RoboComp: a Tool-based Robotics Framework

Luis Manso, Pilar Bachiller, Pablo Bustos, Pedro Núñez, Ramón Cintas, and Luis Calderita*

Robotics and Artificial Vision Laboratory.
University of Extremadura, Spain
{lmanso,pilarb,pbustos,pnuntru,rcintas,lv.calderita}@unex.es
http://robolab.unex.es

Abstract. This paper presents RoboComp, an open-source component-oriented robotics framework. Ease of use and low development effort has proven to be two of the key issues to take into account when building frameworks. Due to the crucial role of development tools in these questions, this paper deeply describes the tools that make RoboComp more than just a middleware. To provide an overview of the developer experience, some examples are given throughout the text. It is also compared to the most open-source relevant projects with similar goals, specifying its weaknesses and strengths.

Keywords: robotics framework, software usability, programming tools

1 INTRODUCTION

Robotics software has to deal with very specific problems such as complexity, code reuse and scalability, distribution, language and platform support, or hardware independence. These problems should be addressed with appropriate software engineering techniques and be transparent to the developer when possible.

Software complexity, from the developer point of view, is an important topic because scalability decrease as complexity increases. The robotics community is already aware of this fact and has steered towards component oriented programming (COP) [1]. Despite the ostensible consensus regarding the use of COP, many different approaches have been developed in order to deal with those robotics specific issues.

We propose RoboComp, a robotics framework that focuses on ease of use and rapid development without lost of technical features. RoboComp is based on Ice[2], a lightweight industrial-quality middleware. Instead of spending large amounts of time developing an ad-hoc middleware, it was found preferable to reuse an existing one.

The most remarkable strength of RoboComp is the set of tools it is accompanied with. These tools make programming an easy and faster experience.

* This work has been supported by grant PRI09A037, from the Ministry of Economy, Trade and Innovation of the Extremaduran Government, and by grant TSI-020301-2009-27, from the Spanish Government and the FEDER funds.

RoboComp also provides a wide variety of components, ranging from hardware interface or data processing to robot behavior. The web of the project gives access to an extensive on-line documentation covering all the information users may need. New components can be easily integrated with existing ones. Moreover, RoboComp components can communicate with other widely used frameworks. In particular, successful integration has been achieved with Orca2, ROS and Player.

2 MAIN CHARACTERISTICS

Several robotics frameworks have been previously proposed to meet different requirements (e.g. Carmen[4], JDE[5], Marie[6], Miro[7], MOOS[8], OpenRTM-aist[9], Orca2[10], OROCOS[11], Player[12], ROS[13], YARP[14]). For a best understanding of what RoboComp provides, this section describes its main characteristics.

2.1 Middleware issues

Creating a new ad-hoc middleware would have involved a considerable amount of work, not just to create the middleware but also to maintain it. We chose Ice because it meets all the requirements we experimentally identified for a robotics framework:

- Different optional communication methods (e.g. RMI, pub/sub, AMI, AMD).
- IDL-based, strongly typed interfaces.
- Good performance with low overhead.
- Efficient communication.
- Multiple language support (e.g. C++, C#, Java, Python).
- Multiple platform support (e.g. Linux, Windows, Mac OS X).

Orca2[10] is also based on Ice. However, RoboComp presents additional features that complements the middleware layer: a) It provides a well-defined component structure that ease software development; b) it includes several tools that reduce the effort needed to create and maintain software components. One of the main reasons why a new framework was created, instead of adopting Orca2, is that it would involve imposing our component structure to previous users. Nevertheless, although they are independent projects, using the same middleware facilitates the interoperation between both frameworks. Therefore, RoboComp and Orca2 users can share their components and benefit from the advantages of both projects.

Despite other frameworks have developed their own communication engine [6][12][13][14], we found preferable to adopt Ice. Besides its features and the time saving, it allows a better interoperability with other frameworks. Moreover, Ice has been used in various critical projects[3], so it can be considered very mature because of the extensive testing it has passed.

2.2 Tools and classes

Two of the aspects in which RoboComp extends Ice are the tools and the numerous set of classes it is equipped with. The set of tools is what enhances user experience, making easier to develop and deploy complex systems. They are deeply described in section 3. The set of classes comprises different issues related to robotics and computer vision such as matrix computation, hardware access, Kalman filtering, graphical widgets, fuzzy logic or robot proprioception.

Among the different available classes, the robot proprioception class, which we call *InnerModel*, plays an important role in robotic software. It deals with robot body representation and geometric transformations between different reference frames. *InnerModel* is based on an XML description of the robot kinematics read from file. In this file, the transformations nodes (joints and links) are identified and described. *InnerModel* also provides different methods to estimate projections and frame transformations. Unlike other frameworks such as [13], RoboComp does not offer a separate component for such functionality. This decision was taken in order to reduce both, software complexity and latency. Indeed, the memory overhead of replicating the same object is negligible.

2.3 HAL characteristics

Component oriented programming frameworks for robotics provide an effective way to keep software complexity under control. However, most of them have ignored one of the most important contributions of Player[12]: the idea of a robotics hardware abstraction layer (HAL). We think this idea is extremely important for code reuse and portability. All sensors and actuators of the same type can provide a common interface as long as they do not have very specific features. When possible, it is preferable to use the same interface: **a)** different users, researchers, or companies can share their components regardless of the underlying hardware, **b)** it reduces the impact of hardware updates, **c)** it prevents software deprecation [14].

Since it is possible to find hardware that does not fit the standard interface, RoboComp does not force the use of the proposed interfaces, just recommends it and emphasizes its benefits. Currently, RoboComp has defined the following HAL interfaces: Camera, DifferentialRobot, GPS, IMU, JointMotor, Joystick, Laser and Micro.

2.4 Component structure

In order to ease development, a framework must provide a well-defined component structure. It helps users to understand which are the necessary code elements as well as the required links among these elements that lead to a correct deployment.

The skeleton of RoboComp components (figure 1) is composed of three main elements: the server interface, the *worker* and the proxies for communicating with other components. The *worker* class is the responsible of implementing

the core functionality of each component. The server interface is a class derived from the Slice (Ice IDL) definition interface, which implements the services of the component. These services are generally handled by interacting with the *worker* class. Since delays are highly undesirable, both in the interface and in the core class, they run in different threads. Proxies are also of remarkable importance within a component. These, which are usually owned by the *worker*, are instances of auto-generated classes that provide access to other components. Components also include a configuration file where all the operational and communication parameters are specified. Using this configuration, the main program makes the necessary initializations to start the component.

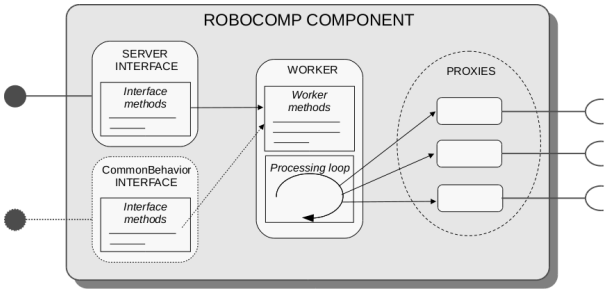

Fig. 1: General structure of a RoboComp component.

Additionally, components may implement a common interface named *CommonBehavior*. This interface provides access to the parameters and status of the component and allows changing some aspects of its behavior at run time (for example, the frequency of its processing loop). This interface can be used to dynamically analyze the status of a component, providing a means of determining wrong configurations or potential execution errors.

3 TOOLS

A good robotics framework should not be just a bag of middleware features. Programming software for autonomous robots also involves other issues such as ease of use or readiness for an agile software development lifecycle. Thus, RoboComp is equipped with different tools that complement the middleware layer.

3.1 componentGenerator

Using a well-defined structure, the process of creating the skeleton of new components can be automated. Thus, to save the programmer from this tedious task, RoboComp includes a component generator tool. The component generator frees the programmer from middleware issues. These code pieces are automatically

generated, so the programmer does not even have to read the whole component code. Our experience with new RoboComp users shows the great benefits of using this tool.

As an example of how to develop interacting RoboComp components using the *componentGenerator* tool, it is shown the source code of two components that communicate to compute the addition of two numbers. This example shows the few lines of code a RoboComp user has to write. The listings shown below correspond to the source files that should be modified by the programmer to add the desired behavior to both components. The client component reads two numbers, asks the server for the result of the addition of those numbers and prints the result. The server waits for client requests and computes the addition of numbers when it is requested to.

Listing 1.1: *worker.cpp* in the client side

```
1  #include "worker.h"
2
3  Worker::Worker(RobolabModAddTwoIntsServer::AddTwoIntsServerPrx
       addtwointsserverprx, QObject *parent) : QObject(parent){
4    addtwointsserver = addtwointsserverprx;
5    connect(&timer, SIGNAL(timeout()), this, SLOT(compute()));
6    timer.start(BASIC_PERIOD);
7  }
8
9  Worker::~Worker(){}
10
11 void Worker::compute(){
12   int a, b, res;
13   cin>>a;  cin>>b;
14   res = addtwointsserver->addTwoInts(a,b);
15   cout<<"a+b="<<res<<endl;
16 }
```

The listing 1.1 shows the only source file that should be modified in the client component. Remaining files of the component are not shown since their code is auto-generated and do not need to be modified. In listing 1.1, only the highlighted lines are written by the programmer. Lines in gray are auto-generated. The line in brown (line 14) shows the sentence invoking the remote method on the server.

Listing 1.2: *AddTwoIntsServer.ice*

```
1  #ifndef ADDTWOINTSSERVER_ICE
2  #define ADDTWOINTSSERVER_ICE
3
4  module RobolabModAddTwoIntsServer{
5    interface AddTwoIntsServer{
6      int addTwoInts(int a, int b);
7    };
8  };
9
10 #endif
```

Listings from 1.2 to 1.4 are the source files of the server component that have to be modified by the programmer in order to add the specific functionality. As in the listing 1.1, lines written by the programmer are highlighted.

Listing 1.2 is the Slice definition, where the programmer has to specify the services that will be provided to other components. Listing 1.3 shows the server interface. It is the implementation of the class derived from the Slice definition that handles client requests. Besides the processing loop, the *worker* class (listing 1.4) includes the final implementation of the core of the component. As it can be observed in this example of server-component development, the programmer only has to include those lines related to service implementation without taking into account low-level issues.

Listing 1.3: *AddTwoIntsServerI.cpp*

```
1  #include "AddTwoIntsServerI.h"
2
3  AddTwoIntsServerI::AddTwoIntsServerI( Worker *_worker, QObject *parent)
       : QObject(parent) {
4    worker = _worker;
5    mutex = worker->mutex;
6  }
7
8  AddTwoIntsServerI::~AddTwoIntsServerI(){}
9
10 int AddTwoIntsServerI::addTwoInts(int a, int b, const Ice::Current&) {
11   return worker->addTwoInts(a,b);
12 }
```

Listing 1.4: *worker.cpp* in the server side

```
1  #include "worker.h"
2
3  Worker::Worker(QObject *parent) : QObject(parent){
4    mutex = new QMutex;
5    connect(&timer, SIGNAL(timeout()), this, SLOT(compute()));
6    timer.start(BASIC_PERIOD);
7  }
8
9  Worker::~Worker() {}
10
11 void Worker::compute() {}
12
13 int Worker::addTwoInts(int a, int b) {
14   return (a + b);
15 }
```

Due to the use of a well-defined component structure, it is easier to create code generation or modification scripts. An example of this advantage is *addProxies*. This tool allows the programmer to automatically modify a previously created component including all the middleware-related code that is necessary to connect to new interfaces.

3.2 managerComp

Components are independently executed programs that interact with each other. They can be executed manually, but when a system contains more than a few

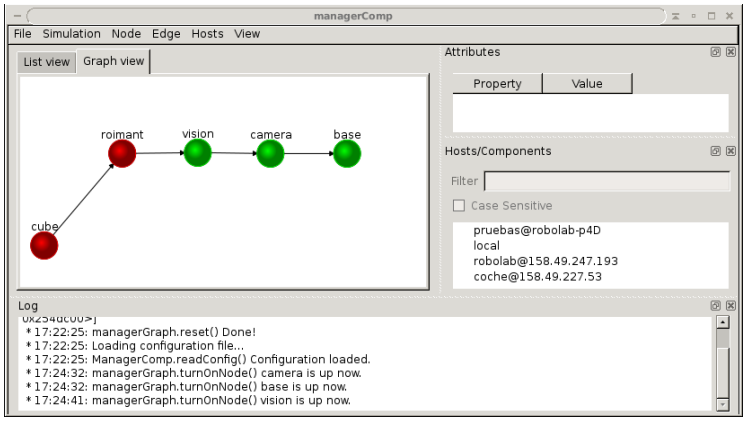

Fig. 2: Graphical interface of the *managerComp* tool.

components, it becomes hard to manage. In order to make this process easier, a component manager, *managerComp*, has been developed (figure 2).

This tool allows users to graphically build and run a system in an intuitive way: **a)** components located within a host list are automatically found; **b)** components are included in the system by using a simple drag&drop operation; **c)** component dependencies are specified drawing arrows connecting nodes of the system graph (see Graph View tab of figure 2); **d)** components can be started/stopped by simply clicking the corresponding node of the graph view. This tool also facilitates system execution: when the user starts a component, all its dependencies are automatically satisfied. All these operations can be performed for components in both, local and remote hosts.

Moreover, *managerComp* can be used not only to manage local or remote components, but also to achieve certain level of introspection. As it was explained in section 2.4, components may optionally implement the common interface *CommonBehavior*. The manager tool uses this interface in order to give visual access to the parameters of the components. This is useful to detect wrong configurations or even to change operational attributes. Thanks to this feature, the manager tool can be used to help users diagnosing potential errors which would otherwise be harder to detect.

3.3 monitorComp

Developing new components may involve the application of a test process to evaluate its operation. Without a specific tool, testing a component entails the creation of another component connecting to the new one in order to check its functionality. To facilitate this test process, RoboComp includes *monitorComp*, a tool for component connection and monitoring. This tool allows the programmer either to include custom monitoring code or to use one of the templates available to test HAL components.

monitorComp provides a graphical interface that helps the programmer to carry out the component test process in an easy way. The first step is to connect to the corresponding component. The programmer has to indicate the endpoint of the component as well as its Slice definition file. Once the connection data is introduced, the next step is the insertion of the monitoring code. The language used for writing this code is Python. It allows to minimize the size and the complexity of the testing code. Once the test code is written, *monitorComp* can run tests without any user input. All this information can also be read from file.

3.4 replayComp

The *replayComp* tool (figure 3) records the output of a set of components in order to subsequently emulate their roles. During emulation, it can run at the desired speed or manually step by step. This feature is extremely useful for debugging purposes: if the inputs are the same, the behavior of programs should also be the same. Thus, it is very helpful when trying to reproduce errors.

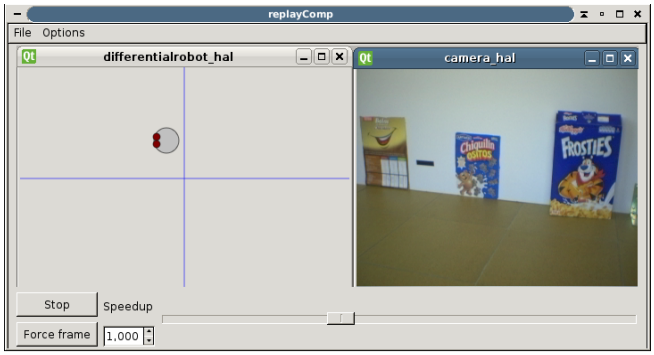

Fig. 3: Graphical view of *replayComp*.

This tool can run in two different modes: capture and replay. In capture mode, *replayComp* can connect to different components and record their outputs. Its output file can then be transparently used in a replaying stage as input for other components (i.e. components reading data from replayComp will not be able to differentiate replayComp from real components). This way, it enables software development when no robot is physically available.

Currently, replayComp supports all the HAL component interfaces out of the box. Besides, the set of components that replayComp can record and replay can be extended by writing small Python scripts that can be included in this tool as *plugins*.

3.5 Simulator support

A replay component is very useful for debugging software that does not need the robot to be active. However, when debugging active robot behaviors, a replay component is not enough. With a real robot, reproducing the same software behavior several times is a useful but very difficult task. Additionally, at initial phases of component development, it can be very helpful to test its operation considering ideal conditions, instead of a noisy real environment. In order to meet these conditions, a robotics simulator seems a very adequate tool. Therefore, RoboComp includes support for Stage and Gazebo, two widely used open-source 2D and 3D robot simulators.

In order to make the use of the simulator transparent for the software, the simulator support has been included by extending the HAL components. As a result, higher-level components do not need any change to run whether on the simulator or on a real robot.

3.6 loggerComp

This tool is an optional standard output alternative designed to analyze the execution of components and their interactions. It provides a graphical interface to display the information of interest, allowing the user to filter by: type, component, time stamp, source code file and line of code. In order to use loggerComp, components only have to include the *loggerComp* proxy (which can be automatically done through the *addProxies* utility) and use the provided helper class *qLog*.

4 FRAMEWORK COMPARISON

Throughout this paper, different robotics framework features have been discussed. Depending on the task, robots may have to use a particular platform or programming language. Different communication methods may also be very interesting features in order to fit to the different interoperational requirements and communication patterns. Finally, due to the complexity of the problems to solve, the toolset has been proved to be one of the most remarkable characteristics of a robotics framework. In order to provide a global view of the features and benefits of RoboComp, the rest of this section presents a comparison with the most relevant frameworks.

Table 1 shows the license, supported platforms and programming languages of the different frameworks. Note that both RoboComp and Orca2 support a wide variety of possibilities derived from the use of the Ice middleware.

Table 2 shows the middleware used by the different frameworks and some of its associated features. Those frameworks providing an IDL-based strongly typed interface are usually preferable: they are easier to understand and, therefore, reduce the possibility of programming errors. In RoboComp and Orca2, the Ice compilation tools are used in order to convert IDL interfaces into code of specific

Table 1: Middleware general aspects

Framework	**License**	**Supported Platforms**	**Programming Languages**
Carmen	GPL	Linux Windows	C C++ Java Python
Marie	LGPL	Linux	C C++
Miro	GPL	Linux	C C++
MOOS	GPL	Linux Mac OS X	C++ Matlab
OpenRTM	EPL	Linux Windows FreeBSD	C++ Python Java
Orca2	LGPL/GPL	Linux Windows Mac OS X Android iPhone	C++ C# Python PHP Ruby Java Objective-C
OROCOS	LGPL/GPL	Linux Windows	C++
RoboComp	GPL	Linux Windows Mac OS X Android iPhone	C++ C# Python PHP Ruby Java Objective-C
ROS	BSD	Linux Windows	C++ Python Octave Lisp
YARP	GPL	Linux Windows QNX	C++ Java Ruby Python Lisp

programming languages. Frameworks using CORBA IDL are also equipped with similar tools. The remaining frameworks do not include any IDL description facilities. Regarding the communication methods, Ice-based frameworks provide a good variety of both synchronous and asynchronous communication mechanisms. This feature allows the programmer to select the most suitable component interaction method regarding to their operation requirements.

Table 2: Networking and API

Framework	**Middleware**	**Comm. Methods**	**IDL**
Carmen	IPC	pub/sub query/response	No IDL
Marie	ACE	data-flows (socket-based)	No IDL
Miro	ACE+TAO	sync/async client/server	CORBA IDL
MOOS	custom	pub/sub	No IDL
OpenRTM	ACE	pub/sub query/response	CORBA IDL
Orca2	Ice	RPC, AMI, AMD, pub/sub	Ice IDL
OROCOS	ACE	commands, events, methods, properties, data-ports	CORBA IDL
RoboComp	Ice	RPC, AMI, AMD,pub/sub	Ice IDL
ROS	custom	pub/sub query/response	No IDL
YARP	ACE	asynchronous data-flows	No IDL

Table 3 shows the tools provided by the reviewed frameworks. *Text* means that the framework provides a console-based tool, *gui* corresponds to a graphical

Table 3: Tools

Framework	**Code Gen.**	**Manager**	**Replay**	**Simulator**	**Log**	**Monitor**
Carmen	no	no	gui	custom 2d	gui	no
Marie	no	no	no	stage gazebo	no	no
Miro	no	no	no	no	gui	no
MOOS	no	text	gui	uMVS	gui	no
OpenRTM	gui	gui	no	stage gazebo	gui	no
Orca2	no	no	gui	stage gazebo	no	no
OROCOS	no	text	no	no	no	no
RoboComp	gui	gui	gui	stage gazebo	gui	gui
ROS	partial (text)	text	gui	stage gazebo	gui	gui
YARP	no	gui	no	icubSim	text	no

user interface, and *no* to the unavailability of a tool. Column *'code gen.'* represents the availability of a code generation tool. Despite RoboComp, OpenRTM and ROS provide this facility, only RoboComp and OpenRTM produce ready-to-use code: the ROS *roscreate_pkg* tool creates the directory tree structure of a ROS package as well as the necessary files for compilation, however it does not deal with source code generation. In addition, RoboComp provides a tool for code modification that allows adding new communication interfaces to a previously written component. Column *'manager'* represents the existence of an easy to use and specific tool to deploy and inspect the status of the deployed components (Orca2 uses IceGrid to deploy components but it is not framework specific or user friendly). Columns *'replay', and 'log'* represent the availability of tools for component replaying and for logging messages, respectively. The *'simulator'* column lists the simulators the different frameworks provide connection to. Allowing components to work with a simulator without even changing the code is a extremely useful feature. Thus, RoboComp provides a completely transparent use of Stage and Gazebo, two well-known robot simulators. The *'monitor'* column represents the availability of a tool to monitor the information that a component is working with, not just its status (i.e. images of a camera or the pose of a robot platform). Both ROS and RoboComp have such tool. However, *monitorComp* is more advanced than the tools ROS is equipped with: *rxbag* has a plugin system that allows users to display new types of data but it can only work with off-line data; *rxplot* works with on-line data but only with a few data types. *monitorComp* provides both features.

On the basis of this comparison, we think that RoboComp has a remarkable set of technical and user-oriented features that make it an outstanding robotics framework. However, we are aware of the big effort that would involve for new users to migrate their software from a framework to another. Thus, the on-line documentation of RoboComp describes the procedure to achieve interoperation with Orca2 and ROS, two of the most widely used frameworks in robotics.

5 CONCLUSIONS AND FUTURE WORKS

This paper has described RoboComp, a tool-based robotics framework, emphasizing the goals it intended to meet and the design decisions that have been made. RoboComp stands up in the comparison in several aspects such as ease of use, portability, language support or its toolset.

Due to the big efforts made by the scientific community, robotics frameworks are improving quickly. Despite RoboComp improves developer experience in comparison to other frameworks, enhancements are already being made: new features, new tools, or more introspection capabilities are under way.

References

1. J. He, X. Li and Z. Liu. *Component-based Software Engineering: the Need to Link Methods and their Theories*. Proc. of ICTAC 2005, Lecture Notes in Computer Science 3722, pp. 70-95, 2005
2. M. Henning and M. Spruiell. *Distributed Programming with Ice*. 2009.
3. ZeroC. *ZeroC Customers, http://zeroc.com/customers.html*.
4. M. Montemerlo, N. Roy and S. Thrun. *Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit*. Proc. of International Conference on Intelligent Robots and Systems, 2003.
5. J. M. Cañas, J. Ruíz-Ayúcar, C. Agüero and F. Martín. *Jde-neoc: component oriented software architecture for robotics*. Journal of Physical Agents, Vol. 1, No. 1, pp 1-6, 2007.
6. C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky and F. Michaud. *Using MARIE for Mobile Robot Component Development and Integration*. Software Engineering for Experimental Robotics, Springer, pp. 211-230, 2007.
7. H. Utz, G. Mayer, U. Kaufmann, and G. Kraetzschmar. *VIP: The Video Image Processing Framework Based on the MIRO Middleware*. Software Engineering for Experimental Robotics, Springer, pp. 325-344, 2007.
8. P. Newman. *MOOS - Mission Orientated Operating Suite*. Massachusets Institute of Technology, Dept. of Ocean Engineering, 2006.
9. National Institute of Advanced Industrial Science and Technology (AIST). *RT-Middleware: OpenRTM-aist*. http://www.openrtm.org/, 2010.
10. A. Brooks, T. Kaupp, A. Makarenko, S. Williams and A. Orebäck. *Orca: A Component Model and Repository*. Software Engineering for Experimental Robotics, Springer, pp. 231-251, 2007.
11. H. Bruyninckx. *Open Robot Control Software: the OROCOS project*. Proc. of International Conference on Intelligent Robots and Systems, pp. 2523-2528, 2001.
12. B. Gerkey, T. Collet and B. MacDonald. *Player 2.0: Toward a Practical Robot Programming Framework*. Proc. of the Australasian Conf. on Robotics and Automation, 2005.
13. M. Quigley, B. Gerkey, K. Conley, J. Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler and Andrew Ng. *ROS: an open-source Robot Operating System*. ICRA Workshop on Open Source Software, 2009.
14. P. Fitzpatrick, G. Metta and L. Natale. *Towards Long-Lived Robot Genes*. Journal of Robotics and Autonomous Systems, vol. 56, num.1, p. 29-45, 2008.