Behavior-based Iterative Component Architecture for soccer applications with the Nao humanoid

Carlos E. Agüero #[1], Jose M. Cañas #[1], Francisco Martín #[1] and Eduardo Perdices #[1]

#*Robotics Group, Universidad Rey Juan Carlos*
Tulipán s/n 28933 Móstoles, Madrid (Spain)
[1]{caguero, jmplaza, fmartin, eperdices}@gsyc.es

***Abstract*— Software architectures are essential for robotic applications development. They organize perception and actuation capabilities in order to achieve the goals the robots for which are developed. In this paper we present the third 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 last release improves some of the shortcomings observed along the experience with the initial design, as well as the addition of new specific components for perception, auto-localization and communication. This architecture has been tested in different domains, mainly the RoboCup Standard Platform 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 abilities and behaviors of the robot.**

I. Introduction

The focus of robotic research continues to shift from industrial environments to mobile service robots operating in a wide variety of scenarios, often in human-habited ones.

In many cases, research is motivated by accomplishment of a difficult task. In robotics there are several competitions which present a problem and must be solved by robots. For example, Grand Challenge or Urban Challenge propose a robotic vehicle to cross hundred of kilometers driving autonomously.

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.

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 focus on the software aspects rather than in the hardware. Since 2008, the official robot in this league is the Nao humanoid. Robot Nao is a fully programmable humanoid robot. Nao is about 55 cm. tall and can perceive in 3D because of the two cameras installed at a high position that enables the robot to calculate the position of the elements that are located on the floor.

With this work, we are proposing a behavior-based software architecture that meets 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 meets the vivacity, reactivity and robustness needed in this environment. This architecture is inherited from the one we presented in [1], but it has been redesigned in order to improve efficiency and reliability.

In section II we will present relevant previous works which are also focused in robot behavior generation. In section III we will summarize the Nao and the programming framework provided by the manufacturer to develop the robot applications. In section IV, the behavior based architecture and their properties will be described. Next, in section V, we will show one useful tool in the architecture. In section VI we will describe some experiments carried out to validate the architecture and some actuation and perception developed components. Finally, section VII will summarize the conclusions.

II. 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, each with a specific task and priority. 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, in 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 [5] 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.

The JDE architecture [8] 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 [6]. 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.

Much research has been done over the Standard Platform League. The B-Human Team [7] 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 machines were written in XABSL language [10], that was interpreted at runtime, and it allows to change and reload the behavior during the robot operation.

A different approach was presented by Cerberus Team [12], 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.

III. 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 the hardware and also 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 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.

NaoQi is voracious, consuming a lot of memory and computing resources. Intensive use of memory, communication or synchronization mechanism provided by NaoQi affects the robots movement. We use NaoQi for motion and camera access mainly through two NaoQi modules: ALMotion and ALVideoDevice.

The 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.

IV. 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 the development of complex applications using NaoQi alone is hard. We need an architecture that lets us 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.

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, which is the basic unit of functionality. At any time, each component can be active or inactive. When it is active, it is iteratively running a `step()` function 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 allows the decomposition of functionality in several components that work together. An application is a set of components in 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.

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 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 1 shows an activation tree composed of three components. ObjectPerception is a low level component that determines the position of an interesting object in the image. 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 FaceObject for the object position and modulates head movement to obtain the global behavior: facing the object.

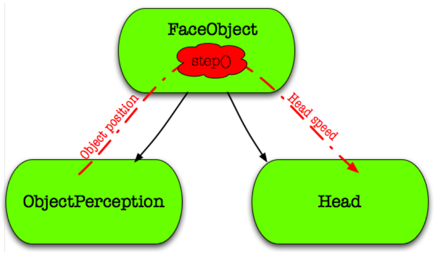

Fig. 1. Activation tree while executing Faceball behavior

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.

Activations and deactivations are made implicit in the components code. There is not an activate method, but each component that wants to activate the 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 component methods in level n-1 to obtain information and calls to lower level component 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 makes no sense that all the components execute at the same frequency. Some pieces of information are needed to be refreshed very quickly, and some decisions are not needed to be taken so quickly. When a `step()` method is called, it checks if the elapsed time since last execution is equal or higher to the one according to its frequency. In that case, it executes (1), (2) and (3) items of the `step()` structure. If the elapsed time is lower, it only executes (1) and (3) items.

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

V. VICODE TOOL: VISUAL COMPONENT DESIGNER

The robot applications are organized as a collection of connected components, perceptive ones and actuation ones. Some actuation components may be successfully 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 a useful tool, named VICODE (VIsual COmponent DEsigner), that automatically generates C++ BICA component 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 writing the code manually.

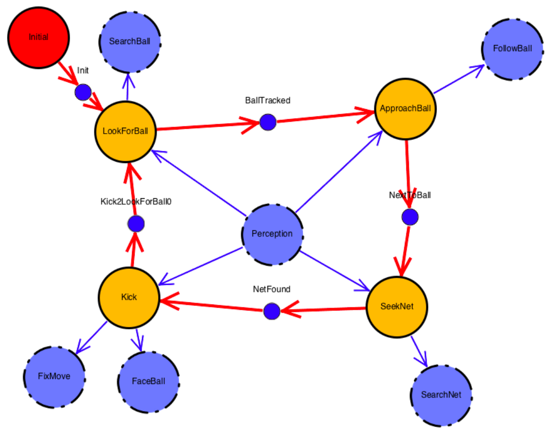

Fig. 2. Finite State Machine for Player behavior

This tool lets us 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 the 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 make the decisions can be provided by other components or by a timer (used for time-based transitions).

VI. EXPERIMENTS IN THE ROBOCUP SCENARIO

In this section we present the experiments carried out to validate this behavior based architecture.

A. *Forward soccer Player*

Using BICA we have developed the forward Player behavior set and tested it at RoboCup 2009 in Graz and at Mediterranean Open 2010 with real robots. A video with a sample of the behavior may be visualized at www.teamchaos.es/index.php/URJC#RoboCup-2009.

Figure 2 shows the finite state machine corresponding to the forward player component. Figure 3 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

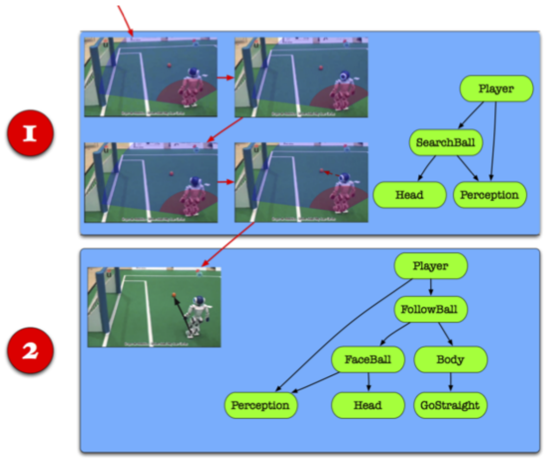

Fig. 3. Ball searching sequence

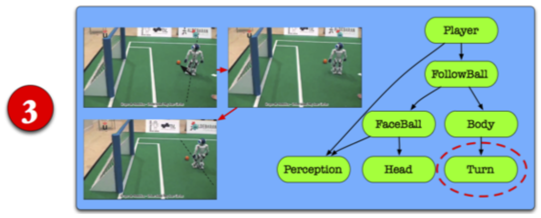

Fig. 4. Ball approaching modulation to make the robot turn

(fourth image in the sequence), SearchBall component is deactivated and FollowBall component is activated, approaching to the ball (fifth image in the sequence).

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.

In Figure 4, 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 5. In that moment, the robot has to decide which kick it has to execute. For this reason, the net has to be detected. In the last image, the conditions to kick the ball are held and the player component deactivates FollowBall component and activates the SearchNet component. The SearchNet component has as output a value that indicates whether the scan is complete or not. The Player component queries in each iteration if the scan is complete.

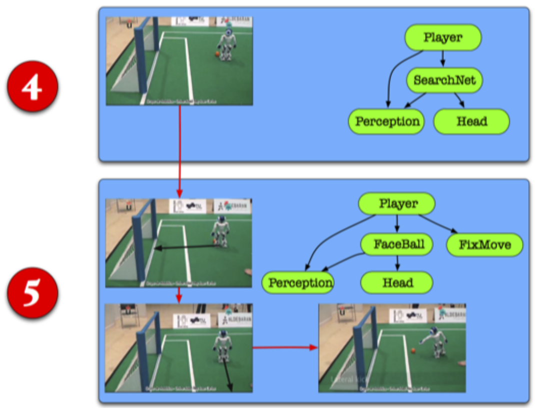

Fig. 5. Search net behavior and kick

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 three types of kicks: front, diagonal and lateral. Actually, we have six 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 requests the ball position in 3D with respect to the robot of the Perception module. 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 was carried out in a real competition environment, where the robot operation showed robust to the noise produced by other robots and people.

B. Ball perception in 3D

Several perceptive components have also been programmed in order to provide the relevant environment 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.

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 lets us 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.

The modular design of BICA has allowed component developers to test different implementations of the same component maintaining the same interface. This is the case of the perception module, where a novel alternative has been developed. The classical algorithm processes the full set of pixels or a region of interest of an image looking for important features.

As we mentioned before we can apply projective geometry to estimate the 3D position of the objects.

In contrast to this approach we have inverted the method maintaining a set of hypothesis in the real 3D world. Each hypothesis is then backprojected to the image to be validated. We have carried out different methods for validating and ranking every hypothesis. For the case of the ball, every hypothesis uses two different cubes. An internal cube 3D is calculated surrounding the hypothetical ball that is representing. An external 3D cube is also calculated to represent the surrounding area of the ball. Both cubes are backprojected into the image plane as we can see in figure 6 generating the proximity windows (pink and cian in figure 6 left).

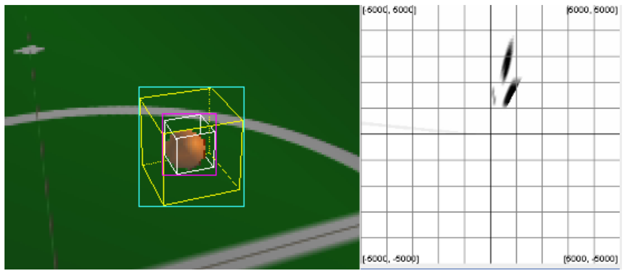

Fig. 6. Cubes, proximity windows (left) and object estimation window (right)

The population of hypothesis is maintained using a genetic algorithm. The fitness function is shown in equation 1, where n is the number of individuals, D_{int} is the density of positive pixels inside the internal proximity window, D_{ext} refers to the density of positive pixels inside the external window deducting the area occupied by the internal window, while α and β are coefficients for weighting each factor.

$$h_i = \frac{1}{n} + ARGMAX(0, \alpha * D_{int.} - \beta * D_{ext.}) \qquad (1)$$

The results have shown a reduction of a 30%-60% (depending on the distance to the ball) in the CPU consumption due to the minor number of pixels checked compared with the classical approach of evaluating all of them. Another advantage in robustness has been deduced from the experiments due to the fact that we do not explore other areas that can cause potential problems as false positives.

C. Visual self-localization

One of the most important tasks in the RoboCup SPL is the robot self-localization, since the robot's behavior has to be different depending on its location. The field where robots play can provide us the information we need, given that it has two colored goals (blue and yellow) and several white lines such as the side lines, the penalty area lines, the halfway line and the central circle.

To solve the robot localization problem there are different approaches used by other teams. The most common are the particle filters, such as Monte Carlo Localization (MCL) [13], that has been used for several teams obtaining good results [14], [15]. However, MCL is only able to handle one solution for robot localization at each iteration, what could take the solution to wrong locations in case of symmetries. Taking this into account, we have designed a new approach specifically created to bear symmetries in the RoboCup field. This new approach uses an evolutionary algorithm, a type of metaheuristic optimization algorithm that is inspired by the biological evolution to solve a problem.

The main idea of the algorithm consists of several races (candidate solutions) competing between each other in different field positions. These races are created, for instance, after getting symmetries from the observations. After some iterations, predictably, the observations will provide information to rule out the wrong races (i.e. solutions) and we will obtain the real robot localization.

The observations have been analyzed by covering the input images horizontally and vertically using a grid of variable size. For each vertical and horizontal line of the grid we perform several color filters to find the borders of the main objects of the field, i.e. goals and lines. With these color filters we obtain isolated points, which must fulfill other filters to be validated, such as not being part of other robots or being close to other points with the same color. Finally, with these points validated we get the characteristic points of the image.

The analysis of the images takes 1.5 ms in the real robot, meanwhile, each iteration of the evolutionary algorithm takes 37.7 ms on average. In the figure 7 we show an experiment to validate the precision of our approach. In this figure we show the trajectory calculated with the algorithm (red), the calculated localization in several check points (blue circles) and the real localization (yellow circles).

The algorithm is capable of following the real movement of the robot with an average error of 14.3 cm and 8.42 degrees. Furthermore, we can emphasize that the trajectory followed by the robot is very stable and is always close to the real location of the robot.

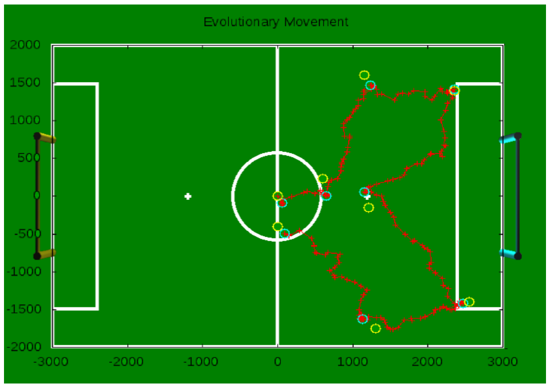

Fig. 7. Movement with evolutionary algorithm

Also, we have performed other experiments with kidnappings and we have tested the algorithm in real competitions obtaining good results.

D. Component communication using Ice middleware

The mature state of the league has helped the integration of team strategies and the development of a wide set of tools for monitoring and debugging purposes. Those applications demand a specific communication module to implement the services required. The common requirements are sending / receiving primitives, data marshalling, subscription / publication services and some level of configuration for switching between reliable data or more efficient but unreliable communication. The classical approach to solve these issues is to use some library based on sockets. This well known alternative could be considered as a low level communication layer since the developers must solve some of the aspects mentioned before.

The Internet Communication Engine (Ice)[16] is an object oriented middleware with support for multiples languages and different operating systems. One of the key features in Ice is the concept of interface. This file (written in a specific language called Slice) describe the set of operations and rules that other components can invoke.

One of the main advantages of BICA is its modularity and with the inclusion of Ice this feature has been reinforced. Every component that can be potentially used by other robot or device must include its own Ice interface. The full set of operations of the interface should be implemented by the component.

Using this alternative the clients can now invoke the operations of a remote component in the same way they would do it if the operation was local. The Ice infrastructure is very flexible and allows some degree of configuration for every interface or operation. For example, we can specify a reliable communication between the Perception component and the external tool for acquiring images, but a fast, efficient and unreliable exchange mechanism of basic data sharing among teammates.

VII. Conclusions

In this paper we have proposed a robotic behavior based architecture for creating robot 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 other components. In every moment, there are active components and latent components that are waiting to be activated. This hierarchy is called activation tree, and dynamically changes during the robot operation.

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 Standard Platform League at the RoboCup. Robots must react very quickly to the stimulus in order to play well. This is an excellent test to the behaviors created within our architecture.

The forward player behavior is made of several components. 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. For instance, we are working on using the humanoid robot in healthcare applications where it serves as a personal assistant for elder people, or as a cognitive stimulation therapeutic tool for Alzheimer patients.

References

[1] Martín F.; Agüero C. E.; Cañas J. M. *Follow ball behavior for an humanoid soccer player*. X Workshop de Agentes Físicos. Cáceres (Spain), 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. *MINERVA: A Tour-Guide Robot that Learns*. Kunstliche Intelligenz, pp. 14-26. Germany, 1999.

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

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

[6] Lenser, S.; Bruce, J.; Veloso, M. 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] 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. B-Human. Team Description and code release 2008. Robocup 2008. Technical report, Germany, 2008.

[8] 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.

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

[10] Loetzsch, M.; Risler, M.; Jungel, M. 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.

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

[12] Akin, H.L.; Merili, .; Merili, T.; Göke, B.; Özkucur, E.; Kavakhoglu, C.; Yildiz, O.T. Cerberus08 Team Report. Technical Report. Turkey, 2008.

[13] Fox D.; Burgard W.; Dellaert F.; Thrun S. Monte carlo localization: Efficient position estimation for mobile robots. In Proceedings of the National Conference on Articial Intelligence, 1999.

[14] Hester T.; Stone P. Negative information and line observations for monte carlo localization. The IEEE International Conference on Robotics and Automation, 2008.

[15] Laue T.; Jeffry de Haas T.; Burchardt A.; Graf C.; Röfer T.; Hartl A.; Rieskamp A. Efficient and reliable sensor models for humanoid soccer robot self-localization. The 2009 IEEE-RAS Intl. Conf. On Humanoid Robots, 2009.

[16] Henning M. A New Approach to Object-Oriented Middleware. IEEE Internet Computing, vol. 8, no. 1, pp. 66-75, 2004.