Jde+: an open-source schema-based framework for robotic applications

J.M. Cañas, D. Lobato & P. Barrera
Robotics Group, Universidad Rey Juan Carlos

***Abstract*— In this paper we present our object-oriented framework *jde+*, which is the second implementation of our cognitive behavior-based architecture JDE. This framework uses *schemas* as the building block of robot applications. They are combined in dynamic hierarchies to unfold behaviors. The schema interfaces and the distributed action selection mechanism are key issues in this hierarchical composition. The lessons learnt in three years using the first implementation are also described. For instance, the schema communication has changed from shared memory to message passing, dynamic load of schemas replaces the static linking, and a hierarchy oscilloscope has been introduced for debugging purposes.**

I. 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 develop 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, Open-R 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 [8][14][24], Carmen [17], Marie [10], Miro [22], CLARAty [18], 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...).

This work was funded by Spanish Education and Science Ministry under project DPI2004-07993-C03-01 and by Comunidad de Madrid under project RoboCity2030 S-0505/DPI/000176.

There are two kinds of robot middleware: those with underlying cognitive basis and those without it. A framework that has no underlying cognitive architecture is developed using purely software engineering criteria, such as scalability or ease of reuse. However it lacks of the theoretical basis to lead the design process. This can become a major limitation for complex behaviors, as long as the software engineering approach has been unable to solve the whole robot programming problem. Architectures that grow on cognitive basis have stronger design ways to divide complexity, for instance using models extracted from other fields for animal or human behavior.

Cognitive basis are interesting as they propose a methodology to face robot behavior generation. They also provide abstractions easy to understand. This is important in academic environments with high programmer rotation where the learning curve must be minimized. For all these reasons we developed the *jdec* 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+* framework development. In this new platform we solved various drawbacks of *jdec* and we also included software engineering techniques to increase our productivity.

In the second section the cognitive architecture JDE underlying *jde+* 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+* framework, the new object-oriented implementation of JDE. Finally, some conclusions summarize the lessons learnt and current state of the project.

II. JDE COGNITIVE ARCHITECTURE

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 reactiveness is very convenient for robotic applications. The behavior-based architectures are another approach to the idea of hierarchy. They have received new support in several works [1][19][23].

Our software frameworks are all based on JDE, an ethology inspired and behavior-based cognitive architecture [7]. Its name comes from its acronym in Spanish: *Dynamic Schema Hierarchy*. 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.

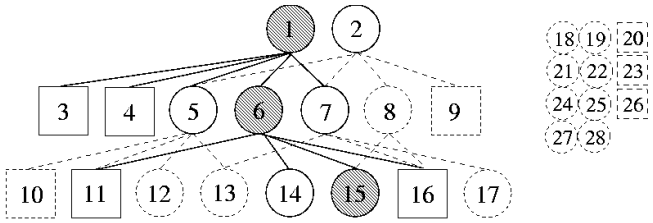

Fig. 1. JDE hierarchy. Motor schemas are represented by circles and perceptual schemas by squares. Current WINNER schemas are shaded.

A. Schema as the behavior unit

The JDE main component and building block is the *schema*. A *schema* is a task-oriented piece of software that is executed independently. Everything in the system is represented as a schema. 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. 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.

Perception and control are the two behavior components, related but different. Both are complex and have to be solved in separated modules of the program. The fragmentation into smaller units makes also easier the code reuse and reduces the complexity in the subproblems faced in each fragment. It also allows locating several schemas in different processors, facilitating distributed implementations.

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. A motor schema, on the other hand, can be in four different states: SLEPT, CHECKING, READY and WINNER. Those states are closely related to how action selection is made in JDE. Control schemas have preconditions.

In addition to its task-oriented nature, a schema has other characteristics in JDE: (1) it is *tunable*, it accepts some parameters to *modulate* its behavior; (2) it is an *iterative* process that makes its work by periodical iterations, providing an output at the end of each one; and (3) it can be stopped or resumed at the end of any iteration.

B. Schemas are combined in dynamic hierarchy

Once the basic behavior unit has been introduced, there are many options to assemble the whole system. Each JDE schema has its own goal, and performs a particular function in which it is an expert. 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 activation and continuous modulation. This activation can be recursively repeated, so various levels appear, where the low level schemas are awaken and modulated by the higher ones. The chain of activations creates a specific hierarchy of schemas for generating a particular global behavior.

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 son does the job. Instead of considering the mission of the son as a step in the sequential plan of the father, JDE understands hierarchy as a co-activation that only means *predisposition*. In JDE a father can activate several sons at the same time, because this does not mean that the sons gain control of the robot, just that they are awaken. Their real activation is left to an action selection mechanism that selects which son must be finally used. This selection is done by competition among brothers in the same level.

All awake schemas (CHECKING, READY and WINNER) run concurrently, similar to the distribution found in behavior-based systems. To avoid incoherent behavior and contradictory commands to actuators JDE proposes hierarchical activation as the skeleton of the collection of schemas. It claims that such hierarchical organization, in the ethological sense, provides many other advantages for robotics like bounded complexity for action selection, action-perception coupling and distributed monitoring. All without losing the reactivity needed to face dynamic and uncertain environments.

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. Actually, that's why the father awoke such schemas, and not others. A 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. This recursive activation of perceptive and actuation schemas conforms a schema hierarchy (figure 1).

Once the father has awaken 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.

Several instances of the same schema can be activated simultaneously or at different moments, probably by different fathers, using different modulations and running in different levels of the hierarchy. This is an example of re-usability.

Hierarchies are built and modified dynamically and are specific for each behavior. Among brothers at a given level, the winner could change if the environment conditions or the final goals of the robot were modified. This would change the one selected, and consequently the whole hierarchy underneath would also be modified: All the active schemas underneath the previous winner would then be deactivated, and a new tree generated under the new winner.

```
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
```

Fig. 2. Pseudo-code of an schema in jdec

Reconfigurations in JDE use to be very fast, given that the arbitration and the decisions made by the schemas are made periodically and at a high enough rate.

III. JDEC SOFTWARE PLATFORM

The JDE cognitive architecture was implemented, written in C language, in the *jdec* software platform. *jdec* has been the framework for several robot applications [5], [6], both research and academic, for three years.

A. Schema

In *jdec* the schemas are implemented as threads (pthreads library in GNU/Linux), one per schema. All of them follow the skeleton shown at figure 2 in pseudo-code. When active, 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, the schema continuously checks its preconditions and the state of its brothers. In case of control overlap with brothers or control absence, 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. Passive perceptive schemas always gain the WINNER state at each iteration, as they do not generate control conflicts.

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; 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.

B. Hierarchy

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

The API to the robot hardware itself 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, that schemas may write.

In addition, each schema provides four *compulsory functions* that allow the hierarchical activation and deactivation: `startup`, `resume`, `suspend` and `iteration`. `Startup` is called by the system to initialize the schema. `Resume` and `suspend` allow other schemas to sleep or activate the schema. `Resume` call has two main parameters: the list of brothers (as long as the same schema can be used in different contexts) and the arbitration function from the father. When a father loses the competition control at its level it deactivates its children calling their `suspend` functions.

C. Limitations and lessons learnt

jdec has been the software platform for many robot applications with the Pioneer robot at our lab. Many schema based behaviors have been developed in the last three years: person following [5], laser-based and vision-based localization, Virtual-Force-Field reactive navigation, Gradient-Path-Planning deliberative navigation [6], etc[1].

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 the four mentioned functions, 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.

One lesson learnt with *jdec* is the importance of distribution. Despite of having the functionality split in several schemas, all of them have to run in the same machine. This imposes a computational bottleneck if, for example, the application deals with vision. Two network servers (otos and oculo) have been developed to provide remote access to sensors and actuators. Using them the application can be placed at a different computer from those with the sensors and motors attached. But in essence *jdec* has centralized execution and so, it is computationally limited.

Although schemas were designed to ease the component reuse, in practice, such a reuse in *jdec* is still a difficult

[1]More information can be found in http://www.robotica-urjc.es

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 application. Truly distributing the GUI is necessary to simplifying the reuse of schemas.

There are also limitations inherent to the C language. For example, 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 with mechanisms to detect that two schemas in the application offer different variables with the same name.

This software implementation does not capture in full extent the capabilities and constraints imposed by the cognitive architecture. For example, in *jdec* there is only one thread per schema and no chance for simultaneous schema instances, which is explicitly allowed by the cognitive JDE architecture. In addition, the data communication is bounded to father-son interactions in JDE, but it is not restricted in *jdec*. One schema can read the perceptive variables defined by another schema, even if they are not related at all, or even while the second one is slept and its variables are not updated. There is no protection mechanism so errors are more difficult to debug.

IV. JDE+ FRAMEWORK

Jde+ framework is the new implementation of the JDE cognitive architecture, written in C++ programming language. We chose this language because its time performance and to reuse most of the work done with *jdec* but taking advantage of the object oriented paradigm and its benefits. The *jde+* implementation follows a careful software design and adds new features like schema dynamic loading, a debugging tool and automatic code generation through a simple XML definition of schemas.

A. Schema

In *jde+* the schemas are implemented as objects with their own execution flow. To simplify the design, a schema class is composed by a set of classes, each one with a particular functionality. The abstract design is shown in figure 3 which contains the five most important pieces of *jde+*. Three of them are directly related with the building unit: `schemainstance`, `schemaimplementation` and `schemainterface`.

`Schemainstance` represents a schema instance, specifically the instance common part (schema identification, state, etc.). The programmer of a given robotic application must know this main class because it defines the API to work with the system (send messages, get relationship information, manage perception and modulation data, etc.). Using classes, it is straightforward to have several different instances of the same logical schema activated at the same time in different places of the hierarchy, as proposed by the cognitive architecture. *State* design pattern [13] is applied to represent

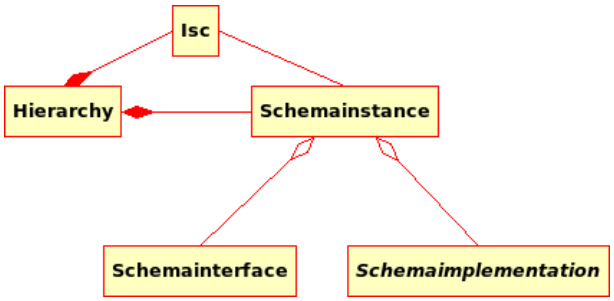

Fig. 3. Class diagram of jde+

schema state of an instance and different behavior for each state. Moreover, to achieve concurrent interactions *Active Object* design pattern [21] concepts are applied to avoid locking on `Schemainstance` methods invocation.

`Schemainterface` defines what that schema offers to the others, and the way to use it (based on the *Interface* design pattern). In addition, a schema defines its dependencies enumerating the interfaces that it needs in order to achieve its own target. The interfaces have a name and a description of the data they contain. This way, *jde+* allows a formal definition of what a schema provides and what it requires. This emphasis on clear interfaces for information exchange has been widely acknowledged in other component-based robotic frameworks [11][12].

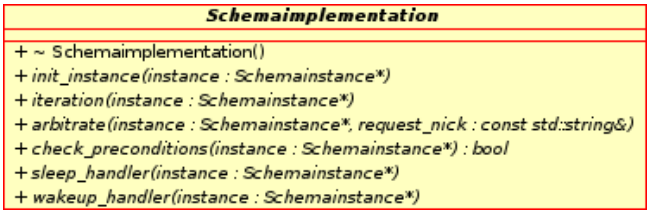

Fig. 4. Schema implementation class

Finally, `schemaimplementation` is an abstract class that user must derive to add the particular behavior of the schema. It defines several abstract methods that must be filled with code in order to implement the particular schema behavior. Figure 4 shows a detail of `schemaimplementation` and the methods that the application programmer has to implement.

This design allows the user to create schemas defining which `schemainterface` they provide and which `schemainterfaces` they need to operate. The schema's own code is confined to the `Schemaimplementation` class.

All schemas are compiled separately as plugins and placed together in several directories. The set of schemas in such libraries provide all the functionality to the system. At runtime, when a schema requires a given interface, the system searches for schemas that provide that interface, looking at some directories. When it finds the right schema, creates an instance of it, and recursively resolves its dependences too.

To locate a particular schema it is only necessary to know its interface and the system will load the correct schema from the libraries. Some concepts from *Abstract Factory* design pattern [13] and object auto registration [2] are applied to the design of these mechanisms.

This approach allows easy software reuse because you can use any existing schema only knowing its interface. Moreover, it allows easy system maintenance because you can reimplement schema internals keeping the same interface, without further recompiling or linking of other parts of the system.

B. Hierarchy

As seen in figure 3, *jde+* has two more pieces, the `hierarchy` and the `isc`. They are responsible for managing the schemas and providing all the *glue* between them. *Hierarchy* class is responsible for containing all instances and their relationships. `Isc` class contains the communication mechanisms and allows the interaction between schema instances.

Each schema instance interacts with others through a message passing mechanism. This mechanism is provided by *jde+*, specifically with the `isc` class. Communication is asynchronous to allow schema execution flows to run decoupled. This decoupling has been identified as desirable in other component based architectures [16].

There are five types of messages, two top-down messages and three bottom-up ones. The top-down messages are `modulation` and `state_change_request`. They flow from a father to its children, which asynchronously receive them and react properly. For instance, the father activates a son sending it a `state_change_request` to the schema instance that implements one of the interfaces it depends on. This triggers the instance creation. The bottom-up messages are `perception`, `arbitrate_request` and `state_change_notification`. A perceptive schema uses `perception` messages to send new computed values of the information it elaborates to its father. `arbitrate_request` messages are generated when a control overlap or absence is detected. `state_change_notifications` are sent from one schema instance to its father when the instance changes its state, for instance from READY to CHECKING.

Hierarchy also provides a dynamic schema loading mechanism based on dynamic class loading. With this mechanism the framework is separated from the schemas so new schemas can be added without rebuilding the whole system. All schemas are dynamically linked in runtime. This feature adds flexibility to the development.

Hierarchy class applies `Monitor Object` design pattern [21] on its public interface, allowing multiple method invocations from schema instances.

C. New programming tools

Jde+ framework also includes some add-on tools that simplify the robot application development process. The first one is a tool to debug a live hierarchy and the second one is a parser of schema definitions in XML files that automatically generates most of the schema code.

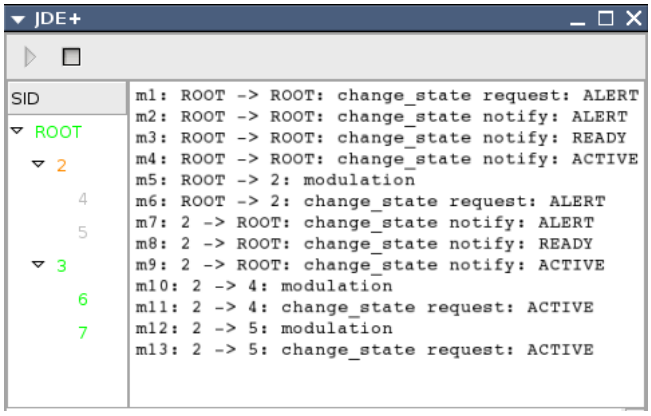

Fig. 5. Oscilloscope of the hierarchy of schemas

Jde+ message passing mechanism through a central communicator class (`isc`) allows sniffing schemas interactions. In this way it is simple to log schema traffic and to verify the system operation. The tool is similar to an oscilloscope that can trace the behavior generation. The tool is called *jdescope* and it is used as the main GUI. It allows schema loading, select a root, and starts or stops the hierarchy. When the hierarchy is running, *jdescope* logs all messages generated in the system. Figure 5 shows a simple hierarchy and several message logs. The hierarchy in the figure has seven schemas. Root schema starts running when *jdescope* sends it a `stage_change_request` message to CHECKING (=alert in figure 5), this is logged as a ROOT to ROOT message. Later on, the schema promotes its state to WINNER (=active in figure) and activates and modulates child 2 that does the same with its own children, 4 and 5.

The second tool provided is an XML parser that automatically creates most of the code needed to implement a new schema. We have defined a simple syntax to specify interfaces and schemas in XML files. This way the programmer of robotic applications can write XML definitions for schema interfaces and schema implementations. The parser validates definition through an XML-Schema template and then generates C++ code using these definitions. After that, a template for the C++ schema has been created and is ready to be filled. Makefiles and other auxiliary files are also generated. A simple XML definition snippet is shown in figure 6. In this example, the *S1* schema implements the *I1* interface and depends on *I2* and *I3* interfaces. Interfaces are in a dynamic library called *interfaces* and two external files, *util.h* and *util.cpp*, are included to provide some utility functions. The parser will generate three files: *S1.h* with the declarations, *S1.cpp* with empty definitions and *Makefile* to build the schema.

V. CONCLUSIONS

A new object-oriented software framework for robotic applications, named *jde+*, has been presented. It is an

```
<schemaDef name="S1">
  <interface name="I1">
    <library>interfaces</library>
  </interface>
  <child>
    <interface name="I2">
      <library>interfaces</library>
    </interface>
  </child>
  <child>
    <interface name="I3">
      <library>e1interfaces</library>
    </interface>
  </child>
  <extheader>util.h</extheader>
  <extsource>util.cpp</extsource>
</schemaDef>
```

Fig. 6. XML schema definition

implementation of the behavior-based cognitive architecture JDE, which uses the schema as the building block for new robot behaviors and combines them in hierarchies to scale in complexity.

It was designed to overcome the limitations observed in the prior JDE implementation, *jdec*. *jdec* probed to be good for reactive behaviors and those not requiring a complex architecture. It allowed flexible sequences of behavior and elaborated behaviors using simple schemas. However it was difficult to generate complex hierarchies in *jdec* because it had some limitations on the code reuse and integration. For this reason, *jdec* was not tested in complex scenarios.

In *jde+* each schema is a component with a clear message API. The schema interfaces are named, explicitly declared and linked to the schemas, both to those that provide them and to those that require them. A well defined API simplifies the code reuse problem, so *jde+* is expected to be the base for more complex behaviors. It also provides new features and tools like dynamic load of schemas, XML schema definition and a hierarchy oscilloscope that makes both developing and debugging easier.

jde+ 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+* 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] S. Behnke and R. Rojas, A hierarchy of reactive behaviors handles complexity, *Balancing Reactivity and Social Deliberation in Mulit-Agent Systems*, LNCS 2103 Springer, 2001, pp 125-136.

[2] J. Beveridge, Self-Registering Objects in C++, Dr. Dobb's Journal, pp. 38-41, August 1998, Vol. 23, Issue 8.

[3] J. Bryson and B. McGonigle, Agent Architecture as Object Oriented Design, *4th Int. Wshp. on Intelligent Agents: Agent Theories, Arquitectures and Languages ATAL'97*, SpringerVerlag, 1997, pp 15-29.

[4] 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.

[5] 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.

[6] J. Cañas, R. Isado and L. García-Pérez, Robot navigation combining the Gradient Method and VFF inside JDE architecture, *VI Workshop de Agentes Fsicos, WAF-2005*, Granada (Spain), 2005, pp. 153-160.

[7] J. Cañas, V. Matellán, Integrating behaviors for mobile robots: an ethological approach, *Cutting Edge Robotics*, Pro Literature Verlag / ARS, 2005, pp 311-330.

[8] 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.

[9] C. Cote, D. Ltourneau, F. Michaud, J. Valin, Y. Brosseau, C. Raievsky, M. Lemay and V. Tran, Code reusability tools for programming mobile robots, *2004 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS-04)*, Sendai (Japan), 2004

[10] C. Cote, Y. Brosseau, D. Letourneau, C. Raievsky adn F. Michaud, Robotic software integration using MARIE, *Int. J. of Advanced Robotic Systems.*, vol. 3(1), 2006, pp 55-60.

[11] A. Cowley, L. Chaimowicz and C. Taylor, Design minimalism in robotics programming, *Int. J. of Advanced Robotic Systems.*, vol. 3(1), 2006, pp 31-36.

[12] 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.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides and G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1998

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] A. Orebäck and H. Christensen, Evaluation of architectures for mobile robotics, *Autonomous Robots*, vol. 14, pp 33-49.

[21] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, vol. 2, John Wiley and Sons, 2000

[22] H. Utz, S. Sablatng, 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.

[23] 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.

[24] R. Vaughan, B. Gerkey and A. Howard, On Device Abstractions For Portable, Resuable Robot Code, *IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003)*, Las Vegas (USA), 2003, pp 2421-2427.

[25] E. Woo, B. MacDonald and F. Trépanier, Distributed mobile robot application infraestructure, *2003 IEEE/RSJ Int. Conf. on Intelligent Robot Systems (IROS 2003)*, Las Vegas (USA), 2003, pp 1475-1480.