Integrating Behaviors for Mobile Robots. An ethological approach.

José María Cañas Plaza and Vicente Matellán Olivera
Robotics Lab. Grupo de Sistemas y Comunicaciones (GSyC)
Universidad Rey Juan Carlos. 28933 Móstoles (Madrid), SPAIN

1. Introduction

Robots available nowadays in the everyday markets can be divided into two major groups. First, those that are oriented to a single task, as vacuum cleaners (roomba[1], robocleaner[2], ...), lawn mowers (automower[3], robomower[4], ...), etc; and second, those oriented to various tasks, as robotic pets (Aibo[5], Necoro[6],...), museum guiders, or research platforms (Pioneer, Koala, etc.). In order to get service robots or personal assistants we need to improve the abilities of the second ones. These abilities have to do with their ability to smartly combine their basic skills to obtain behaviors that are more complex.

The generation of autonomous behavior is a very complex issue. Within a multi-goal system, like service robots, the focus is more on the integration than on the control of perception algorithms. The organization becomes the critical issue; robustness and flexibility are key features. As animals can do it, we can see no compelling reason why robots could not, but more research on robot architectures is needed to reach an acceptable performance.

Several paradigms have been historically proposed for behaviour generation in robots. These paradigms are also known as architectures. Dynamic and uncertain environments forced the evolution from symbolic AI (Nilsson 1984) to reactive and behavior based systems (Brooks 1986; Arkin 1989).

The behaviors based systems adapt smoothly. They have no anticipation and no state, but they have shown poor scalability for complex systems. Hybrid architectures have been predominant since mid 90s (Simmons 1994; Konolige 1998; Bonasso 1997), mainly those three-tiered ones that add two layers to behavior based ones, usually a sequencer, and a deliberator.

Several architectures have been explored after the hybrid three-tiered architectures became the de facto standard. In particular, many reviews of the hierarchy principle have been proposed in last years (Arkin 2003, Saffiotti 2003, Nicolescu 2002, Behnke 2001, Bryson 2001), trying to overcome subsumption limitations.

[1] http://www.roombavac.com
[2] http://www.robocleaner.de
[3] http://www.automower.com
[4] http://www.friendlyrobotics.com
[5] http://www.aibo-europe.com
[6] http://www.necoro.com

Our contribution in this area is a novel hierarchical approach named JDE. This new hierarchical approach, ethologically inspired, is based on the selective activation of schemas, which generates the dynamic hierarchies that govern the robot behavior. In section 2 we describe its major characteristics.

Besides the conceptual fundamentals, every architecture has to be implemented in software. In section 3 we present some of the existing software platforms for the development of robotic software. JDE also provides a software infrastructure to ease the development of robotics software. In section 4 this software architecture is described in detail.

2. **JDE Architecture**

In JDE terminology a "behavior" is considered the close combination of perception and actuacion in the same platform. Both are splited in small units called schemas (Arkin 1989). *Perceptive schemas* build some piece of information concerning the environment or the robot itself, and *actuation schemas* make control decisions considering such information like speeding up the motors or moving some joint. JDE also proposes the schemas can be combined in a dynamic hierarchy to unfold the behavior repertoire of the robot accordingly to current goals and environment situation. Foundations, and further details of JDE can be found in the PhD dissertation (Cañas 2003).

2.1 *Schemas*

JDE follows the Arkin's definition (Arkin 1998) "a *schema* is the basic unit of behavior from which complex actions can be constructed; it consists of the knowledge of how to act or perceive as well as the computational process by which it is enacted".

They are thought as continuous feedback units, similar to teleo-reactive programs in (Nilsson 1997) and are close to control circuitry, which continuously update its outputs from its inputs. In JDE each schema has a time scale and a state associated. They can be switched on and off at will and they accept some modulation parameters. The schemas are thought as iterative processes with a goal, which continuous execution allows them to build and keep updated some stimuli or to develop some behavior.

There are two types of schemas, perceptive and actuation ones. Each *perceptive schema* builds some information piece about the environment or the robot itself, which we'll call a stimulus, and keeps it updated and grounded. It can take as input the value of sensor readings, or even stimuli elaborated by other schemas. It also accepts modulating parameters. Perceptive schemas can be in "slept" or in "active" state.

Each *actuation schema* takes control decisions in order to achieve or maintain its own goals, taking into account the information gathered by different sensors and the associated perceptive schemas. The output of an actuation schema typically orders commands to actuators, and can also activate other schemas, both perceptive and actuation ones. Such new schemas are regarded as its children, and the parent schema often modulates them through mentioned parameters.

Actuation schemas can be in several states: “slept”, “checking”, “ready” and “active”, closely related to how action selection is solved in JDE. Control schemas have preconditions, which implicitly describe in which situations such schema is applicable and may achieve its goals.

The algorithm running inside the schema contains all the task-knowledge about how to perceive relevant items and how to act, whatever technique be used. JDE architecture only imposes the interface: selective activation (on-off) and parameters for modulation, as long as it affects the way all the pieces are assembled together into the system. Continuous (iterative) execution is assumed, actively mapping its inputs into its outputs, regardless how it is achieved: case-based actuation, fuzzy controller, crisp function, finite-state-automata, production sytem, etc.

2.2 *Hierarchy*

All schemas that are awake (those in “checking”, “ready” or “active” states) 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 also claims that such hierarchical organization, in the ethological sense, provides many other advantages for roboticists like bounded complexity for action selection, action-perception coupling and distributed monitoring. All of them without losing the reactivity needed to face dynamic and uncertain environments. We consider that these features make easier the development of versatile and flexible artificial behavior systems.

In JDE there is hierarchy as long as one schema can activate other schemas. An actuation schema may command to actuators directly or may awake a set of new child schemas. These children will execute concurrently and they will achieve in conjunction the father's goal while pursuing their own. Actually, that's why the father awoke such schemas, and not others. A continuous control competition between all the brothers determines whether each child schema will finally get the “active” state or remains silent in “checking” or “ready” state. Only one winner, if any, pass to “active” state and is allowed to send commands to the actuators or spring their own child schemas. The father also activates the perceptive schemas needed to resolve 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.

For instance, in the figure 1 perceptive schemas are squares and actuation ones are circles. Schema 1 activates the perceptive children 3 and 4 to build relevant information, and awakes the actuation schemas 5, 6, and 7. The 6 wins the control competition and then activates 11, 17, 14, and 15. The last one, 15, gains control and really sends commands to the actuators. All winning schemas have been represented in dark circles. All the dashed schemas are in "slept" state and do not consume computing power at all, they are not relevant to the current context.

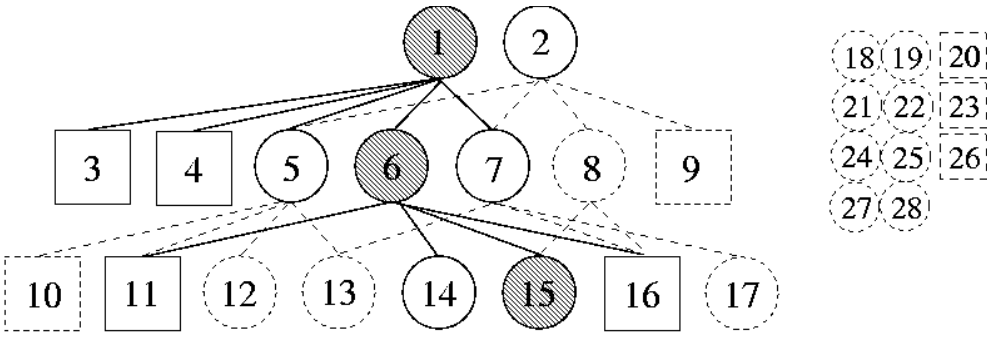

Figure 1. Schema hierarchy in JDE

Once the father has awaken their children it keeps itself executing, continuously checking its own preconditions, monitoring the effects of its current kids, modulating them appropriately and keeping them awake, or maybe changing to other children if they can face better the new situation. At its abstraction level, the father has the entire behavior context to react accordingly to changes in situation. With continuous modulation, the father may adjust the system actuation to small changes in situation, if needed. Changing the children schema lets the hierarchy to reconfigure itself, in response to more significant variations in environment. This way the activation flows from the root adapting the system accordingly to the dynamic situation. Bigger changes may cease the satisfaction of the father's preconditions, and all its children are deactivated, forcing the reconfiguration of the hierarchy at a higher level.

There is no privileged number of levels in JDE, it depends on the complexity of the task at hand and may evolve as new situations are encountered. Higher levels tend to be more abstract and with slower time cycle. The layer concept is weak in JDE, as they are dynamic and not always with the same units inside. The schema is the unit, more than the layer.

2.3 *Action Selection*

JDE decomposes the whole Action Selection Mechanism (ASM) into several simpler action selection contests. At each level of the hierarchy, there is a winner-takes-all competition among all actuation schemas of such level. There is only one winner at a time, and that is the only one allowed to send commands to the actuators or even spring more child schemas.

Before a given schema wins the control, it has to go through several states. First, when the parent awakes it, it passes from "slept" to "checking" state. Second, the schema promotes from "checking" to "ready" when its preconditions match current situation. The preconditions are seen here as ethological triggers (Tinbergen 1951), and preconditions of teleo-reactive programs (Nilsson 1997). They define the set of situations in which such schema is applicable and may achieve its goal, which in JDE is named the *activation region* of such schema. Third, typically, the preconditions of only one schema will hold and it will move from "ready" to "active" state. In case of several (or none) "ready" brothers, the parent is called for choosing one winner, using a fixed priority rule o even a more flexible arbitration.

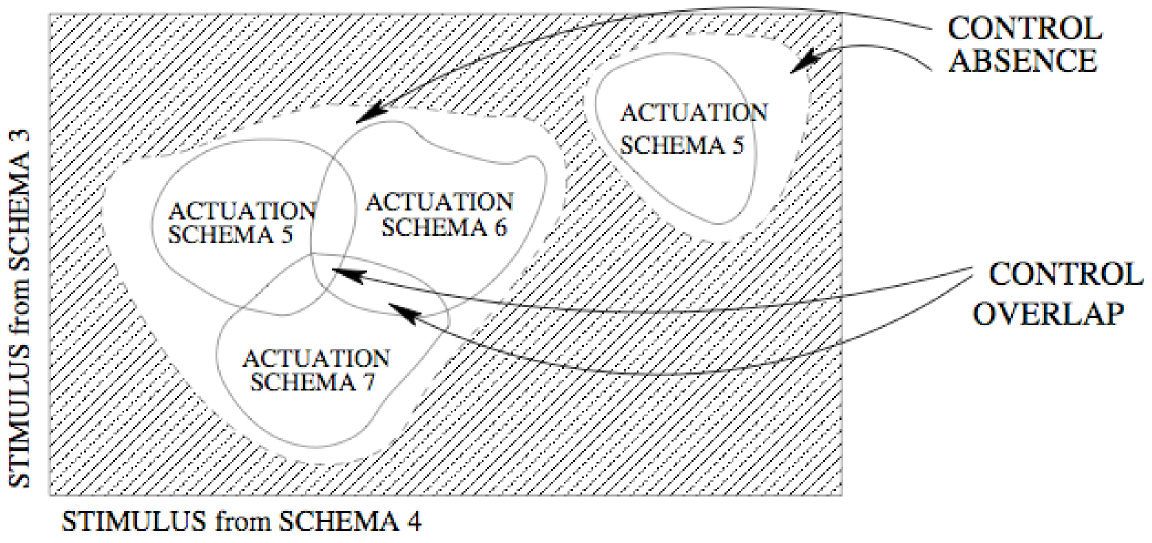

Figure 2 Activation regions in all plausible situations of such context

For instance, in figure 2 the activation regions of schemas 5, 6, and 7 of figure 1 are displayed. They are defined over the space of possible values for stimuli built by perceptive schemas 3 and 4. Impossible combinations are displayed in shadow. The situation is a point in such space and it may fall inside an activation region, such schema will promote to "ready" or outside. This picture is similar to state-space diagrams in (Arkin 2003). The father, schema 1, modulates the activation regions of its children through parameters in order to get more or less disjoint and exhaustive regions. Uncovered or overlap areas explicitly solved in the ad-hoc arbitration when needed.

The JDE architecture is goal oriented and situated. It is goal oriented because the winner lies in the set of schemas already awaken by the father, and no others. It is situated because the environment chooses among them the most suitable to cope with current situation. It is also fast: a new action selection takes place at every child iteration, which allows timely reaction to environment changes.

This approach reduces the complexity, because in a population of *n* schemas there is no need to resolve $O(n^2)$ interactions, but only a number of small competitions where typically 3 or 4 schemas compete at a time for control. In addition, such competitions arise in the context of the parent schema being “active”, so the possible situations are bounded.

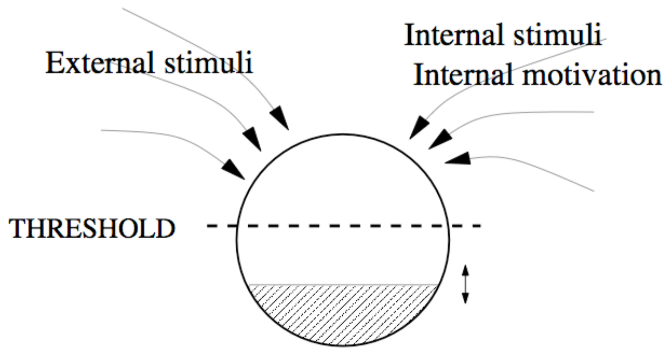

Figure 3. Additive combination of stimuli in JDE

The preconditions can be thought not as the crisp evaluation of the truth of a single condition, but as a thresholded continuous variable which accumulates the contribution of different stimuli to the trigger itself, as can be seen in figure 3. This way it provides room for the additive combination of stimuli and the triggered response observed in the Innate Release Mechanism of animals (Lorenz 1981). Some of such stimuli may be internal ones or motivations (like thirst, fear, hunger, sexual appetite, etc.) with their own dynamics. Several robotics works have explored the effect of such internal variables in action selection (Tyrrell 1993, Arkin 2003).

2.4 *Perception*

Perception has traditionally deserved little interest in the community devoted to research on robotics architectures, but we think that it is an essential part of behavior generation systems, as it provides *all* the information on which an autonomous robot must make its control decisions. Perception is relevant both in action selection, data for triggering conditions (which actuation schema will actually gain control), and in the normal execution of actuation schemas (which actuation decision, or motor value will such schema order to actuators).

The active schemas may change as time passes, so the perception output is a dynamic collection of stimuli more than a general world model. In JDE, perception is task oriented. Actually, it is tightly coupled to action, as the parent schema awakes both the actuation children, and, at the same time, the perceptive schemas which build the relevant information.

Symbols and state are allowed in JDE. The activation of one schema indicates the internal predisposition to such stimulus, as some computing power is devoted to search for and update its internal symbols. Continuous execution of perceptive schemas keeps their symbols grounded just to perceive them when they appear. The stimuli may appear in reality but if no internal process exists attending to it, the robot will miss it. The event has just not existed for the robot.

Hierarchy naturally offers an attention mechanism, and so JDE perception is selective and situated. Only the data relevant to the current context are searched for, as only such perceptive schemas are “active”. No computational resources are spent building stimuli not needed now, because the control schemas that need them and the associated perceptive schemas are “slept”, which is the default state. In contrast, in Brooks system (Brooks 1986) all the automata are always active.

This selective perception in JDE is very convenient because the number of relevant stimuli of the environment, which need to be internalized, grows when the whole system is going to exhibit a wide range of behaviors. Perception costs computing time, which is always limited. This all-time perception of all potential stimuli does not scale to complex and versatile systems, especially if vision is involved.

An attention mechanism is clearly needed. Again, this requirement is not needed if the robot is going to do a single task, but is essential if a full set of behaviors is going to be integrated in a single system.

Sensor fusion and complex stimuli (Arkin98) may be generated in JDE as perceptive schema may activate perceptive children and fuse their outputs into a compound stimulus.

2.5 *Behavior patterns in JDE*

JDE can be used to generate usual behavior patterns found in the robotics literature. For instance, a fixed sequence can be generated with states inside a given schema, and the simple time passing as the triggering condition from one state to the next one.

Taxias (Lorenz 1981) can be easily generated in JDE as the conjunction of two schemas: the perceptive one characterizes the key stimulus, its position, etc.; and the actuation schema decides what to do. For instance, a tracking behavior uses negative feedback, and an obstacle avoidance behavior uses a positive one.

Flexible behavior sequences are identified in animals (Lorenz 1981) and in robotics. In order to implement them trigger condition can be used in JDE. These conditions allow proceeding with the next action in the sequence. For instance, if a sequence of three behaviors want to be implemented, the third schema will really carry on the action that achieves the father's goal. The stimulus that has to be present to promote this schema will be produced by a second schema, that is, the execution of the second schema increases the chance of the stimulus to appear, so it is an appetitive behavior. The same applies to the second with its stimulus and the first.

In this way, JDE hierarchy is not intended as a task-decomposition in primitive steps, which must be executed in sequence, but as a predisposition. There is no a priori strict commitment to a single course of action. Actually the father doesn't fully activate their actuation children, just awakes them. Which one really gains control at every instant depends on the situation, so the sequence adapts its unfolding order to the environmental conditions.

Like reactive planning (Nilsson 1997, Firby 1992), many courses of action are valid and all of them are contained in the behavior specification as a collection of child schemas. This flexibility allows the system to take advantage of environment opportunities and to face little contingencies.

3. **Programming real robots**

All the previously described patterns have to be implemented in a real robot, which means that they have to be implemented in a particular programming language, adapted to different robotic platforms, etc. This is the other sense of the word "architecture" when applied to robotics, the software infrastructure. Different research groups, manufacturers, etc. have developed different software architectures, in this section we will review their main characteristics.

3.1 *Implementing a robotic architecture*

The ultimate goal of robotic architectures is to be able to make working robots. This requires making choices about which software components use, and how to configure them. This decision is conditioned by several requirements:

- Mobile robots works in real world domains where real-time is required, may be not hard real-time, but al least soft real-time. Besides, there are several types of sensors and actuators, as well as interfaces that the programmer must know.
- Typical robotics applications must manage various sources of information (sensors) and pursuing various goals at the same time,
- User interface in robotics applications is another major concern in robotics, not only from the human computer interaction point of view, in the interaction of robots with users; but for the debugging of the pa
- Robotic software is increasingly distributed, as Woo et al. (Woo 2003) have pointed out. It is usual that robotic applications have to communicate with other processes, in the same machine or in other computers.
- There are few knowledge about how generate and organize robotic behavior. How to organize it is still a research issue and there is no general guides about.
- There are no open standards for robotic software developent (Utz 2002, Montemerlo 2003, Cote 2004). In other computer science fields there libraries that

programmer can use. However, in robotics there are few reusable software, most programs have to be built from scratch. This is due to the heterogeneity of robotic hardware and to the immaturity of the robotic industry.

The way robots are programmed has been changing. Historically the robots were unique, they were not produced in series, and programs were built directly using device drivers; operating systems were minimal. Application programs read sensor data directly from sensors using the drivers, and wrote the commands directly on the motors using the library provided by the robot manufacturer.

The reduction in the number of manufacturers, and their consolidation, as well as the work of many research groups, have made possible the appearance of development platforms that simplify the development of software, as well as different tools.

3.2 *Development tools*

Creating software for mobile robots is in essence as any other application development. Programmer has to write the program in a particular programming language, compile it, link it with the libraries, and finally execute it in the on-board computers. In many cases, all this process is performed in a PC, using cross compilers.

There are some specific programming languages for robotics, as for instance TDL (Task Description Language) (Simmons 1998) or RAP (Reactive Action Packages) (Firby 1994). However, they have not been a success, and usual programming languages (C, C++, Java, etc.) are the dominant ones.

Another set of useful tools is the simulators. Early simulators were little realistic; only simulate flat worlds populated by static obstacles. Nowadays simulators have been greatly improved. There are 3-D simulators like Gazebo[7], or able to simulate vision (Lozano 2001), or to include many robots in the same world, as Player/Stage[7]. Besides, many robotic manufacturers include a simulator for their robots, as for instance, EyeSim for the EyeBot, Webots for Khepera and Koala, etc. However, we think that general simulators are better option. Among the open source simulators not affiliated to any manufacturer, we think that SRISim[8] and Stage[7] are the most relevant.

3.3 *Robotic software development platforms*

In many areas of software development “middleware“ has appeared as a way to simplify software development. Middleware provides predefined data structures, communication

[7] http://playerstage.sourceforge.net
[8] http://www.ai.sri.com/~konolige/saphira

protocols, sincronization mechanisms, etc. In the robotics software different platforms have appeared (Utz 2002).

Hattig et al. (Horswill 2003) have pointed out that uniform access to hardware is the first step towards software reuse in the robotics industry. This characteristic can be found in many platforms, but each one implements it in its own way. For example, in ARIA the API for accessing sensors and actuators has been made up by a set of methods, while in Player/Stage or in JDE has been implemented as a protocol between the applications and servers.

JDE software infrastructure will be described in next chapter. In order to compare it with other alternatives available, we will analyze in this section some software development platforms available nowadays for the robotics programmers. We will analyze both commercial platforms, such as ARIA from ActivMedia, OPEN-R[9] from Sony; as well as, open source efforts as Player/Stage, MIRO, Orocos (Bruyninckx 2001), or CARMEN.

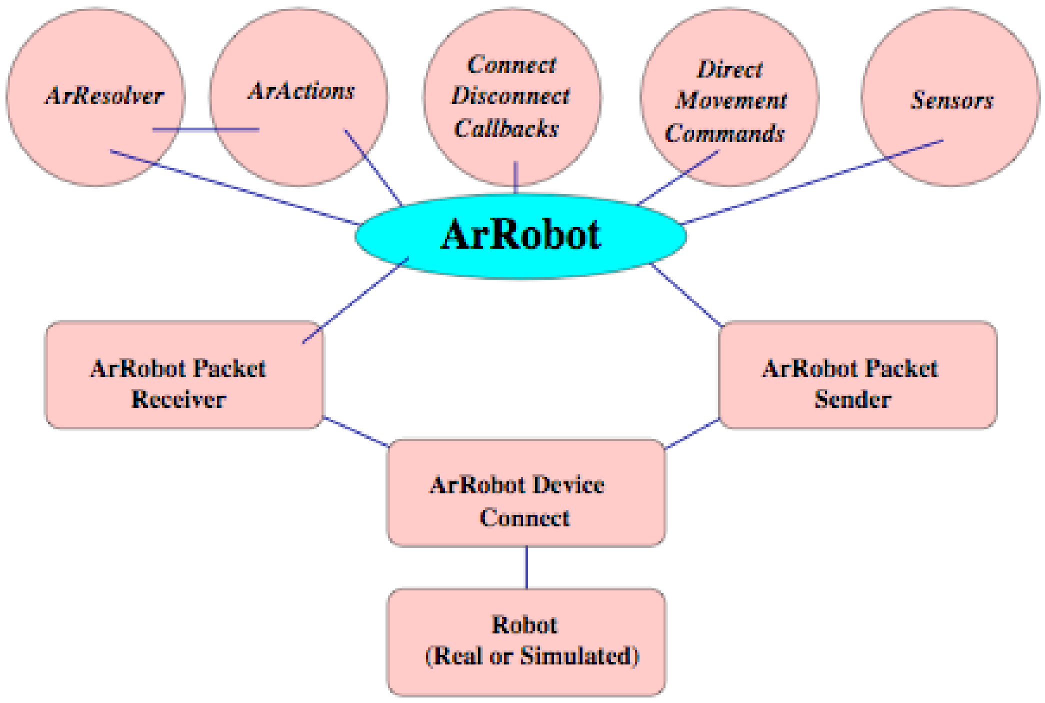

Fig 4. Architecture of the ARIA API

<u>ARIA</u>

ActivMedia has become the leader in the robotics research manufacturer business after the shutdown of Nomadic and RWI. ARIA (Activmedia Robotics Interface for

[9] http://www.openr.aibo.com

Applications) (ActivMedia 2002) is the development environment distributed by ActivMedia with its robots (Pioneer, Peoplebot, Amigobot, etc.).

Underlying the API offered by ARIA, there is a client-server design. ActivMedia robots are managed by a micro-controller that implements the server. The application written using ARIA connects to this server using a set of predefined messages (protocol) through a serial port.

ARIA offers an object-oriented environment (applications have to be written in C++). It offers multitasking support and communications facilities. In this case, Aria objects are not distributed; they are placed in the computer that is physically connected to the robot. Anyway, ARIA offers the `ArNetworking` class to manage communications.

Hardware access in Aria is made through a collection of classes as shown in figure 4. Main class is `ArRobot` where methods as `ArRobot::setVel` commands a translational velocity and `ArRobot::setRotVel` the rotational one. `Packet Receiver` and `Packet Sender`, manage the sending and reception of packages through the serial port. Other classes as `ArSonarDevice` or `ArSick` contain methods for accessing those devices.

ARIA is offered for Linux and Win32 and it is distributed under GPL license. It also includes some basic behaviors as obstacle avoidance, but more complex ones are offered as proprietary separate libraries: MAPPER for managing maps, ARNL (Robot Navigation and Localization) for navigation algorithms, ACTS (Color-Tracking Software) to identify objects, etc.

<u>MIRO</u>

Miro[10] is a distributed object-oriented platform developed at Ulm University and distributed under GPL license. It uses CORBA as the underlying standard for object distribution, to be exact TAO.

Miro is composed by three levels: devices, services, and classes. The first one, “Miro Device Layer“, is the one devoted to access hardware devices; it offers different objects for every device. That is, sensors and actuators are accessed through methods of different objects, depending on the hardware. For example, `RangeSensor` defines an interface for sensors such as sonar, infrared, laser, etc. `DifferentialMotion` contains the methods to move the robot, etc. Second layer, “Miro Services Layer”, provides descriptions of the sensors and actuators as CORBA IDLs, so they can be accessed remotely by objects running in other computers, independently of the operating system.

[10] http://smart.informatk.uni-ulm.de/MIRO

Third level, “Miro Class Framework” groups the tools for visualization and logs management, as well as general use modules, as map building, path planning, or behavior generation, etc.

An application developed using Miro is a collection of local and remote objects. Each one runs in its own computer and all of them communicate using the infrastructure of the platform. Objects can be written in any language that supports the CORBA standard. Miro itself has been written in C++.

There are other platforms using distributed objects, and CORBA. One of the better known is Orocos (Open RObot COntrol Software), the open-source project funded by the European Union.

OPEN-R

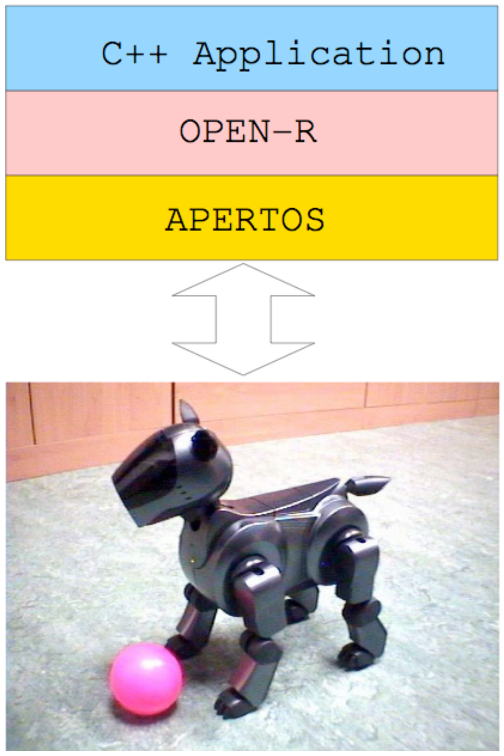

Fig 5. Software architecture for programming Sony Aibo robot

Open-R is the API released by Sony for programming the Aibo robot. Aibo is a robotic pet, whose main sensor is a camera situated in the head, and whose actuators are the four legs, the tail and the neck. It is an autonomous robot based in a 64 bit RISC processor, able to communicate using an integrated WiFi card.

The operating system used in this robot is Aperios (previously know as Apertos, and Muse). It is a real-time object oriented operating system. It is proprietary software, and Sony has just release the Open-R API.

Open-R offers just a C++ interface, and applications can consist in one or more Open-R objects (Martín 2004). Application objects can use a set of basic objects, and that can send and receive messages among them. Each object runs in its own thread and control flow is event-based.

Among the basic objects, `OvirtualRobotComm` lets applications access to images and joints through services as `Effectors`, that moves the dog joints, or switches the LEDs on and off; `Sensor` that reads the position of the joits; or `OfbkImageSensor` to access the recorded images. `ANT` object manages the TCP/IP stack to let applications communicate outside the robot, offering the `Send` and `Receive` services.

Development environment using Open-R is a PC using a cross compiler for the Aibo RISC. Programs developed are written in a memory stick that it is inserted in the robotic dog.

Player/Stage/Gazebo

Another open-source platform is Player/Stage/Gazebo (PSG). Initially developed at Southern California (Gerkey 2003), nowadays is an open-source project. Nowadays it support different robots (Aibo, Pioneer, Segway, etc.), and the simulator included makes him a complete platform.

In PSG sensors and actuators are managed as files (Vaughan 2003) as in Unix operating system. Five basic operations can be performed on those files: open, close, read, write, and configure. Every type of device is defined in PSG by an interface; for example, the ultrasonic sensors of two different robots will be an instance of the same interface.

PSG is based on client-server design. Any application has to establish a dialogue using TCP/IP with the server Player. This idea let applications be really independent (i.e. they can be written in any programming language) and imposes minimal requirements on their architecture.

PSG is oriented to offer an abstract interface of the hardware of the robots, not to offer common blocks. However, these blocks can be added defining new messages for the protocol. For instance, probabilistic localization has been added as another interface `localization`, that provides multiple hypotheses. This new interface overwrites the traditional `position` based just in odometry.

There are other well-know platforms, as for instance Mobility, the one developed by RWI (B21, B14 are classical robots); Evolution Robotics sells its ERSP; CARMEN is offered by Carnegie Mellon university; etc. We have described the previous four just to show the reader the major alternatives faced when developing JDE software infrastructure described in next section.

4. JDE Software infrastructure

Once we have seen existing platforms, we will describe in this section the characteristics of JDE software infrastructure. Current JDE implementation consists of the infrastructure software (JDE servers), a collection of behavior specific schemas (JDE basic schemas), and auxiliary tools and libraries.

A typical robot control program in JDE programming environment is made up of a collection of several concurrent asynchronous threads, corresponding to the JDE schemas. Over basic schemas there may be perceptive and actuation schemas. Perceptive schemas make some data processing to provide information about the world or the robot. Actuation schemas make decisions in order to reach or maintain some goal. They order motor commands or activate new schemas, because the schemas can be combined forming hierarchies. The underlying theoretical foundations have been described in section 2.

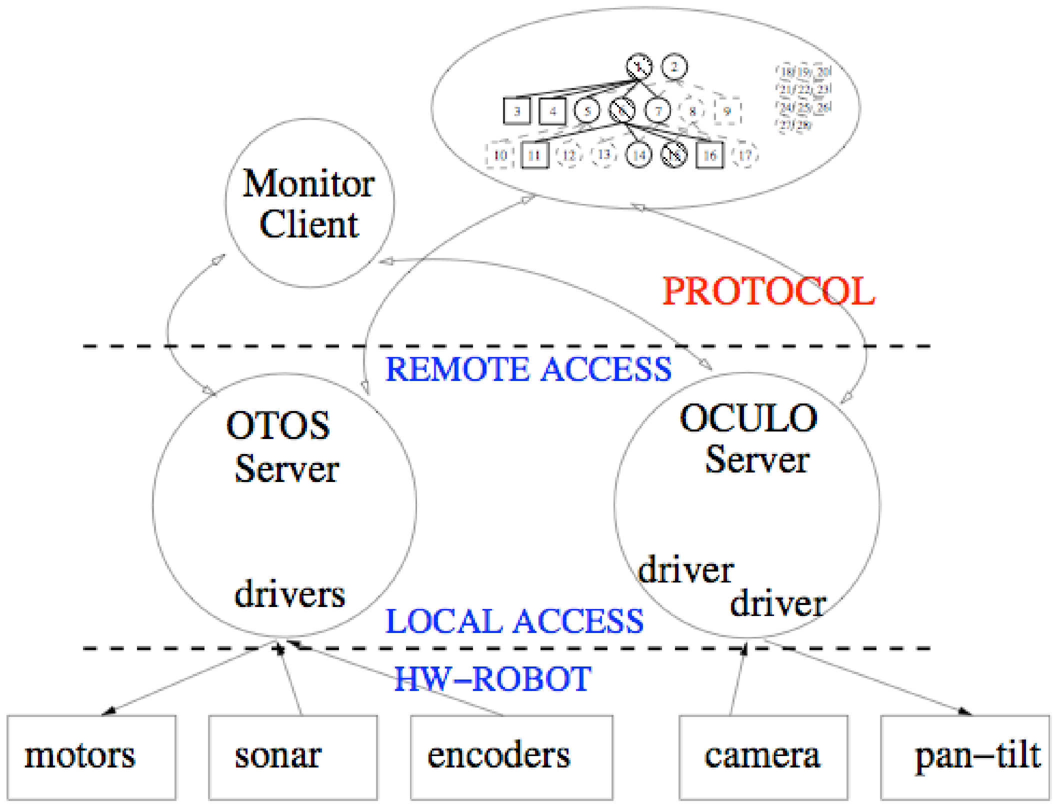

Figure 6. JDE software architecture. OTOS and OCULO servers

Figure 6 summarizes this architecture. Two JDE servers (OTOS and OCULO) are in the middle, represented as two circles. These servers access the robot sensors and actuators (in the low part of the figure 4) using specialized drivers, and provide the data to clients (small circles in the top of the figure 4) through a protocol named *JDE Protocol*. Developers can build their applications using directly these servers; they just need to implement the protocol to communicate with the servers.

JDE also provides some auxiliary clients, as for instance a monitor of the sensors (circle at the top left of the figure). Others clients implement the schemas in which JDE is based on.

In order to facilitate the implementation of JDE schemas, the distribution provides some basic schemas that interact with the servers, offering sensor data as shared variables, which is much more simple than implementing the protocol. The release also includes the skeleton of the schemas to ease its use.

In summary, JDE software implementation offers two options to develop programs, directly using JDE servers, or using the service schemas. The second one is the recommended one both because is easier, and because is closer to the conceptual ideas behind JDE. Figure 7 shows these two options.

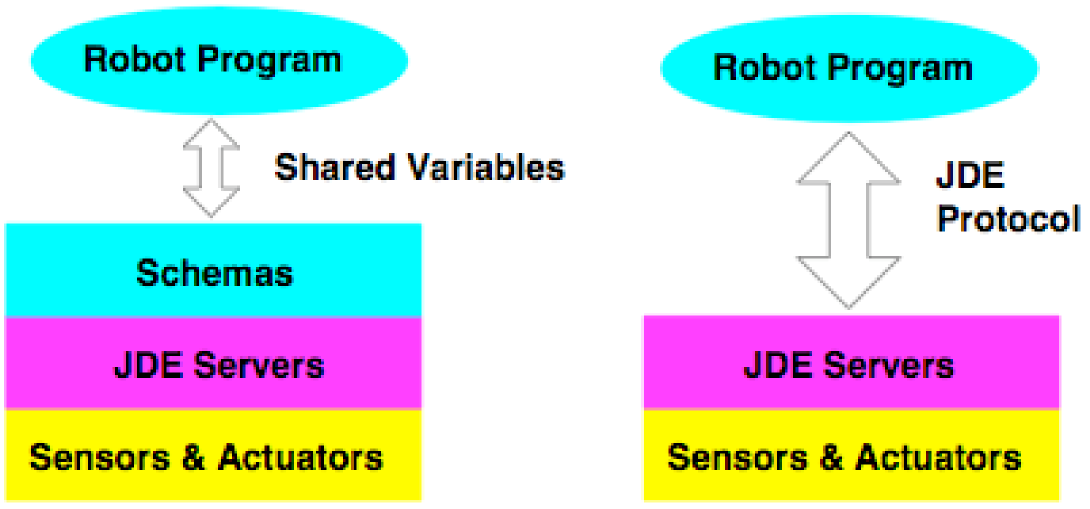

Figure 7. Programming options using JDE

4.1 *JDE servers*

Robot application gets sensor readings and order actuator commands sending network messages to the JDE servers. There are subscription messages for short sensor readings like odometer, sonar or laser. Those data are continuously sent from the server to the subscribed clients. Large readings, like camera images, are sent only on demand. There are also actuation messages from clients to servers. All these messages make up a protocol that settles a hardware abstraction layer and provides language and operating system independence to the robot applications.

OTOS server manages proximity sensors, such as sonar, laser, infrared, and tactile sensors. It also manages propioceptive sensors such as odometers, and battery power. It also sends messages to the robot actuators.

The protocol to communicate with OTOS servers is mainly based on direct subscription, that is, once the client has connected it can subscribe to a particular sensor and the sender periodically sends the new data. Each client can subscribe to those sensor it is interested on.

OTOS server has been implemented using five different threads, as shown in figure 8. One is devoted to attend new clients; a second one serves already connected clients. The other ones are devoted to listen measurements for the different sensors.

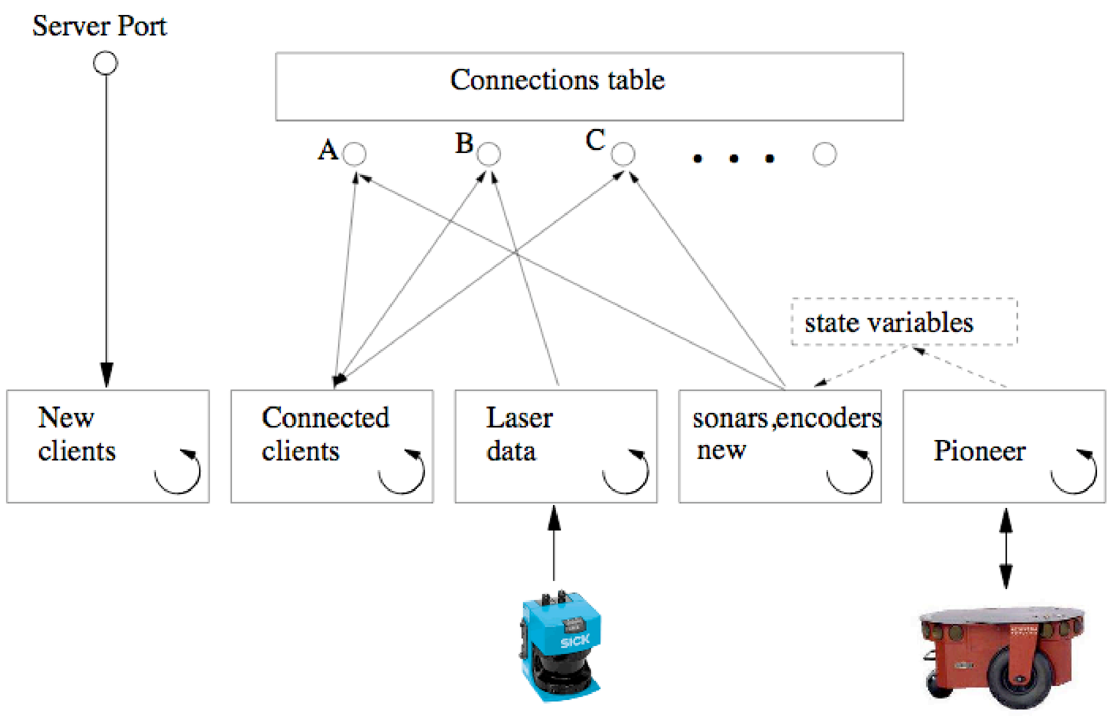

Figure 8. OTOS implementation using 5 different threads

OCULO server manages the functions related to the camera and the pan-tilt unit. It lets clients send messages to point the camera, and it sends them the images flow. In this case the images are send under demand, that is, each time the client needs an image, it has to ask for it.

OCULO has been implemented using 4 threads, as shown in figure 9. As in the OTOS case, two threads are devoted to attend new clients and to interact with already connected ones. The other two manage the interaction with the camera and the pan-tilt unit. Communication with the cameras has been implemented using the video4linux standard.

Developing an application over JDE servers is the more flexible way of using JDE, however it is the hardest way of using it. The alternative is to use the service schemas provided in the distribution.

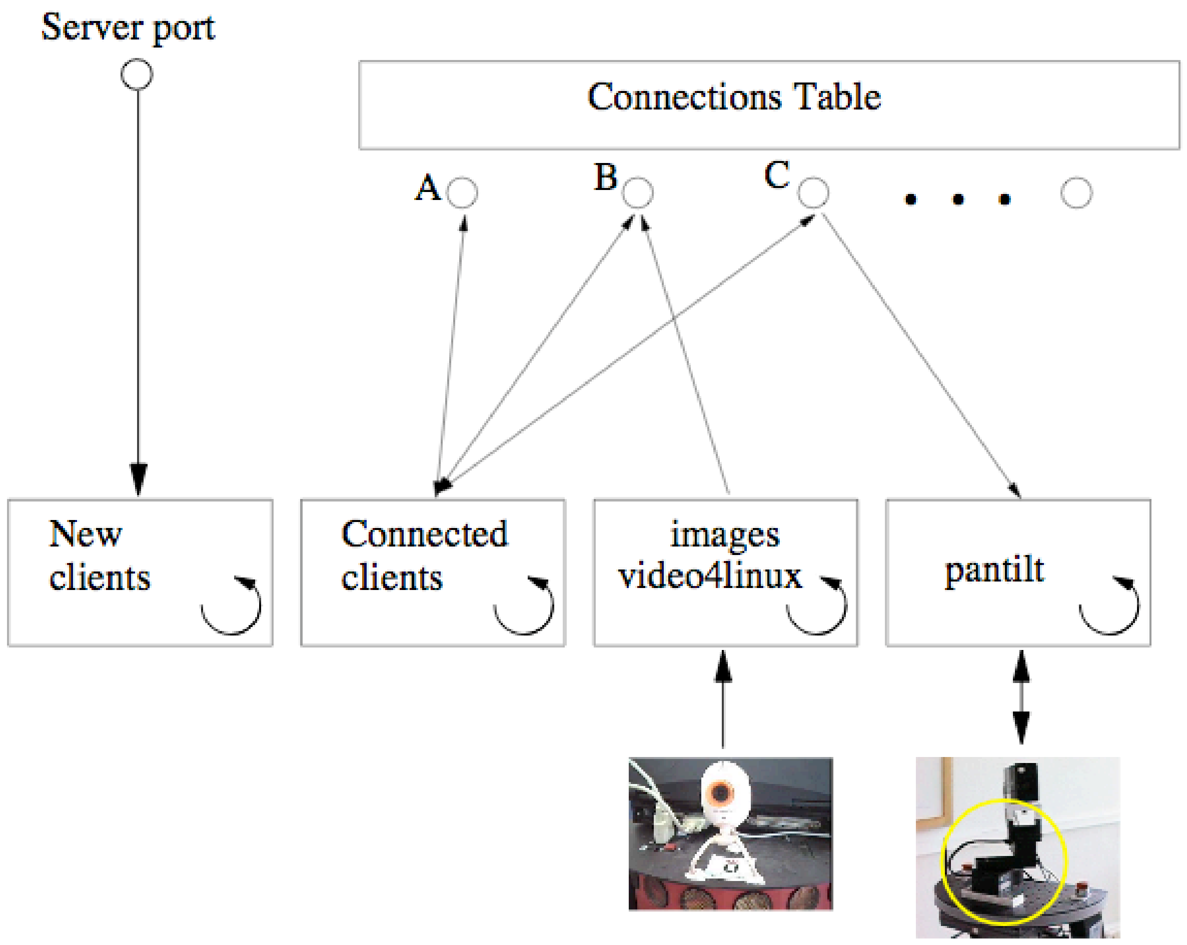

Figure 9. OCULO implementation using 4 different threads

4.2 *Service schemas*

From the theoretical point of view of JDE, everything should be implemented as a schema. It is not just a fundamentalism issue; it is a practical one. It is easier to implement complex behavior using schemas than as traditional programs. JDE distribution includes schema skeletons to make easy the construction of new schemas, and it also includes completely implemented schemas, named “service schemas” to facilitate the interaction with JDE servers.

Getting sensor measurements in JDE using schemas is as simple as reading a local variable, and ordering motor commands as easy as writing an actuator variable.

A set of basic schemas updates those sensor variables with fresh readings and implements such actuator variables. They connect to real sensors and actuators, directly on local devices or through remote socket servers. Pioneer and B21 robots, SRIsim and Stage simulators, video4linux cameras, firewire cameras, Directed Perception pan-tilt units, and SICK laser scanners are fully supported in current version of JDE.

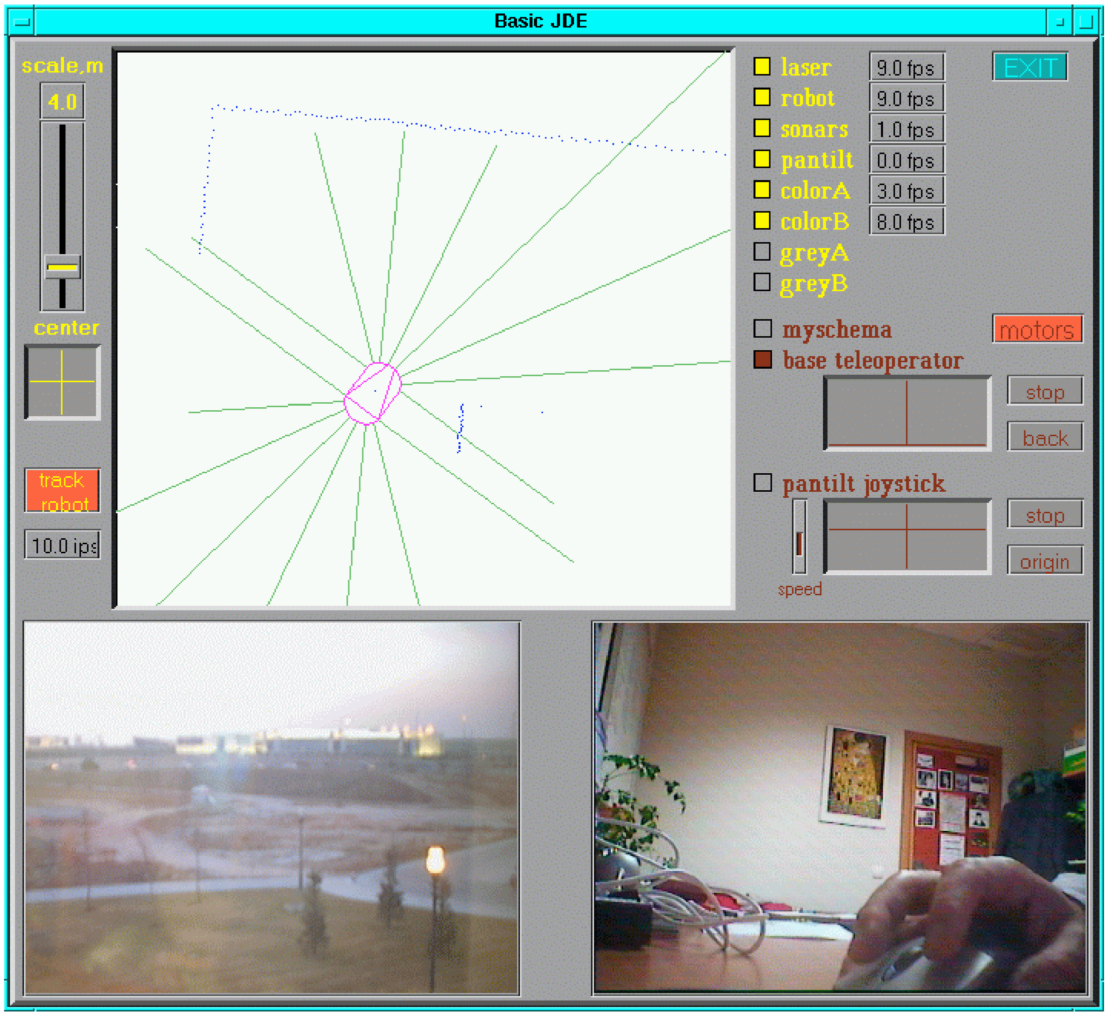

Figure 10. JDE GUI connected to a Pioneer and two different cameras (off-board robot)

Five *service schemas* are included in the JDE distribution. “Pan-tilt-motors” and “base-motors” periodically translate the actuation variables (like translation *v* and rotation *w* speeds) into actuation messages to the servers. “Oculo-sensations” and “otos-sensations” receive network messages from the servers and update the variables that store corresponding sensor readings. Finally, “gui-xforms” displays sensor readings, internal states and allows the explicit activation from the GUI (Figure 10). It is useful for debugging and monitoring.

Applications in JDE can be written, either using the JDE servers, or using the schemas. In both cases, some auxiliary tools are available.

4.3 *Auxiliary tools*

A very useful client named *Record* has also been included in JDE release. This client stores all sensorial data in a file, as shown in the left part of the figure 8. This client is used together with the *Replay* client. This client replaces the OTOS and OCULO server reproducing off-line the data stored by *Record* client. This couple is very useful for instance when we want to compare different algortihms, or to debug a schema. The *Replay* server can also reproduce data at different speeds (1x, 2x, etc.). Both clients are based on time stamps with a resolution of microseconds.

Another useful client interacting with OTOS y OCULO server is the *Monitor*. This client continiously shows the values received from the sensors for a human operator. This client also provides a visual joystick to teleoperate the robot.

Various clients of this type can be used simultaneously. In the same way, it can be used concurrently with other clients that were implementing the behavior of the robot. This monitor can be switched on and off according to the needs of the debugging, saving computer power or resources if needed.

Auxiliary libraries have been also written for grid manipulations (*gridslib*) and fuzzy control (*Fuzzylib*), to make easier the behavior generation. In the same way, various tools have been developed to ease the development of the behaviors.

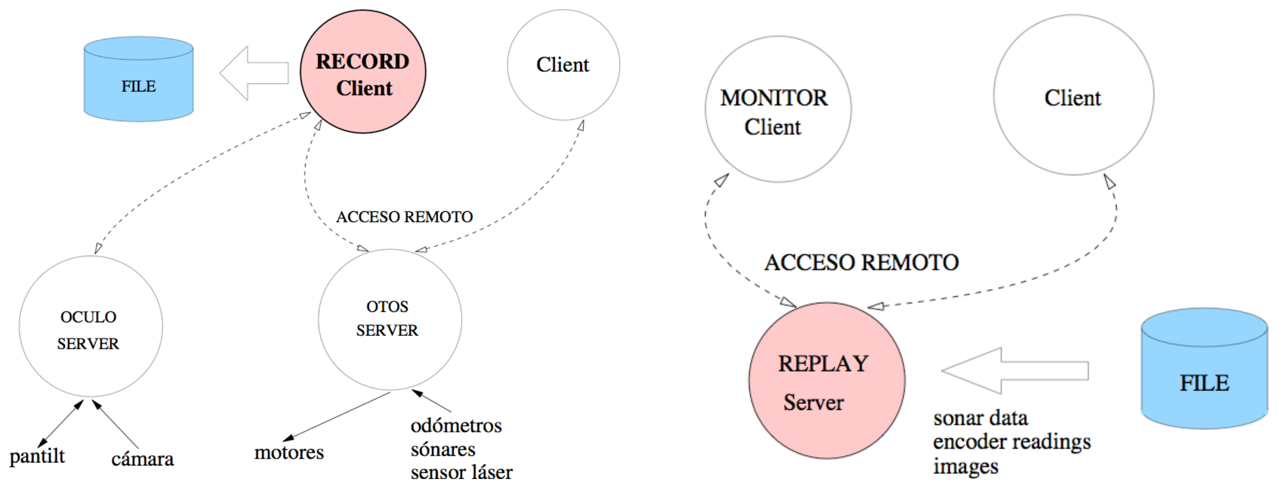

Figure 11. Record client (left) and Replay server (right) for off-line work.

For instance, *ReplayServer* provides sensor data (sonar, laser, encoders, images, etc.) previously recorded through the same protocol, which is very convenient for off-line work. *HSItuner* in the same way can be used to tune the values of color filter in HSI space.

4.4 *Implementing JDE schemas*

JDE schemas have been implemented as independent kernel threads (we have used Pthreads library), so they run concurrently. At the initialization time, all threads are started but immediately stopped to “slept” state, waiting for someone to awake them. The parent thread can resume the execution of a child thread or require it to stop, providing room for the selective activation.

All schemas assume iterative execution. The frequency of the iterations determines the schema time scale, and it is enforced with intentional delays between consecutive iterations. So, the computing power demand comes in periodic bursts.

The iterative execution is a discrete approximation of the continuous feedback of the theoretical model, and a good one if period is fast enough compared to the events in the surroundings (Nilsson 1994).

The schemas communicate each others through selective activation, stimuli, and modulation parameters. The last two are implemented through shared variables. As long as several schemas may access to such data concurrently, locks are used to protect them and avoid race conditions.

Each actuation schema includes a *preconditions function*, which at every iteration checks whether such schema should promote to “ready” state or not. In addition, the parent provides each of its children a *list of brothers* and a pointer to the *arbitration function*.

Every kid checks its own preconditions and its brothers' state to detect any control collision or absence. If its own preconditions are not satisfied, it searches whether there is another “active” brother. If not, then it calls the arbitration function to resolve the control absence. If its preconditions hold, then it searches whether there is another “ready” or “active” brother, and in such a case calls the arbitration function to resolve that control collision. If it is the only “ready” or “active” child, it simply executes its specific code for that iteration.

The arbitration function is already implemented in the father's code and the caller is indicated whether it must promote to “active” or remain in “ready” state. This way the arbitration is configured by the father, but it takes place when a child needs it.

5. Conclusions and further research

An architecture for behaviour generation named JDE has been presented. Its distinguish characteristics are its hierarchical nature and the use of schemas as the basic building blocks. Its hierarchy model is taken from ethology and intended as a predisposition of the system to perceive certain stimuli and to response in a certain way to them.

JDE explicitly introduces perception into the architectural model, closely tied to action, offering also a selective attention mechanism. The action selection also takes benefit from the hierarchical organization as it is decomposed into several small competitions. This bounds its complexity and allows flexible coordination of the schemas.

One of the weak points in this architecture is that it lacks of high level reasoning that allows predictions about the future. There is no explicit symbolic deliberation in the classical sense. So, all relevant data for the behaviour must be anticipated by the designer, in particular the corresponding stimuli and perceptive schemas have to be allocated in advance. All relevant situations must be anticipated by the designer and so the corresponding actuation schemas to deal with them allocated in advance. In a similar way, perception in JDE can be seen as propositional calculus, but no as first order predicate logic.

We are currently working in the introduction of active perception into JDE. The proposal consists in a perceptive schema having actuation children whose goal is to help in the perception process of its father. More schemas to increase the behaviors pool and a new JDE software implementation are also under development.

We are committed to general platforms, we think that the only that claims made by robotic researchers can be checked is if it can be reproduced. In this way JDE has support for general protocols (for instance video4linux) and it has been developed first on B21 robots and later for ActivMedia family of robots (Pioneer, etc.) using Aria. In the same way, we think that the only way to really check the developments is by releasing programs as *libre*[11] software.

All the code developed to implement JDE has been developed in C language for GNU/Linux machines, and is released under GPL. There are GNU/Linux Debian packages available for i386 architectures in http://gsyc.escet.urjc.es/robotica/software. It can be directly installed in a Debian based Linux using `apt-get` adding the following lines to the `sources.list` file:

```
deb http://gscy010.dat.escet.urjc.es/debian dist main
deb-src http://gscy010.dat.escet.urjc.es/debian dist main
```

[11] *Libre* is the word in Spanish for free as in “free speech“, to avoid misunderstanding in English with “free beer“

6. References

ActivMedia (2002). ARIA Reference Manual (v. 1.1.10).

J.S. Albus (1999). The engineering of mind. *Information Sciences*, Vol. 117, no. 1-2, pp. 1-18.

R.C. Arkin, M. Fujita, T. Takagi and R. Hasegawa (2003), An ethological and emotional basis human-robot interaction, *Robotics and Autonomous Systems*, vol 42, pp. 191-201.

R.C. Arkin (1989), Motor schema-based mobile robot navigation, *International Journal of Robotics Research*, vol 8, no 4, pp. 92-112.

R. Arkin (1998), *Behavior-based robotics*, MIT Press, 1998.

S. Behnke and R. Rojas (2001), A hierarchy of reactive behaviors handles complexity, *Balancing reactivity and social deliberation in multi-agent systems*, LNCS 2103, pp. 125-136

R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D.P. Miller and M.G. Slack (1997), Experiences with an architecture for intelligent reactive agents, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 237-256.

R.A. Brooks (1986), A robust layered control system for a mobile robot, *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, March, pp. 14-23.

H. Bruynincks (2001), Open Robot Control Software: the OROCOS project, *Proceedings of the 2001 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS-2001)*, vol. 3, pp. 2523-2528.

J.J. Bryson and L.A. Stein (2001), Modularity and design in reactive intelligence", in *Proceedings of the 17th Int. Joint Conf. on Artificial Intelligence*, pp. 1115-1120.

J.M Cañas, and V. Matellán (2002). "Dynamic schema hierarchies for an autonomous robot". Advances in Artificial Intelligence (IBERAMIA-02). Lecture Notes in Artificial Intelligence (LNAI-2527), pp. 903-912.

J.M. Cañas(2003), *Jerarquía dinámica de esquemas para la generación de comportamiento artificial*, PhD dissertation, Universidad Politécnica de Madrid.

C. Cote, D. Létourneau, F. Michaud, J. M. Valin, Y. Brosseau, C. Raïevsky, M. Lemay, and V. Tran (2004). "Code reusability tools for programming mobile robots". *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-04).*

R. J. Firby (1992), Building symbolic primitives with continuous control routines", in *Proceedings of the Int. Conf. on AI Planning Systems*, AIPS'92, pp. 62-69.

R. J. Firby (1994), "Task networks for controlling continuous processes". Proceedings of the 2nd International Conference on AI Planning Systems, pp. 49-54.

B. P. Gerkey, R. Vaughan, and A. Howard (2003). "The Player/Stage project: tools for multi-robot and distributed sensor systems". *Proceedings of the 11th International Conference on Advanced Robotics (ICAR-03),* pp. 317-323.

M. Hattig, I. Horswill , and J. Butler (2003), "Roadmap for mobile robot specifications", *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-2003)*, vol. 3, pp. 2410-2414.

I. Horswill (1997), Visual architecture and cognitive architecture, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2, 1997, pp. 277-292.

K. Konolige and K.L. Myers (1998), The Saphira architecture for autonomous mobile robots, *Artificial Intelligence and Mobile Robots: case studies of successful robot systems*, MIT Press, pp. 211-242.

K. Lorenz (1981), *Fundations of Ethology*, Springer Verlag, New York, Wien; 1981.

M.A. Lozano, I. Iborra, D. Gallardo (2001). "Entorno Java para la simulación de robots móviles". Actas de la IX Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA-01. pp.1249-1258.

F. Martín, R. González, J.M. Cañas, and V. Matellán (2004), "Programming model based on concurrent objects for the Aibo robot", *Actas de las XII Jornadas de Concurrencia y Sistemas Distribuidos*, pp. 367-379.

M. Montemerlo, N. Roy and S. Thrun. "Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation CARMEN Toolkit". *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*. Vol. 3, pp. 2436-2441.

M.N. Nicolescu and M.J. Mataric (2002), ``A hierarchical architecture for behavior-based robots'', in *Proceedings of the 1st Int. Joint Conf. on Autonomous Agent*s and Multiagents Systems, AMAAS'02, pp. 227-233.

N. J. Nilsson (1984), Shakey the robot, SRI Technical Report 323, April 1984.

N. J. Nilsson (1997), Visual architecture and cognitive architecture, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 277-292.

A. Saffiotti and Z. Wasik (2003), Using hierarchical fuzzy behaviors in the RoboCup domain, *Autonomous Robotic Systems*, Springer, 2003, pp. 235-262.

R.G. Simmons (1994), Structured control for autonomous robots, *IEEE Journal of Robotics and Automation*, vol. 10, no. 1, February, pp. 34-43.

R.G. Simmons, D. Apfelbaum (1998). "A Task Description Language for robot control". Proceedings of the 1998 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS-98). Vol. 3. pp.1931-1937.

N. Tinbergen (1951), *The study of instinct*, Oxford University Press, London; 1951.

T. Tyrrell (1994), The use of hierarchies for action selection, *Journal of Adaptive Behavior*, vol. 2, no. 4, pp. 307-348.

H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar (2002), "MIRO – Middleware for mobile robot applications, *IEEE Transactions on Robotics and Automation*. Special Issue on Object-Oriented Distributed Control Architectures. Vol. 18, no. 4, pp. 493-497.

R. T. Vaughan, B. P. Gerkey, and A. Howard (2003). "On device abstractions for portable, reusable robot code". Proceedings of the 2003 IEEE/RJS International Conference on Intelligent Robots and Systems (IROS-2003). Vol. 3 pp. 2121-2127.

E. Woo, B. A. MacDonald, and F. Trépanier. "Distributed mobile robot application infrastructure". *Procceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03).* Las Vegas (USA), Vol 3, pp. 2410-2414.