A Simple, Efficient, and Scalable Behavior-based Architecture for Robotic Applications

Francisco Martín[1], Carlos E. Aguero[2], José M. Cañas[1]

[1] Robotics group - Rey Juan Carlos University
C/ Tulipán s/n, 28933, Móstoles (Madrid, Spain),
francisco.rico@urjc.es
josemaria.plaza@urjc.es
[2] Open Source Robotic Foundation, 419 N Shoreline Blvd Mountain View, CA 94043, USA,
caguero@osrfoundation.org

Abstract. In the robotics field, behavior-based architectures are software systems that define how complex robot behaviors are decomposed into single units, how they access sensors and motors, and the mechanisms for communication, monitoring, and setup. This paper describes the main ideas of a simple, efficient, and scalable software architecture for robotic applications. Using a convenient design of the basic building blocks and their interaction, developers can face complex applications without any limitations. This architecture has proven to be convenient for different applications like robot soccer and therapy for Alzheimer patients.

Keywords: Behavior architectures, autonomous humanoid robots, real-time processing

1 Introduction

In recent years, many developments have been made in mobile robotics. Robots are equipped with more complex actuators (even for diving or flying), richer sensors (as cameras RGB-D), batteries that increment robot autonomy, and more powerful processors that let the robot process huge data onboard. Moreover, many commercial robotic platforms are available now at a low cost. This enables us to focus on the development of software without addressing the development of a complete robot from scratch. Despite this advantage, the development of software for robots is a complex task.

Software for robots defines robot operation. Roodney Brooks [1] presented the basis of behavior-based robotics. This paradigm describes how the complex behaviors are built up to decompose into simpler behavior modules, which can be organized as modules. How these simple modules work, how they organize, and how they interact among them are open questions that continue to receive attention from the robotics community. The active ROS [11] community and its impact on the industry denotes this importance during the last few years.

We have developed a behavior-based architecture, which includes simple and novel ideas for effective development of robotic applications. We have defined a basic building block and an effective and simple mechanism for making them cooperate. Each behavior is implemented as a basic building block that decomposes its complexity, explicitly executing another building block. We will describe mechanisms to warranty that the information produced by perceptual building blocks does not expire before being used, avoiding race conditions using a single thread scheme, and recovering from high-load situation using a graceful degradation approach. These concepts are related to the scheduling of real-time systems, which is a very convenient approach when developing software for a robot that interacts with the real world.

An important part of this paper is the focus on a complete, deep technical description. We think that this approach is valid for making this work really useful, letting robotic software developers include some of these ideas in their software architectures.

We develop complex mechanisms using the following building blocks: visual attention, perception, or debugging mechanisms. These capabilities can be considered basic to the software for robots, but they are crucial during the development of high-level behaviors. An effective, clean, and scalable design lets developers face complex behaviors without any limitations. Monitoring and debugging tools are essential when developing robotic software, in which the usual techniques from computers (messages to stdout or debuggers) are not effective on robots.

An important aspect of this work is that all these ideas have been implemented in a real robot, and used to implement real applications. Because of the experience and feedback along these years, some ideas have been incorporated, redefined, or discarded. Nowadays, we can affirm that this architecture is mature and effective. Using this architecture, we have developed a complete software system for a team of robots that participates in the Standard Platform League of the RoboCup. This competition presents a dynamic scenario, a soccer match, where we have implemented fast response behaviors [2], self-localization [3], navigation, coordination, attention, and perception [4] algorithms using this architecture. We also used this architecture to develop a complete therapy system [5] for Alzheimer patients by using robots as a cognitive activator actor.

In section 2 we will present the existing works on software architectures. Section 3 describes the core ideas of the architecture presented in this work. In section 4 we extend these general principles with some optional communication or debugging mechanism. Section 5 presents mechanisms, such as perception and visual attention, particular to robotic humanoids equipped with cameras. A brief description of two successful uses of this architecture is presented in section 6.

2 Related work

Robotic frameworks can be grouped into two main paradigms: those tightly coupled with a cognitive model in their designs and those designed just from

pure engineering criteria. The former *forces* the user to follow a set of rules to program certain robotic behaviors, while the latter are just a collection of tools that can flexibly be put together in several ways to accomplish the task.

Cognitive robotic frameworks were popular in the 1990s and strongly influenced by the AI, where planning was one of the main keys. Indeed, one of the strengths of such frameworks was their planning modules built around a sensed reality. A good example of cognitive frameworks was Saphira [6], based on a behavior-based cognitive model. Some of its low-level functionality was rewritten as a C++ library called ARIA [7] that it is still supplied with the popular robotic platforms from MobileRobots/ActivMedia. Even though the underlying cognitive model is usually a good practice guide for programming robots, this hardwired coupling often leads the user to problems that are difficult to solve while trying to do something the framework is not designed to do.

Current robotic frameworks focus their design on the requirements that robotics applications need and let the user (the programmer) choose the organization that better fits with the specific application. The main requirements driving the designs are: multitasking, distribution, ease of use, and code reusability. Another requirement, that we believe is the main key, is the open source code, which creates a synergy between the user and the developer.

The key achievements of modern frameworks are the hardware abstraction, hiding the complexity of accessing heterogeneous hardware (sensors and actuators) under standard interfaces, the distributed capabilities that allow running complex systems spread over a network of computers, the multiplatform and multilanguage capabilities that enable the user to run the software in multiple architectures, and the existence of big communities of software that share codes and ideas.

As mentioned before, we believe that open source plays a major role in the development of modern robotic frameworks. Proof of this is the two most popular robotic frameworks: Player/Stage [8–10], which has been the *standard de facto* in the last decade and ROS [11], which is taking its place currently. As seen in other major software projects as GNU/Linux kernel or the Apache web server, to name but a few, the creation of communities that interact and share codes and ideas could be greatly beneficial to the robotic community. The main examples of open source modern frameworks are the aforementioned Player/Stage and ROS. Another important example is ORCA [12, 13]. In this work, we describe them briefly.

There are other open source frameworks that have had some impact on the current state of the art, such as RoboComp [14] by Universidad de Extremadura, CARMEN [15] by Carnegie Mellon, and Miro [16] by University of Ulm. All these use some component-based approach to organize robotic software using ICE, IPC, and CORBA, respectively, to communicate their modules.

We can find non-open-source solutions as well, such as Microsoft Robotics Studio or ERSP by Evolution Robotics.

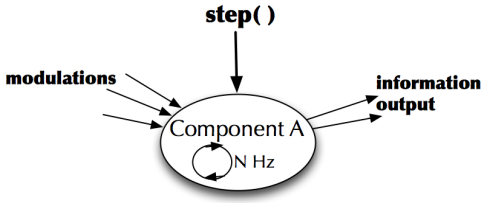

Fig. 1. BICA component.

3 Basic principles

The basic building block in BICA is the **component** (Figure 1), which is the basic unit of functionality. The main idea is to build a component that does only one thing, but efficiently. A component is composed of three main parts:

- **Modulations**: The modulation methods set operation modes or set up the next component iterations.
- **Execution**: All the components inherit from the virtual class `component`, which defines the mandatory methods to be implemented. The most important method is `step()`. This method performs an iteration of this component. This is the entry point for a **component-explicit execution**.
- **Output**: The results method is used to get the information produced in the last iteration.

It is noteworthy to highlight that modulations and results methods only write or read internal variables; so the execution time of these methods is in the range of a few microseconds. The computation time of a component can be assumed to be the `step()` method execution time.

Class `component` also determines that each component is a Singleton. This makes it easy to have only one instance of each component, and obtain a reference to it from any other component. The next code describes the initialization and `step()` method of the component A, that uses the component B to get some information. This information is used to modulate the execution of component C. Note the order of the calls. If the component needs information, it executes the `step()` method first. If the component modulates another component, it calls the modulation method before calling the `step()` method.

```
//Two components used by A
B A::b;
C A::c;

A::A()
{
        b = B::GetInstance();
        c = C::GetInstance();
}

A::step()
{
        b->step();

        int info = b->getInfo();
```

```
        //Do component A work

        c->setInfo(info);

        c->step();
}
```

The scheduler implemented in this architecture calls the `step()` method of a list of components. Usually, this list only contains a reference to the top-level component, which defines a behavior that can be specified in a setup file. Additionally, we can add more references to components from the monitoring tools in order to execute more components. The implementation is simple:

```
while(true)
        for(i=CompList.begin();i!=CompList.end();++i)
                i->step();
```

Each component is set up to a different frequency, depending on the particular function it does. As a simple orientation, the frequency set to any perceptive component depends on the time for which this information can be considered valid for the current actuation. As an example, image processing is usually made at the camera frame rate (30 Hz) and the locomotion controller sends to the walking engine speed commands at 2 Hz. To carry out this property, the class `component` implements methods `setFreq()` (called during component initialization) and `isTime2Run()`, that returns true if the elapsed time since the last time it returned true is longer than the one specified by the frequency (500 ms if 2 Hz, for example). Using this method, a usual `step()` implementation is

```
A::step()
{
        b->step();

        if(isTime2Run())
        {
                int info = b->getInfo();
                c->setInfo(info);
        }

        c->step();
}
```

The ideas presented above add two important characteristics to this architecture:

- The behavior architecture is **thread safe**. The scheduler does not create multiple threads to execute components. Only one thread calls sequentially to the scheduler list of components. This thread executes in cascade the components, in the order defined depending on the relation of the components (modulation or results).
- If any of the components sporadically spends more time than that desired, the systems suffers from what in real-time literature is called **graceful degradation**. The execution of the other components is delayed, but no executions are cancelled or overlapped. In the development phase of the components, offender components are detected because `istime2Run()` methods of each component periodically test if the frequency is achieved, generating a warning if not.

– The set of of components that a component can activate varies dynamically. As there is no explicit deactivation method, its step function is simply not called anymore; we have to design the component having in mind that a component does not know when it is going to be called again. This is called **quiet shutdown**.

Components can be very simple or very complex. Simple components communicate with the underlaying system methods to communicate with sensors or motors, or use a fixed number of components. Complex components can be implemented as finite state machine, changing the set of components it activates dynamically depending on the state. We have developed a useful tool for designing these complex components. This tool generates the code of the graphically represented behavior.

4 Extensions

The previous section described the basic principles of our software architecture. These principles are the core of the implementation of an architecture that follows the principles described in this paper. This section describes optional extension to this architecture that we have developed to have more functionality when developing any of the possible applications.

4.1 Communications

By the communication mechanism, components can be acceded from remote applications or even components running in other robots. This mechanism is implemented using the ZroC ICE communications framework. This framework hides all the complexity of programming over sockets and provides a convenient RPC paradigm for our communications. Each robot runs an ICE broker with a predefined set of interfaces that connects directly to the component's methods. This is allowed only to the modulation or retrieving information method of components. This is a convenient way to implement teleoperation applications or cooperating behaviors in groups of robots.

Communication mechanism uses a separate thread for managing the remote calls. The components that can be remotely called must implement a mutual exclusion mechanism to have one thread active executing in a component (in the `step()` method or in any modulation/retrieving information method).

The next section presents a debugging mechanism using the communication approach described in this section. The data exchanged, a list of structures that defines graphical elements (circles, boxes, etc.), are defined as an ICE data type. When running, each part of the communication (computer GUI and BICA) runs a broker, and the call to the ICE interface`getDebugData()` is directly mapped to the `Debug::getDebugData()` method.

4.2 Debug

Debug mechanisms are critical while developing new components. The low-level mechanisms (stout, stderr, and gdb) are assumed to be available all the time. Actually, class `component` already contains two functions (`startDebugInfofo()` and `endDebugInfo()`) used to periodically (each 10 s) print to stdout info about the real frequency and CPU time.

This mechanism is a convenient way of detecting if a component is consuming too much time, or if the set frequency is not reached because the system has a high load, probably produced by a high consumption component. Despite this, the development of robotic software also needs higher level of debugging mechanisms. We need to know if the sensor information (image, ultrasound, etc.) is correct, or if a self-localization algorithm is working correctly, for example.

We have developed a mechanism, which lets us graphically debug the internal information of the components. An external application can connect to this architecture to retrieve a list of graphical primitives (points, lines, ellipses, boxes, text, and images) produced by the components. A component can be debugged in the image space, in robot relative coordinates and global coordinates. To illustrate the importance of this concept, we can think of a self-localization component based on Extended Kaman Filter that could be implemented to return a list of ellipses in global coordinates, representing the state and uncertainty representing the robot position, while a self-localization component based on Particle Filters could return a list of point with an arrow, representing the position and orientation of each particle. Only the component developer knows which information is crucial for debugging, and this model affords us this functionality.

We have implemented an external graphic application that communicates with the scheduler to add components to the execution list and with a `Debug` component, also in the execution list, for which the set of components are marked for debugging. Any component able to be debugged, has to inherit from the abstract class `debuggeable` and implement a `getDebugInfo()` method:

This debugging mechanism works as follows:

- The GUI activates a set of components and marks them for debugging.
- Once selected, the debugging space (image, relatives, or absolutes), it periodically asks to the `debug` component for a list of primitives in that space.
- The `debug` component calls the `getDebugInfo()` of every component marked for debugging and returns to the GUI a list with the information retrieved.
- The GUI draws the graphical primitives retrieved.

5 Basic capabilities applied to humanoid robot

The BICA Architecture, in which we have shown the key ideas in the last section, has been implemented in the humanoid robot Nao using NaoiQi, a programming framework provided by the manufacturer. It is important to note that these ideas do not depend on the robot, and they are suitable to be implemented in any other platform. Actually, it would be very easy to be implemented inside a ROS node

for using all the services it provides, and to be available for a great variety of robots. The design of the contributions described in the next sections are focused on robots with legs, whose main sensor is a camera with a limited field of view, that can be oriented to cover all the surroundings.

Nao is a fully programmable humanoid robot. It is equipped with a x86 AMD Geode 500 Mhz CPU, 1 GB flash memory, 256 MB SDRAM, two speakers, two cameras (nonstereo), wi-fi connectivity and ethernet port. It has 25 degrees of freedom. The operating system is Linux 2.6 with some real-time patches. The robot is equipped with a microcontroller ARM 7 allocated in its chest to control the robot motors and sensors, called DCM. NaoQi is a distributed object framework which allows several distributed binaries (called brokers), each containing several software modules to communicate together. Robot functionality is encapsulated in software modules, so we can communicate to specific modules in order to access sensors and actuators.

BICA is implemented inside one of these modules. Only a limited set of BICA components, those that provide access to motors and sensors, are NaoQi-dependent, making nonblocking calls to the services that the robot provides. The rest of the components are independent of this platform.

5.1 Perception and Visual Memory

The main source of information about the robot environment is mainly provided by cameras. There are components that provide information from ultrasound sensors or buttons to the behaviors, but the camera provides the richest information, and is also a more complex sensor to manage.

The processing of the image has a set of visual stimuli relevant to the robot behaviors as output: obstacles, human faces, other robots, landmarks, balls, for example. This detection is based on the color, shape, size, and position. This processing starts labeling the color of each pixel in the image. This step is common to the detection of every visual stimuli. Next steps are particular for each stimuli.

We have taken advantage of the component-based architecture that BICA provides to avoid unnecessary processing, when some of these visual stimuli are not needed by the active behaviors during the robot operation. First of all, we have developed a component called `Camera` which labels each pixel with its color using a fast lookup table, and makes this information available for other components. Particular stimuli detection is made by particular components, using the labeled image as input, and performing the rest of the processing.

Detectors are also responsible for updating a local estimation of the local stimuli with the detection made in every cycle. Detectors use a list of extended kalman filters to maintain the detected stimuli. Behaviors retrieve this filtered information to make decisions.

5.2 Attention

The robot's camera has a limited field of view. Because of this, the robot has to move the neck to cover all the surroundings and perceive with the camera, fitted in its head, all the relevant visual stimuli. This is carried out by a visual attention system. This system is in charge of searching, tracking, and revisiting the visual stimuli.

The visual attention system developed inside the BICA architecture has the `Attention` component as the central element. This component has three functions:

1. It sends to the `Head` component the 3D points where the cameras have to be orientated.
2. It receives the perceptive requirements from the behavior components with perceptive needs. Using this information, it decides which visual stimulus governs attention in every moment.
3. It asks to the detector in charge of the visual stimulus that governs attention for a 3D point. This point is sent to the `Head` component.

The most relevant benefit of this system is that the searching and tracking is specialized for each visual stimulus. Some stimulus can be searched on the floor whereas others in the skyline; it can only have one instance of each stimulus, or more. This is decided by each detector?s developer. In [4], we have presented three different implementations of attention mechanism using the flexibility and modularity that this design allows.

6 Robotic applications

In the previous sections we have described the key ideas of the BICA architecture. In this section, we will present two different scenarios where we have applied this architecture.

6.1 Robot soccer

Using the BICA architecture we have developed a complete set of behaviors for a team of robotic players [2] of the Standard Platform League of the RoboCup. This competition presents a challenging and dynamic scenario where teams of robots have to play in a soccer match. In this league, in particular, all the robots are similar so all the efforts are focused on developing software capable of dealing with this problem.

This application needs reactive behaviors, where collisions, error in perception, and kidnappings are common. We have successfully developed a set of behaviors to deal with this problem, including cooperation among robots, navigation and self-localization algorithms, perception, and reactive visual attention. The details of this implementation can be found in[5].

6.2 Therapy for Alzheimer patients

This architecture has also shown to be adequate for a completely different application, such as using this robot for therapies in patients having Alzheimer's disease. For this scenario, we have developed a component that plays therapy scripts using new components capable of playing music and speech and managing the robot?s leads. In addition, the reproduction of the script is controlled by applications running in a tablet, or receiving commands from a wiimote.

7 Conclusion

This paper has presented novel ideas in the designing of behavior-based architecture. This architecture decomposes complex behaviors into building blocks that cooperate among them, called components. These concepts provide an effective, clean, and scalable way of designing complex behaviors in limited resources robots, with real-time requirements. Our architecture has no separation into layers, but the organization is made explicit by the relation between component activation. This contrasts with the classical approaches, such as Xavier [19] or the one proposed by Arkin [20]. In Xavier, the work is made out of four layers: obstacle avoidance, navigation, path planning, and task planning. The behavior arises from the combination of these separate layers, each with a specific task and priority. Arkin designed a hybrid architecture, in which the behavior is divided into three components: deliberative planning, reactive control, and motivation drives. Deliberative planning made the navigation tasks. Reactive control provided with the necessary sensorimotor control integration for response reactively to the events in its surroundings. Motivation drives were responsible for monitoring the robot behavior.

ROS [11] is nowadays the reference in software architectures. It decomposed applications into nodes that can communicate among them using direct messages or notifications to published data. It is thread safe and provides many graphical and test-based tools for debugging, monitoring, and developing. ROS community is very active, developing many software libraries and drivers that can be easily reused. Our approach is a more compact design. Instead of making each component a separate process, all the BICA components share the same memory space and the calls are local, making our approach more efficient for embedded applications, but limited in distributed applications. The design of each component as a singleton makes it another very easy to use component without any risk.

We have described the details of the execution of these components. These details give this architecture some important characteristics when developing real-time robotic applications: thread safe approach, graceful degradation to high-load events, and efficient perceptive pipeline. We have also described how we have used these concepts to develop visual attention or debugging mechanisms. One of the most important characteristic of this paper is that these concepts are described from a technical point of view, being useful to incorporate these ideas to any robotic software.

This architecture has been successfully used to deal with two complex applications such as the robot soccer or the therapy for Alzheimer patients. The robot soccer presents a challenging environment where the robot perceives the relevant visual stimulus, processes this information, and generates a fast response in terms of actuation and active perception commands. We have demonstrated how our approach is efficient enough to deal with perception, navigation, self-localization, team coordination, action generation, and active vision when all the processing is performed onboard. The perception requirements and component activations dynamically change during the robot operation, and our approach adapts to these changes, saving computing resources. The other application, focused on using new interfaces to communicate with the robot during therapies, shows how the approach is also valid in such a different environment.

We have also shown that this architecture is convenient for developing robotic applications. Our approach is scalable and provides a simple way to decompose the functionality in components that can be run isolated during development phase and being debugged with the efficient debugging mechanism described in this paper. At this time, this architecture is mature enough to be used in any application. It has been developed for the humanoid robot Nao, but we are currently working on an implementation inside an ROS node, obtaining the benefits of ROS, and the properties of our approach.

Acknowledgments

This research has been sponsored by grant No. DPI2013-40534-R by Spanish Ministry of Economy and Competitiveness corresponding to SIRMAVED project[21].

References

1. R. A Brooks, *Intelligence Without Representation*, Artificial Intelligence 47 (1991) 139-159.
2. F. Martín, C. Aguero, J. M. Caas, E. Perdices, *Humanoid Soccer Player Design*. Robot Soccer. Ed: Vladan Papic, pp 67-100. IN-TECH, 2010.
3. F. Martín, C. Aguero, J. M. Caas, *Localization of legged robots combining a fuzzy-Markov method and a population of extended Kalman filters*. Robotics and Autonomous Systems, Volume 55, pp 870–880, 2007.
4. F. Martín, C. Aguero, L.Rubio, J. M. Caas, *Comparison of Smart Visual Attention Mechanisms for Humanoid Robots*. International Journal of Advanced Robotic Systems: Smart Sensors for Smart Robots, Vol. 9, pp 1-10. 2012.
5. F. Martín, C. Aguero, J. M. Caas, P. Martínez, M. Valenti, *RoboTherapy with Alzheimer Patients*, International Journal of Advanced Robotic Systems: Humanoid. Vol. 9, pp 1-7. 2012.
6. K. Konolige, K. Myers, E. Ruspini, A. Saffiotti, *The Saphira architecture: A design for autonomy*, Journal of experimental & theoretical artificial intelligence 9 (2-3), 215-235. 1998.

7. K. Konolige, *Saphira robot control architecture*, Technical Report, SRI International, Menlo Park, Calif, USA, 2002.
8. B. Gerkey, R. T. Vaughan, A. Howard, *The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems*, In Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003), pages 317-323, Coimbra, Portugal, June 2003.
9. H .J. Collett, B. A. MacDonald, B. Gerkey, *Player 2.0: Toward a Practical Robot Programming Framework*, In Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005), Sydney, Australia, December 2005.
10. R. T. Vaughan, *Massively multi-robot simulations in Stage*, Swarm Intelligence, 2(2-4):189-208, 2008.
11. M. Quigley, C. Ken, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, Y. Andrew, *ROS: an open-source Robot Operating System*. Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics, 2009.
12. A. Brooks, T. Kaupp, A. Makarenko, A. Orebck, S. Williams, *Towards Component-Based Robotics*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005) p. 163–168. 2005.
13. A. Makarenko, A. Brooks, T. Kaupp, *On the Benefits of Making Robotic Software Frameworks Thin*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007). Workshop on Evaluation of Middleware and Architectures. 2007.
14. R. Cintas, L. J. Manso, L. Pinero, P. Bachiller, P. Bustos, *Robust Behavior and Perception using Hierarchical State Machines: A Pallet Manipulation Experiment*. Proceedings, Journal of Physical Agents, ISSN 1888-0258. Vol. 5, No. 1, pp 35-44. March 2011.
15. M. Montemerlo, N. Roy, S. Thrun, *Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit*. IROS 2003: 2436-2441. 2003.
16. G. K. Kraetzschmar, H. Utz, S. Sablatng, S. Enderle, and G. Palm, *Miro - Middleware for Cooperative Robotics*. Proceedings of RoboCup-2001 Symposium, volume 2377 of Lecture Notes in Artificial Intelligence, pages 411-416, Berlin, Heidelberg, Germany, 2002. Springer-Verlag.
17. M. Henning, *The Rise and Fall of CORBA*, ACM Queue Magazine (Vol 4, Issue 5, June 2006).
18. J.M. Canas, V. Matellán, *From bioinspired vs psychoinspired to ethoinspired robots*. Robotics and Autonomous Systems, Volume 55, pp 841-850, 2007.
19. R. Simmons, R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, M. Veloso, *Xavier: Experience with a Layered Robot Architecture*. SIGART Bull., Vol. 8, No. 1-4, pp. 22–33. 1997.
20. Stoytchev, A., Arkin, R.C., *Combining deliberation, reactivity, and motivation in the context of a behavior-based robot architecture*, Computational Intelligence in Robotics and Automation, 2001. Proceedings 2001 IEEE International Symposium on , vol., no., pp.290–295, 2001
21. Miguel Cazorla, José García-Rodríguez, José María Cañas Plaza, Ismael García Varea, Vicente Matellán, Francisco Martín Rico, Jesús Martínez-Gómez, Francisco Javier Rodríguez Lera, Cristina Suarez Mejias and Maria Encarnación Martínez Sahuquillo. *SIRVAMED: Development of a comprehensive robotic system for monitoring and interaction for people with acquired brain damage and dependent people*. XVI Conferencia de la Asociacin Española para la Inteligencia Artificial (CAEPIA) (2015).