

Jde-neoc: component oriented software architecture for robotics *

José M. Cañas, Jesús Ruíz-Ayúcar, Carlos Agüero, Francisco Martín

Dept. Ingeniería Telemática y Tec. Electrónica

Universidad Rey Juan Carlos

28933 Móstoles

jmplaza@gsync.escet.urjc.es

Abstract

In this paper we present our software framework for robotic applications, *jde-neoc*. This is the second implementation of our cognitive behavior-based architecture JDE, and it is aimed to overcome some of the limitations observed in three years using the first one. *jde-neoc* uses *schemas* as the basic component of robot applications, which are combined in dynamic hierarchies to unfold the global behavior. Each schema is built separately into a plugin and dynamically linked to the framework when needed. It keeps its own graphical user interface. Some tools like a hierarchy oscilloscope and a 3D sensors-and-motors GUI have been created and added to the framework.

1 Introduction

Beyond the sensor and motor capabilities, the intelligence of a robot lies on its software. For simple behaviors almost any software organization works. If we want the robot to unfold complex behaviors or integrate several functionalities in the same system then to have good organization principles and a good software architecture makes the difference.

Robot programmers have to deal with heterogeneous hardware and software. There is no widely accepted software standards to de-

velop robot applications. In the last years, several frameworks and middleware have been created to help in that task. Robot manufacturers and private companies provide their own development kits. ARIA from ActivMedia, ERSP from Evolution Robotics, OpenR from Sony and Microsoft Robotic Studio are just a few examples. Many universities and research centers have also created their own frameworks. For instance, Player/Stage [7][10][19], Carmen [13], Marie [8], Miro [17], CLARAty [14], etc.

Each framework encapsulates functionality in different ways, providing different abstraction levels and making easier complex behavior generation. Modern middlewares provide methods to reuse code or behaviors in order to increase productivity. They impose several constraints to the organization of the robot software and split functionality into small building blocks or components that are easier to reuse.

Traditionally the organization of the robot capabilities to unfold autonomous behavior has been the focus of robotics research. Reactive, behavior-based, deliberative and hybrid paradigms are the most relevant schools. The cognitive architectures provide valuable guiding principles to organize the robot software. In addition, as any other computer science area, robot programming can also take advantage of the most advanced techniques and tools from the software engineering (object orientation, design patterns...).

Cognitive basis are interesting as they propose a methodology to face robot behavior

*This work has been funded by Spanish Ministerio de Ciencia y Tecnología, under the project DPI2004-07993-C03-01 and Comunidad de Madrid under the project RoboCity 2030: S-0505/DPI/0176

generation. They also provide abstractions easy to understand. This is important in academic environments with high programmer rotation, like ours, where the learning curve must be minimized. For all these reasons we developed the *jde-neoc* framework following our JDE cognitive architecture. After using *jdec* for three years we have found several limitations of this framework so we started the *jde-neoc* framework development. In this new platform we solved various drawbacks of *jdec*.

In the second section the cognitive architecture JDE underlying *jde-neoc* is briefly presented. It provides the schema and hierarchy concepts that will be used in the rest of the paper. Third section describes *jdec*, the first implementation of JDE, its features and limitations to develop robotic applications. Fourth section describes the *jde-neoc* framework, the new component-oriented implementation of JDE. Finally, some conclusions summarize the lessons learnt and current state of the project.

2 JDE cognitive architecture for robot applications

The idea of hierarchy has been widely used to cope with complexity in robotics. Hybrid cognitive architectures have successfully been used in the last years. Their ability to combine deliberation and reactivity is very convenient for robotic applications. The behavior-based architectures are another approach to the idea of hierarchy that have received support in several works [2][15][18].

Our software frameworks are all based on JDE, an ethology inspired and behavior-based cognitive architecture [6]. The goal of this architecture is to reduce the overall system complexity with a *divide and conquer* approach, similar to some hierarchies proposed by ethologist to explain the behavior generation in animals.

2.1 Schema as the behavior unit

The JDE main component and building block is the *schema* [1]. A *schema* is a task-oriented

piece of software that is executed independently. At any time there can be several schemas in execution. Each one is built to complete a particular task or to achieve some goal or mission. A schema in JDE: (1) is *tunable*, it accepts some parameters to *modulate* its behavior; (2) is an *iterative* process that makes its work by periodical iterations, providing an output at the end of each one; and (3) can be stopped or resumed at the end of any iteration.

To generate autonomous behavior in a robot perception and control must be faced. They both are complex and their fragmentation into smaller units reduces the complexity of the subproblems faced in each fragment and makes easier their reutilization. Accordingly, there are *perceptive schemas* and *motor schemas*. Perceptual ones transform sensor data or simple information into more complex stimuli that can be used by other schemas. Motor schemas access to perceptual data and generate control outputs which can be motor commands or activation signals for other low level schemas (perceptual or motor) and their modulation parameters.

Each schema has an associated state. The state defines the current schema's activation level. For example a perceptive schema can be in any of SLEPT or WINNER states. Motor schemas have preconditions, they can be in four different states: SLEPT, CHECKING preconditions, READY and WINNER. Those states are closely related to how action selection is made in JDE.

2.2 Combination in dynamic hierarchy

Hierarchy appears because a schema can take advantage of the functionality of others to perform its mission. This is implemented in JDE by means of coactivation and continuous modulation. This coactivation can be recursively repeated, so various levels appear, where the low level schemas are awakened and modulated by the higher ones.

The hierarchy that JDE proposes is not the classical one based on direct function invocation, where the father activates a son to carry out a mission and waits for the result while the

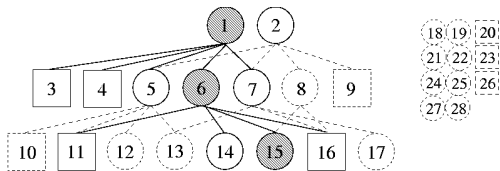


Figure 1: JDE hierarchy. Motor schemas are represented by circles and perceptual schemas by squares. Current WINNER schemas are shaded.

son does the job. Instead, JDE understands hierarchy as a co-activation that only means *pre disposition*. In JDE, a father can coactivate several sons at the same time, because this does not mean that all the sons gain control of the robot. Their real activation is left to an action selection mechanism that continuously selects which one gains control at each iteration, according to the current goals and current environment situation.

JDE claims that such hierarchical organization provides many advantages for robotics like bounded complexity for action selection, action-perception coupling and distributed monitoring. At the same time, this type of hierarchy does not loose the reactivity needed to face dynamic and uncertain environments.

This hierarchical activation is the skeleton of the collection of schemas, and several competitions among brothers take place, one at each level, to avoid incoherent behavior and contradictory commands to actuators. There is only one winner per level. A motor schema may command to actuators directly or may awake a set of new child schemas. These children will execute concurrently and they will in conjunction achieve the father's goal while pursuing their own. The continuous competition between all the actuation siblings determines whether each child schema will finally get the WINNER state or will remain silent in CHECKING or READY state. Only the winner, if any, passes to the WINNER state and is allowed to send commands to the actuators or spring its own child schemas.

The father activates the perceptive schemas that provide the information needed to solve the control competition between its actuation

children and the information needed for them to work and take control decisions. The chain of activations creates a specific hierarchy of schemas for generating a particular global behavior (figure 1). All awake schemas (CHECKING, READY and WINNER) run concurrently, similar to the distribution found in behavior-based systems.

Once the father has awoken its children it keeps itself executing, continuously checking its own preconditions, monitoring the effects of its current children, modulating them appropriately and keeping them awake, or maybe changing to other children if they can face better the new situation.

Hierarchies are specific for each global behavior. They are built and can be changed dynamically: among brothers at a given level, the current winner may change if the environment conditions or the final goals of the robot were modified. In such a case the whole hierarchy would also be modified: All the active schemas underneath the previous winner would then be consequently deactivated, and a new tree generated under the new winner.

3 Jdec: C implementation and its limitations

The JDE cognitive architecture was implemented, written in C language, in the *jdec* software platform. The reference hardware that *jdec* supports is the Pioneer robot of figure 3. *jdec* has been the framework for many robot applications in our group, both research and academic, for three years. Many schema based behaviors have been developed: person following [4], laser-based and vision-based localization, Virtual-Force-Field reactive navigation, Gradient-Path-Planning deliberative navigation [5], etc¹.

3.1 Schema

In *jdec* the schemas are implemented as threads, one per schema. All of them follow the skeleton shown at figure 2. When active,

¹More information can be found at www.robotica-urjc.es

```

initialization code
loop
  if (slept) stop_the_schema
  action_selection
    check preconditions
    check brother's state
    if (collision OR absence)
      father_arbitrates
    if (winner) then schema_iteration
  msleep
end_loop

```

Figure 2: Pseudo-code of an schema in jdec

each schema executes iterations. All the task dependant code lies in the iteration function, which is called periodically at a controlled frequency.

Following the JDE action selection mechanism, a motor schema continuously checks its preconditions and the state of its brothers. In case of none (control collision), two or more brothers (control overlap) fulfilling their preconditions, it invokes the arbitration function at the father level. The schema that wins the current control competition at that level of the hierarchy gains the WINNER state and executes its `schema_iteration` in that iteration. Perceptive schemas do not compete in the action selection and always gain the WINNER state without problems.

The iterative execution avoids excessive CPU consumption and forces to design the application in a reactive way. For instance, instead of having a "rotate-90°", command we prefer the loop "rotate-rotate-...-rotate-stop", where the iteration realizing that no more rotation is need directly stops the motors. This is very convenient to reactive applications and also provides room to deliberative schemas that use plans as resources instead of explicit courses of action.

3.2 Hierarchy

Each schema provides a set of *shared variables* to communicate with other schemas. Such communication is carried out by shared mem-

ory in a very efficient way, using mutexes to prevent race conditions. This is fully asynchronous and straightforward as all schemas are threads of the same process.

First, the schema defines and updates continuously its output variables when is in WINNER state. They are offered to other schemas, which can read them. For instance, they can be used to store the outcome of a perceptive schema. Second, the schema defines and continuously reads its modulation variables when WINNER. Other schemas may write there the modulation to bias the current behavior of the schema, mainly its father. The interaction is not constrained to a given instant (as in the parameters of a function invocation) but carried out as a continuous modulation, which may change from one iteration to the next.



Figure 3: Reference robot, sensors and actuators

The API to the robot hardware itself (figure 3) is a set of global variables: on the one side sensor variables like encoders, laser, etc. that schemas may read and, on the other side, motor variables like rotation and translation speeds, position of the pantilt, etc. that schemas may write.

3.3 Limitations

To write a robot application the programmer has to design it in schema terms. Each schema is written in two separate C files: `myschema.h` with the declaration of shared variables of the schema, and `myschema.c` with their definitions and implementation. Both are compiled together in a single C object module. All the schemas of the application are statically linked together in the executable.

In order to speed up the development, there is a schema template with common parts of code ready to reuse, so the programmer focuses herself just on the iteration function of her schemas, their preconditions and their arbitration functions.

Although schemas were designed to ease the component reuse, in practice, such a reuse in *jdec* is still a difficult task. Each application starts from the bare software platform and adds its own schemas. There is a strong coupling between schemas and it is difficult to remove the dependences between object modules. Moreover, adding the graphical interface of a schema requires the modification of the system GUI, as it is unique for the whole application.

Using shared variables opens the door to name collisions. All the schema variables are joint in a single name space. The variable names must be unique in a given application, but the system does not provide mechanisms to detect that two schemas in the application offer different variables with the same name.

4 Jde-neoc software architecture

With the limitations of the first implementation of JDE in mind, a second one has been designed and developed from scratch. It is named *jde-neoc* [16]. Its main goal was to improve the component-orientation of the framework in order to ease the development of new robot applications and favor the component reuse. Another goal was to replace the old-fashioned libraries underlying *jdec* by better, standard and more portable libraries like Glib, GTK for visualization, etc. New tools for de-

bugging and development have been also programmed and added to the framework, like the hierarchy oscilloscope and the sensors-and-motors GUI.

4.1 Modules

In *jde-neoc* each schema is compiled separately into a plugin, loaded and linked at runtime to the framework. This provides more flexibility to the system. The components to load for a given application are specified in a configuration file, the framework will load them at the beginning of its execution.

In addition, the applications become more modular as they are always composed of the framework executable and the collection of plugins for the application itself (for instance `schema1.so` and `schema2.so` files). Instead of having one executable file per application, in *jde-neoc* the executable is always the same, and it does not require to be rebuilt for each application. Only the set of plugins change from one application to another.

Another advantage of this approach is that improving the code of the schema does not require the recompilation of all the applications that depend on it. It only requires building the new release of the plugin. The programs will dynamically link to the improved schema when loading the new release.

While in *jdec* there was a unique name space, with all the symbols linked at building time, in *jde-neoc* each schema keeps its own name space or symbol table. If one schema needs some variable defined by another one, it must explicitly import it at run time before using it. If one schema offers some variable to the others, it must explicitly export it. The *jde-neoc* infrastructure keeps updated a list of shared variables, with all the variables exported by the schemas. It provides two functions, `getSchemaVar` and `putSchemaVar`, to import and export the variable's symbol respectively. This way no name collision can occur as long as there can be two or more variables with the same name, but they will belong to different schemas, and so, they can be treated independently. Unresolved dependencies on those variables are properly reported

when found.

For other symbols required for one schema to run but not provided by other schemas, i.e. symbols defined at a library, the linking is the regular one, at building time.

The schema interface is more strictly enforced than in *jdec*. Besides the exported variables, each schema must define an API with several standard symbols and functions that allow its use by other schemas and its integration in the hierarchy. For instance: `schema_startup` and `schema_suspend` functions allow other schemas to activate or sleep it, `schema_interval` variable is the modulation parameter that determines the frequency of the schema iterations.

Regarding implementation details, instead of using directly the `dlopen` and `pthread` libraries, *jde-neoc* uses `GModule` and `GThreads` types, both inside `Glib`, to implement the plugins and the multithreading (one thread per schema). This way the framework is more portable than the old one.

4.2 Distributed visualization

Following the modular design, the Graphical User Interface in *jde-neoc* is already distributed in several GUIs. Every schema may integrate its own visualization code and show its own window. There is no single window for the whole application, and so, adding new schemas does not require the reprogramming of any existing GUI. That was a drawback of *jdec*, where the GUI was shared for all the available schemas.

The visualization is considered optional, an schema may or may not have visualization code. In case of having it, its code is included in the plugin. In addition, the schema GUI can be activated or deactivated at will. Often it is useful only while debugging, but not while the robot is in operation. The schema API includes two functions: `schema_gui_startup` and `schema_gui_suspend` to activate or hide it at will at run time. They may be empty if the schema does not offer visualization at all.

Regarding implementation details, the old-fashioned XForms graphical library has been replaced by GTK, a widely used library more

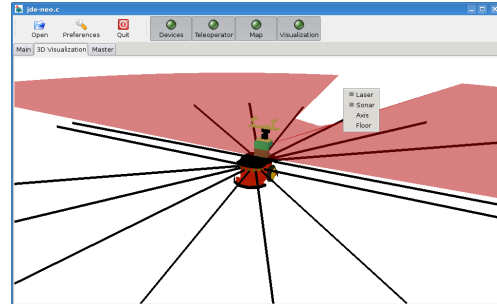


Figure 4: 3D visualization of the robot

powerful and standard. New visualization code has also developed to show 3D sketches of the robot, its laser and sonar readings, and the world around (figure 4). This display is nicer than the old 2D one and uses the standard OpenGL libraries. These libraries reduce the computational load of visualization as it is carried out on the Graphics Card not the main CPU.

4.3 New tools

Two new tools have been created and added to the *jde-neoc* framework. They makes easier the development of robotic applications.

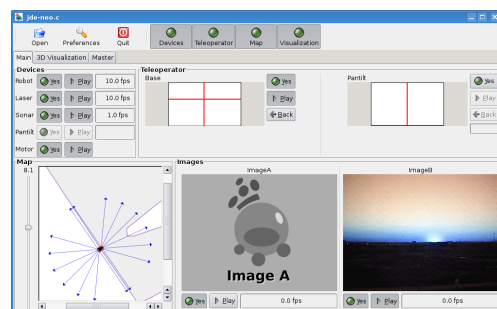


Figure 5: Tool for sensor visualization and motor teleoperation

First, the visualization tool shown at figure 5, named sensors-and-motors GUI. It displays the sensor values in a graphical and intuitive way, for instance images from the robot cameras, laser readings, etc.. It also allows teleop-

eration for robot motors, both the robot base and the pantilt unit (red crosses in the upper part of figure 5). This tool is useful to check the proper functioning of sensors and motors, and to manually move the robot without pushing or pulling it.

Schema	State	Show	Play	Cycle time (milliseconds)
example	winner	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	75,0
goon	active	<input type="checkbox"/>	<input type="checkbox"/>	200,0
stop	winner	<input type="checkbox"/>	<input type="checkbox"/>	200,0
vff	active	<input type="checkbox"/>	<input type="checkbox"/>	200,0
padre	winner	<input type="checkbox"/>	<input checked="" type="checkbox"/>	1000,0
hijo1	active	<input type="checkbox"/>	<input type="checkbox"/>	1000,0
hijo2	active	<input type="checkbox"/>	<input type="checkbox"/>	1000,0

Figure 6: Hierarchy oscilloscope

Second, the management tool at figure 6 shows the collection of the loaded schemas, their current state, its cycle time among iterations and the current hierarchical relationships among them. It allows the manual activation and deactivation of their schemas (“play” column) and their GUI (“show” column). This is very convenient when debugging the preconditions of a set of schemas, to see which one really wins the competition at each level depending on the current goals and environment situations.

5 Example

Beyond programming the whole *jde-neoc* infrastructure a toy example has been also developed, just to check the framework mechanisms for hierarchy, schema use, shared variable list, etc. The whole infrastructure are about 7000 lines of C code, including several drivers, in particular one that connects *jde-neoc* to SRIsim simulator.

The toy behavior is the safe reactive navigation, and it is generated with four schemas. The father, named **example**, has null preconditions and it is alone at its level, so it is always in WINNER state. The human user clicks with

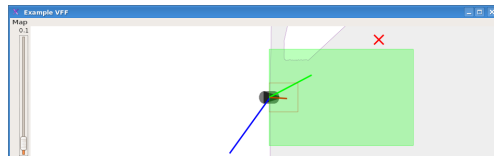


Figure 7: GUI of the **example** schema

the mouse on its GUI to specify the destination of the safe navigation (red cross at figure 7). It coactivates three different actuation schemas: **go-on**, **vff** and **stop**. Their preconditions are approximately disjoint, so depending on the distance to the closest obstacle only one of them will win the selection competition and gain control of robot motors.

The **go-on** schema drives the robot towards the destination point at high speed. Such destination point is its modulation parameter and can be asynchronously changed by its father. Its precondition is to have the rectangular shaded area around the robot at figure 7 free of obstacles. The **vff** schema drives the robot to avoid a close obstacle while approaching to the destination point. Such point is also a modulation parameter of this schema. Its precondition is to have an obstacle in the shaded area. Finally the **stop** schema brakes the robot motors when there is any obstacle in a very close area around the robot (the inner rectangle inside the shaded area at figure 7).

Schema	State	Show	Play	Cycle time (milliseconds)
example	winner	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	75,0
goon	checking	<input type="checkbox"/>	<input type="checkbox"/>	75,0
stop	winner	<input type="checkbox"/>	<input type="checkbox"/>	200,0
vff	ready	<input type="checkbox"/>	<input type="checkbox"/>	75,0

Figure 8: Hierarchy state at given time

The hierarchy oscilloscope snapshot at figure 8 shows the state of the system at a given time: the **example** schema is the only one at the upper level, is WINNER. It has three active children: **go-on** in CHECKING state as its preconditions are not fulfilled, **vff** in READY state as its preconditions are fulfilled, but it loses the control competition against **stop** schema. This matches a situation where there is an ob-

stacle both inside the shaded area and inside the inner rectangle. The control collision is detected by the children and the father, through its arbitration function, chooses the winner between `stop` and `vff`.

6 Conclusions

A new component-oriented software framework for robotic applications, named *jde-neoc*, has been presented. It is an implementation of the behavior-based cognitive architecture JDE and it was designed to overcome some of the limitations observed in the prior JDE implementation, *jdec*.

jdec proved to be good for reactive behaviors and those not requiring a complex architecture. However it was difficult to generate complex behaviors in *jdec* because it had some limitations for the code reuse and integration.

In *jde-neoc*, the perception and control are distributed among a collection of schemas. Each schema is a software component with a clear API that is built as a plugin on a separate file. The schema has its own name space, the symbols required from other schemas are dynamically imported and those offered to other schemas explicitly exported through a shared variable list that *jde-neoc* provides. This approach provides great flexibility to the robot application development. Collision in variable names, typical at *jdec*, are no longer a problem. There is no need to rebuild the whole system after a schema improvement, only that plugin must to be recompiled.

The visualization is also distributed in schemas and can be (de)activated at will in run time. It is optional, as the schema may or may not have visualization code. In case of having it, it will be included in the plugin.

Several powerful and standard libraries, like Glib and OpenGL, have been selected as the base for *jde-neoc*, making it more portable. In addition, new tools have been developed and added to the framework: a sensors-and-motors GUI and a hierarchy oscilloscope. They make application developing and debugging easier.

jde-neoc is an on going project. The core

software architecture has been designed and fully implemented in C. We are currently migrating from *jdec* to *jde-neoc* the drivers for the different sensors, actuators, robots, and simulators available at our laboratory. In the near future we intend to develop new behaviors using this infrastructure in order to get the experimental feedback about its real usefulness.

References

- [1] Arkin, R., *Behavior Based Robotics*, The MIT Press, 1998.
- [2] S. Behnke and R. Rojas, *A hierarchy of reactive behaviors handles complexity*, Balancing Reactivity and Social Deliberation in Mult-Agent Systems, LNCS 2103 Springer, 2001, pp 125-136.
- [3] J. Bryson and L. Stein, *Modularity and design in reactive intelligence*, Int. Joint Conf. on Artificial Intelligence IJCAI-2001, Seattle (USA), 2001, pp 1115-1120.
- [4] R. Calvo, J.M. Cañas and L. García-Pérez, *Person following behavior generated with JDE schema hierarchy*, ICINCO 2nd Int. Conf. on Informatics in Control, Automation and Robotics, Barcelona (Spain), 2005, pp 463-466.
- [5] J. Cañas, R. Isado and L. García-Pérez, *Robot navigation combining the Gradient Method and VFF inside JDE architecture*, VI Workshop de Agentes Físicos, WAF-2005, Granada (Spain), 2005, pp. 153-160.
- [6] J. Cañas, V. Matellán, *Integrating behaviors for mobile robots: an ethological approach*, Cutting Edge Robotics, Pro Literature Verlag / ARS, 2005, pp 311-330.
- [7] T. Collett, B. MacDonald and B. Gerkey, *Player 2.0: Toward a Practical Robot Programming Framework*, Australasian Conf. on Robotics and Automation (ACRA 2005), Sydney (Australia), 2005.

- [8] C. Cote, Y. Brosseau, D. Letourneau, C. Raievsky and F. Michaud, *Robotic software integration using MARIE*, Int. J. of Advanced Robotic Systems, vol. 3(1), 2006, pp 55-60.
- [9] A. Farinelli, G. Grisetti and L. Iocchi, *Design and implementation of modular software for programming mobile robots*, Int. J. of Advanced Robotic Systems, vol. 3(1), 2006, pp 37-43.
- [10] B. Gerkey, R. Vaughan and A. Howard, *The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems*, 11th Int. Conf. on Advanced Robotics (ICAR 2003), Coimbra (Portugal), 2003, pp 317-323.
- [11] M. Hattig, I. Horswill and J. Butler, *Roadmap for mobile robot specifications*, 2003 IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003), Las Vegas (USA), 2003, pp 2410-2414.
- [12] G. Metta, P. Fitzpatrick and L. Natale, *YARP: Yet Another Robot Platform*, Int. Journal of Advanced Robotic Systems, vol. 3(1), 2006, pp 43-48.
- [13] M. Montemerlo, N. Roy and S. Thrun, *Perspectives on standarization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) toolkit*, 2003 IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003), Las Vegas (USA), 2003, pp 2436-2441.
- [14] I. Nesnas, A. Wright, M. Bajracharya, R. Simmons and T. Estlin, *CLARAty and challenges of developing interoperable robotic software*, 2003 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS-03), vol. 3, 2003, pp 2428-2435.
- [15] M. Nicolescu and M. Mataric, *A hierarchical architecture for behavior-based robots*, Int. Joint Conf. on Autonomous Agents and Multiagent systems, Bologna (Italy), 2002, pp 227-233.
- [16] J. Ruiz-Ayúcar, *Jdeneo.c - Una plataforma de desarrollo de aplicaciones robóticas*, Degree Project, Universidad Rey Juan Carlos, 2007.
- [17] H. Utz, S. Sablatnög, S. Enderle and G. Kraetzschmar, *Miro - Middleware for mobile robot applications*, IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures, vol. 18, no. 4, 2002, pp 493-497.
- [18] H. Utz, G. Kraetzschmar, G. Mayer and G. Palm, *Hierarchical behavior organization*, 2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS-05), Edmonton (Canada), 2005.
- [19] R. Vaughan, B. Gerkey and A. Howard, *On Device Abstractions For Portable, Reusable Robot Code*, IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003), Las Vegas (USA), 2003, pp 2421-2427.
- [20] E. Woo, B. MacDonald and F. Trépanier, *Distributed mobile robot application infrastructure*, 2003 IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003), Las Vegas (USA), 2003, pp 1475-1480.