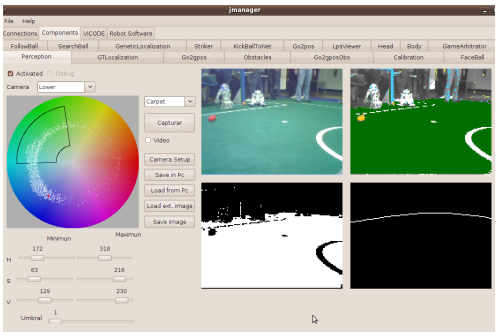

Figure 7: Debugging tool: JManager.

specific tab inside JManager for its debugging.

For instance, a color filter tuner tab is shown at Figure 7, which lets us select on the fly the right thresholds for the color filter component inside BICA. The computation of the End-Of-Field in the image can also be debugged.

JManager runs at an external computer and connects to the BICA software inside the Nao humanoid using an ad-hoc communication protocol through the wireless or wired network connection. It has been programmed in Java.

6 Experiments in the RoboCup scenario

In this section we will present, not only the experiments carried out to validate this behavioral architecture, but the previous approach we took in the architecture design.

6.1 First BICA design

Not always the first steps are the right ones. In this architecture design, the proposed solution is not the first approach we took. At the beginning we tried to exploit all the benefits that NaoQi provides. This software lets to decompose our application functionality in NaoQi modules which cooperate among them to achieve a goal. Each module performs some processing task and sends data to other modules. This would let to implement our architecture in a natural way using this approach. NaoQi has a functionality 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. This was the first approach we took to design our architecture. Unfortunately, an intensive use of these mechanisms had a big impact in NaoQi performance and some real time critical tasks were severely affected. One of them is the movement generation. When the performance in a task is poor, the movement is affected and the robot fell to floor.

6.2 Forward soccer Player

Using BICA we have developed the fordward Player behavior set and tested it at RoboCup 2009 in Graz with real robots. Before the real tests we used the tools described in section 5 to calibrate the colors of the relevant elements in the environment. Once tuned, the robot is ready to work. The next sequence has been extracted from a video which full version may be visualized at www.teamchaos.es/index.php/URJC#RoboCup-2009.

Figure 8 shows the finite state machine corresponding to the forward player component. Figure 9 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. Player component is continuously 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 activated, approaching to the ball (fifth image in the sequence).

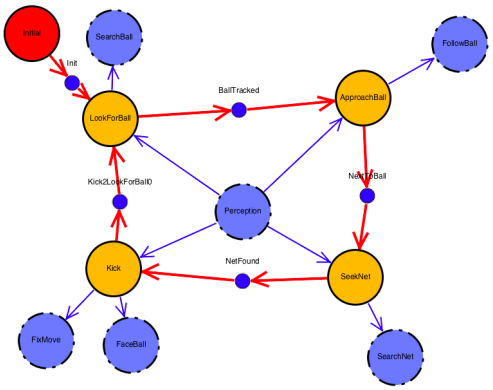

Figure 8: Finite State Machine for Player behavior

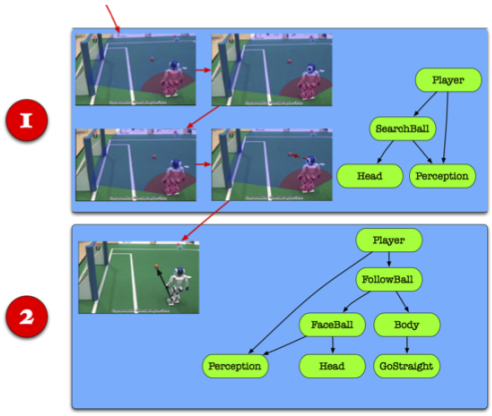

Figure 9: Ball searching sequence.

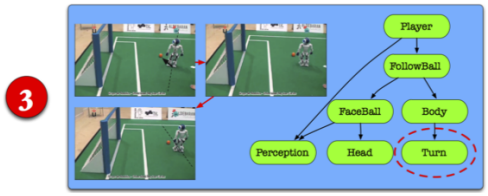

Figure 10: Ball approaching modulation to make the robot turn.

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 taken into account. Only the head pan is used by the Body component to approach the ball.

In Figure 10, while 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 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.

The robot reaches the ball while 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 11. 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

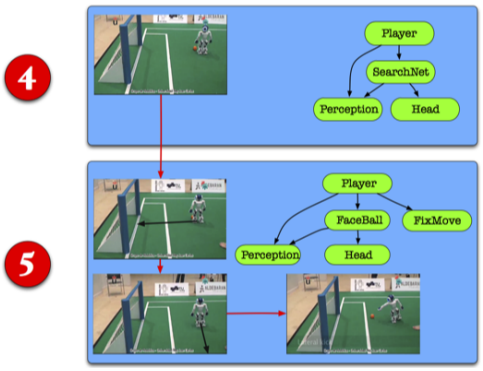

Figure 11: Search net behavior and kick.

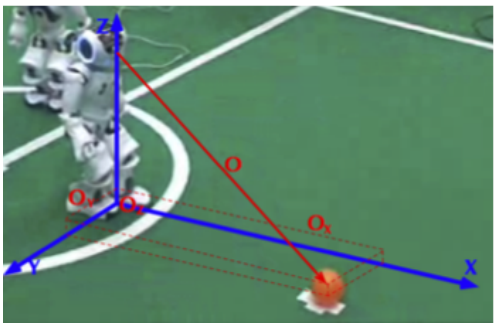

Figure 12: 3D perception.

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 kicking the ball, the robot must be aligned in order to situate itself in the right position to do an effective kick. For this purpose, the player component asks to the Perception module the ball position in 3D with respect to the robot. This is the only time the ball position is estimated. The player component activates Fixmove component with the selected kick and a lateral and straight alignment. As we can see in third and fourth images, the robot moves on its left and back to do the kick. While the kick is performing and after the kick, FaceBall component is activated to continue tracking the ball.

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 people.

6.3 Player perception

Several perceptive components have also been programmed in order to provide the relevant enviroment information for control decisions. At the RoboCup competition, the environment is designed to be perceived using vision and all the elements have a particular color and shape.

In our architecture, perception is decomposed in several components. There are a different component for each different stimuli (ball, net, other robot, field lines,...). This let us to save processing time when a stimulus is not needed. For example, if FaceBall component is the only one active, it activates the BallDetector components and it looks only for the ball in the image. If localization is active, it activates the components that perceives nets and field lines.

We perform a 3D perception using projective geometry. Taking into account all the joint positions, we can determine the camera position and orientation in 3D. This let us to project the image pixels in the 3D world to determine the 3D position (knowing other data, like the Z coordinate) and backproject a 3D hypothesis on the image to validate an image detection.

Each perceptual component stores the element position, and actualizes it with odometry in order to maintain a perceptual memory. This is useful to develop behaviors which depends on several stimulus, and they are not always present in the image.

7 Conclusions

In this paper we have proposed a robotic behavior based architecture. With this architecture we can create robotic 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 and latent components that are waiting for be activated. This hierarchy is called activation tree, and dinamically changes during the robot operation. The components whose output is not needed are deactivated in order to save the limited resources of the robot.

In this paper we have shown how the behaviors are implemented within the architecture. As a test, we have created a forward player behavior to play soccer in Standar Platform League at the RoboCup. This is a dynamic environment where the conditions are very hard. Robots must react very fast to the stimulus in order to play well. This is an excellent test to the behaviors created within our architecture.

We have developed several components to get a forward soccer player behavior. These components are latent until a component activate it to use it. These components have a standard modulation interface, perfect to be reused by others without any modification in the source code or to support multiple different interfaces. The highest level component is the Player component. This component has been implemented as a finite state machine using one BICA tool, VICODE, for the visual design of FSM. It activates the previously described components in order to obtain the forward player behavior.

This Player behavior has been tested in the RoboCup environment, but BICA architecture is not limited to that scenario. We want to use this architecture to create robot behaviors to solve other problems out of this environment. For instance we are working in using the humanoid robot in healthcare applications where it serves as a personal assistant for elder people, or as a cognitive estimulation therapeutic tools for Alzheimer patients.

Acknowledgments

This work has been funded by Spanish Ministerio de Ciencia y Tecnología, National Program of Design and Industrial Development Project, under the COCOGROM project DPI2007-66556-C03-01 and Comunidad de Madrid under the project RoboCity2030-II: S2009/DPI-1559.

References

[1] Francisco Martín, Carlos E. Aguero and José María Cañas (2009). *Follow ball behavior for an humanoid soccer player*. X Workshop de Agentes Físicos. Cáceres (Spain), Septiembre 2009.

[2] 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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] Cañas, 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.

[10] 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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.