Native robot software framework inter-operation

Geoffrey Biggs, Noriaki Ando, and Tetsuo Kotoku

Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology (AIST)
AIST Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan

Abstract. Component-based software is a major recent design trend in robotics. It brings many benefits to system design, implementation, maintenance and architectural flexibility. While previous methods of structuring software led to monolithic robot software incapable of easily using multiple architectures, the component-based approach, combined with some suitable software engineering decisions, opens new possibilities for interaction amongst architectures. Rather than using a single architecture for an entire robot system, we now have the opportunity to use the best architecture for the job in each part of the robot without inefficient wrapping or translation. For example, a real-time architecture for actuator control and a planning-based architecture for intelligence. In this paper, we use the example of a native ROS transport for the OpenRTM-aist architecture to illustrate the benefits of allowing architectures to interact in this way.

1 Introduction

A component-based software system, built from the ground up using a component-based middleware such as CORBA, will only have one method of communicating between components. This has been the traditional approach to component-based software since the concept was conceived. The system is designed around the architecture to be used.

In robotics, it has recently become popular to create a component-based framework, either with or without the support of an existing middleware. The framework's goal is to reduce the time for creating a new software system and to foster greater re-use of software resources.

Unfortunately, software created using a framework is only re-usable within that framework. This is not true re-use [2]. It is common for a robot developer to find functional software they wish to use, only to discover it was written for a different framework than the one chosen by the developer.

Fortunately, the component-based approach to software development lends itself well to greater inter-operation between different frameworks. It is common for a component-based framework to abstract away the details of the transportation of data from one component to another. For example, the OpenRTM-aist framework [1] provides a one-way data flow port abstraction which, assuming the same serialisation method is used, supports any method of moving the data between two ports. At a higher level, it also provides an abstraction for the basic concept of a port, allowing transports with any semantics to be created.

We can take advantage of this abstraction to integrate support for other frameworks directly in OpenRTM-aist. This paper describes the implementation of support for the ROS framework's transport [7]. The implementation provides native support for communicating between OpenRTM-aist components and ROS nodes. No wrappers or protocol translators are required, ensuring no loss in efficiency for inter-operation between OpenRTM-aist and ROS.

The next section discusses previous methods of implementing inter-operation between frameworks. Section 3 describes how ROS support was added to OpenRTM-aist. Section 4 discusses the support, including what advantages it brings. Conclusions are given in Section 5.

2 Architecture interaction

The continued development of a range of robot frameworks, with their own unique properties, has produced a continuing desire to use frameworks together. While a robot software system will likely be mainly using one framework, it may be desirable to use another at the same time in some parts of the software. Reasons for this can include:

1. The other framework includes functional software that the authors of the robot software system wish to use, without needing to rewrite or otherwise modify that software.
2. The other framework is more suited to some part of the robot under development. For example, the main framework may not support hard real-time, while another framework does. Using the other framework in the real-time control loops of the robot becomes desirable. We view this as important in future robot development as various frameworks arise with differing capabilities.

In practical experience, the first is more common. Researchers tend to develop their functional software heavily integrated into their framework of choice. When another researcher wishes to use it, they must choose either to fork the software in order to use it directly, make their framework of choice interact with the framework containing the desired functional software, or change the framework used in the rest of the system. The third option is obviously the least desirable, and may even be impossible. The first option requires a large time commitment, making the second option the most desirable. This option saves development time and is less likely to introduce new bugs into existing software.

Previous approaches to architecture interaction have included wrappers and translators, which are very similar.

In the following sections, we use the term "client" to refer to the software written by the robot developer that is executing in the primary framework. This will typically be the body of a component. The term "primary framework" indicates the framework primarily being used in the development of a robot, while “secondary framework” refers to another framework being utilised to gain some benefit that it offers for a specific part of the robot, such as real-time capability.

2.1 Wrappers

A wrapper is the most straight-forward approach to making one framework interact with another. The goal of a wrapper is to “wrap” the desired behaviour of the secondary framework in a component for the primary framework. It creates the appearance of providing that wrapped behaviour natively in the primary framework. A wrapper could also be called a mimic component. Illustrations of this method are shown in Figures 1 and 2.

Using a wrapper to enable communication between two frameworks involves treating the secondary framework as a software resource to be used by client software of the primary framework. This client software forms the wrapper. It is responsible for servicing requests from other components of the primary framework. In order to do this, it communicates with the desired behaviour of the secondary framework as with any other software or hardware resource. The secondary framework is not a first-class entity of the complete software system.

A wrapper may also be more of a translator, attempting to mediate between the two frameworks at a level closer to the protocol than at the software functionality level.

The largest problem with using a wrapper is that it introduces inefficiencies into the robot system. Even if the data types match exactly (a rare occurrence), data must be read from one connection and written to another.

Wrappers are typically specific to the data type or behaviour of the secondary framework that they are wrapping. This means that multiple wrappers are usually necessary to support all the possible behaviour offered by the secondary framework - a somewhat daunting and unreasonable time sink for any robot developer. Maintenance is also difficult. Any changes to the secondary framework require updating the corresponding wrappers. Practical experience has shown that wrappers typically become out-of-date and unusable within a short period of time. These challenges to robot developers make wrappers less than ideal in robot development and restrict the free use of multiple frameworks.

Numerous wrappers have been created over the years. They are not typically afforded much fanfare. A set of examples can be found in the library of components for the ORCA2 framework. There are several components present that wrap interfaces of the Player framework. They provide services to other ORCA2 components, the behaviour of which is defined by the Player interfaces that they wrap. A specific example is the `LaserScanner2dPlayerClient` driver[1]. This provides the behaviour of a two-dimensional laser scanner, with the actual

[1] `http://orca-robotics.sourceforge.net/group__hydro__driver__laserscanner2dplayerclient.html`

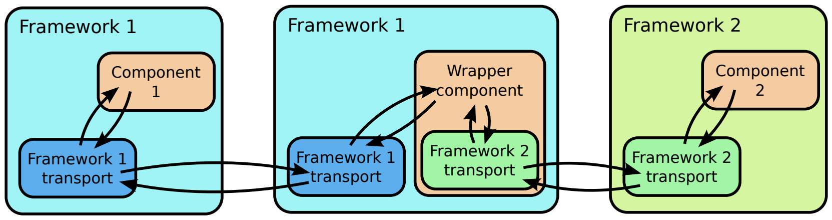

Fig. 1. Using a wrapper component is the most common approach to connecting two architectures. A separate component in the primary framework uses the secondary framework as though using any other software or hardware resource.

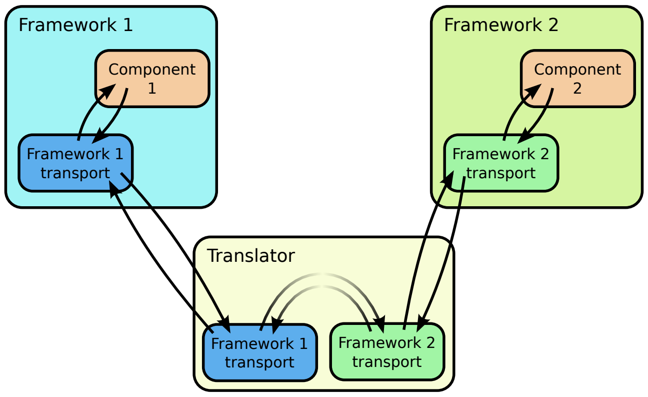

Fig. 2. A translator is a variation of a wrapper component. However, it is not a direct user of either framework, unlike a wrapper. It merely translates the protocols.

behaviour implemented on the other side of the divide between ORCA2 and Player.

Another example of the wrapper approach is the MARIE project [5]. This project was begun specifically with the goal of allowing the multiple robot software frameworks that existed at the time to interact. It features a large collection of wrappers for various robot software frameworks, including Player and CARMEN.

2.2 Multiple transports

A “transport” in a component-based software system is the method of communication between two separate components. The transport is responsible for ensuring data is moved from a source to a destination. This may involve serialisation of the data and transmission across a network using an underlying network protocol. Transports may provide one-way communication or a request-reply form of communication, in which case the transport may also be responsible for ensuring a reply of some kind is delivered to the requester.

A framework is not limited to a single transport. Multiple transports are often supported. The concept of multiple transports is not new. The Player framework has featured both TCP and UDP transports since 2004 [4]. Supporting multiple transports allows the best transport for a link between two components to be chosen. In the case of Player, the reliable TCP transport can be used in parts of the robot system where all messages must be delivered. The unreliable UDP transport can be used when missing some messages will not degrade the robot's performance - gaining the performance improvement that UDP offers in those situations.

The multiple transports concept has not yet been widely exploited for allowing communication between frameworks. It does, however, offer many advantages towards achieving this goal:

– Unlike a wrapper component, transport management code exists outside the scope of the client code. The cognitive load on the developer for utilising the alternative transport is reduced.
– No translation is necessary. The data from the secondary framework can be utilised directly in the primary framework's components.
– No extra software components existing solely to translate between frameworks are necessary. The removal of an extra communication step and change in protocol removes an inefficiency present in wrappers.
– In an ideal design, the transports can be swapped transparently to the client code. This increases the re-usability of components while maintaining flexibility to communicate with other frameworks. (We discuss what is necessary to achieve this in Section 5, below.)
– Less maintenance overhead: changes in the API of the secondary framework require changes to the primary framework's use of the transport, but these should not affect client code. The wrapper approach is effectively crippled by changes in the secondary framework.

Figure 3 illustrates how a single framework can use multiple transports to support communication with components running both the same framework and other frameworks.

The next section describes the implementation of a ROS transport for the OpenRTM-aist framework. The benefits this transport brings to both OpenRTM-aist and ROS are discussed in Section 4

3 Native ROS transport for OpenRTM-aist

This section describes the addition of a native transport for the ROS framework to the OpenRTM-aist framework.

3.1 OpenRTM-aist

OpenRTM-aist [1] is a component-based framework for intelligent systems. It features a component model that governs the life-cycle of each component, managers for controlling deployment of components, a powerful properties system

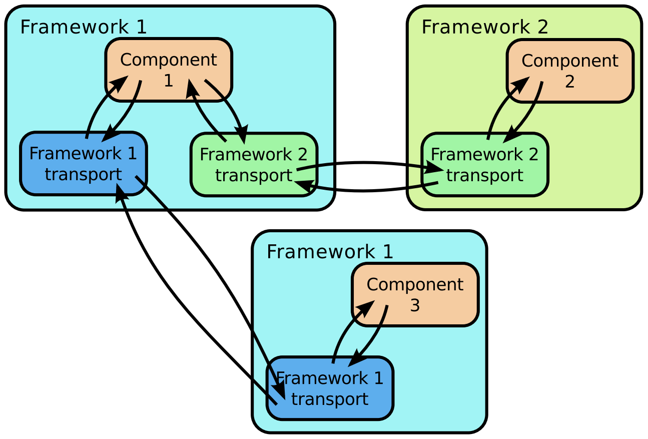

Fig. 3. Multiple transports supported within a framework allow components to directly speak the language of the peer they are communicating with. This structure applies just as well when only one framework with multiple transport types are involved.

for fine control over component configurations, and an in-depth introspection system. The introspection system complies with the Robot Technology Component standard from the Object Management Group [8], allowing OpenRTM-aist to work with any introspection tool compliant with this standard, such as the RTSystemEditor tool [9] used to manage RT Systems. The introspection system provides both inspection and control.

OpenRTM-aist has two built-in transports, both CORBA-based. The first is a basic request-reply transport. This effectively exposes a CORBA interface implementation from one component for other components to use as clients. The second transport is a one-way transport, called the data-flow transport. It uses CORBA's RPC mechanisms to effect a transfer of data from a provider to a consumer. Most importantly for the purposes of this discussion, the data-flow transport is point-to-point communication.

OpenRTM-aist relies on "ports" to support its transports. A port is defined in the RTC standard as a communication point; no further detail is given as to how that communication is carried out, making the introspection system ignorant to the transport details. (This does not stop the introspection from configuring transports, as the introspection interface can carry abstract properties to a transport for configuration purposes). Within a component implementation, a port is an object that provides a proxy for the transport connection to allow reading and/or writing of data. The abstraction of the transport implementation from the rest of the framework is what allows us to so easily implement native transports for interacting with other frameworks.

OpenRTM-aist's component model supports hard real-time control. A component in OpenRTM-aist is known as an RT Component.

3.2 ROS

ROS, the Robot Operating System, is a relatively new component-based framework [7]. It is the opposite of OpenRTM-aist in that it provides no component model or life-cycle, preferring instead to grant the developer as much freedom in component structure as possible.

Like OpenRTM-aist, ROS has a request-reply transport and a one-way transport. The request-reply transport exposes call-back functions in the cal lee to callers.

The one-way transport differs significantly from OpenRTM-aist's one-way transport. Rather than being point-to-point, it is semantically channel-based communication. Publishers place data into a channel, and any components subscribed to that channel will receive it. The channel can be persistent: if a component subscribes after the last data was written into the channel, it can still retrieve that value. This is an important capability in robotics, where irregularly- and infrequently-updating data is common. For example, a map provider.

ROS provides introspection for its communications, allowing the existing channels to be inspected from introspection tools. ROS does not directly support any form of real-time.

3.3 Native ROS transport

As the previous sections noted, OpenRTM-aist and ROS differ in a number of ways. Most importantly from the point of view of OpenRTM-aist is that ROS provides a persistent channel-based transport, a feature with wide benefits in robotics. ROS also has a very large library of functional robot software that, were it available to OpenRTM-aist, would be very useful.

From the perspective of ROS, the hard real-time capability of OpenRTM-aist would be useful in low-level real-time control of robot hardware. Currently a robot program that requires real-time must drop out of ROS to gain it.

We have implemented a new transport for OpenRTM-aist that allows RTCs to speak ROS's language. The transport can be used both to communicate with ROS components and to communicate with other RT Components using the channel-based communications.

As mentioned earlier, OpenRTM-aist uses ports to provide access to its transports. The ROS transport provides a set of new port types: channel publication, channel subscription, service provision and service utilisation.

The output and input ports provide publisher and subscriber access, respectively, to ROS channels. The request and service ports provide access to and provision of ROS services.

These ports can be used directly in a component's code. They are written to and read from using the same API as that used for the existing data ports, in the case of the input and output ports. Due to fundamental differences in the way the OpenRTM-aist and ROS service ports operate, the service ports in the ROS transport use an API similar to that used by ROS components.

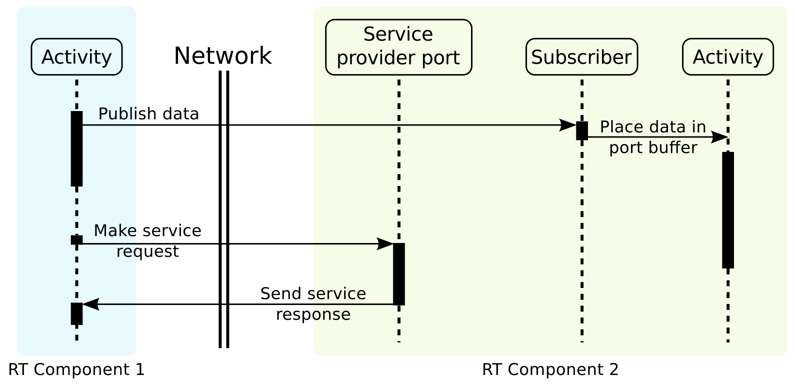

Fig. 4. The communication between processes involved in a one-way data flow and a two-way request for the ROS transport port.

As shown in Figure 5, the ROS transport in OpenRTM-aist fits into the model shown in Figure 3.

The transport itself is implemented directly in OpenRTM-aist, making it a native transport. Unlike a wrapper, which requires that the client code of the component manage the transport, all transport management is handled within the architecture. In OpenRTM-aist, this means utilising parallel threads of control (illustrated in Figure 4).

For each port that can receive data or requests, a parallel thread is started that manages the port. These are not complex tasks; typically it consists of providing execution time to the internal ROS systems that manage the transport. The details for each port type are given below. The actual implementation of the transport is left to the ROS libraries; OpenRTM-aist merely utilises those libraries. This means that any improvements made to ROS will indirectly benefit OpenRTM-aist as well.

Publishing port The publishing report is very simple. It merely places the given piece of data into the ROS channel via the standard ROS publication functions. The ROS library takes care of serialisation and transportation from that point.

Subscribing port As with the original OpenRTM-aist input port, a separate variable provides access to the most recent piece of data received. The ROS subscribing port uses a callback function, as is standard with ROS subscribers. The callback is provided by the port; the user does not need to provide it themselves. The responsibility of the callback is to copy the most-recently-received data into a buffer. When the port is “read” by the component, the next value in this buffer is copied into the target variable. A separate thread is used to give

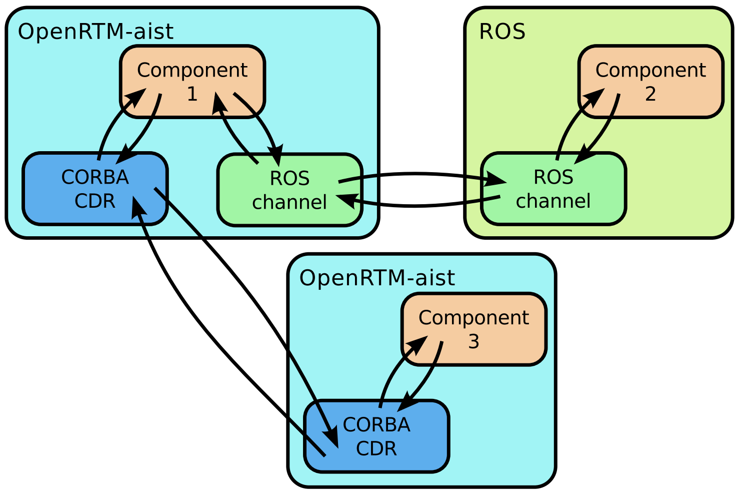

Fig. 5. The ROS transport provided by OpenRTM-aist allows a single component to communicate freely with both other OpenRTM-aist components, using the native OpenRTM-aist CORBA-based transport, and ROS components, using ROS's channel-based transport. The ROS transport can even be used between OpenRTM-aist components, if so desired.

execution time to `ros::spin()`, the ROS function responsible for calling callbacks when data arrives. This allows data reception to be handled out-of-band from the component execution.

Service port ROS services are callback-based. Each service in a ROS component exposes a single callback. The ROS transport for OpenRTM-aist works in the same way. The developer provides a callback functor object. The port uses a parallel thread to execute the ROS functions that handle receiving service requests, and ensures that the callback functor is called.

Request port The request port does not use a parallel thread. As with the native OpenRTM-aist request port, requests are blocking. The implementation is very simple. It calls the standard ROS function for calling a service on the ROS transport object contained within the port, blocking until a reply is received.

4 Discussion

The use of a native transport to interact with another framework brings many benefits over writing a component wrapper:

– A new wrapper must be written for each message type to be exported from the secondary framework to the primary framework. The ROS transport

avoids this by allowing components interested in data from the secondary framework to access it directly. This is a more natural approach for developers.
- A wrapper is often only effective in one direction – both the ORCA2 and MARIE wrappers are effective at bringing behaviour from the secondary frameworks to the primary, but are difficult at best to use in the opposite direction. This transport allows the benefits to pass both ways. For example, ROS gains access to a framework it can use for hard real-time parts of the robot, while OpenRTM-aist gains a persistent channel-based transport.
- A wrapper only allows us to gain the behaviour contained in components. The transport approach means that we also gain the benefits of the secondary framework's transport itself. We are not limited to using this transport to talk to the secondary framework. We can also use it between components of the primary framework as an alternative to the existing transports.
- The transport is as efficient as working directly in the secondary framework. No translation of data or reading-and-re-sending is required.

There are specific benefits both to OpenRTM-aist and to ROS of this particular transport.

- OpenRTM-aist gains a persistent channel-based transport. This has a variety of applications in robotics.
- OpenRTM-aist gains access to the huge library of functional software provided by and for ROS.
- ROS gains access to components that can execute in hard real-time.

An interesting application of native transports for framework interaction is in creating layered systems. A typical layered architecture features interaction points between the layers of some kind. With native transports, it is possible to utilise one architecture per layer, using the best architecture for the job. For example, the low-level control layer could be implemented in OpenRTM-aist, with its strong life-cycle control and hard real-time support, while a higher level for planning could utilise ROS, with its greater degree of flexibility and large library of perception and planning components. It is the author's opinion that this method of implementing layered systems has promise in bringing greater flexibility to how such systems are implemented.

One thought that may occur is that it is tempting to design one component to act as the gateway between the two frameworks, repeating the situation we have with wrappers. This may be the case when the data types between the two frameworks do not match. Figure 3 certainly seems to imply this approach. However, it is important to remember that the transport can be used in any component, and so the more likely scenario is that in which the appropriate transport is used on a connection-by-connection basis. The ease of using either transport facilitates this.

The specific implementation presented in this paper is not perfect. It has two flaws that limit its usefulness – the lack of commonality in data type representation and incompatible introspection interfaces.

OpenRTM-aist and ROS use different and incompatible serialisation schemes in their transports. OpenRTM-aist uses CORBA CDR, with the OMG IDL language for data description, while ROS uses a home-grown scheme based on generated serialisation functions and its own IDL for data description. This means that any data types that must be sent over both transports must be specified twice, once in each IDL.

OpenRTM-aist and ROS also use different interfaces for introspection. While a ROS component network is relatively static once running, in OpenRTM-aist an introspection tool is necessary to set up the component network at run-time. Because the OpenRTM-aist introspection tools are not aware of ROS nodes or channels, it is not possible to use them to directly connect to ROS channels at run-time. Instead, ROS channel connections must be fixed prior to run-time. This is less than optimal, from the point of view of OpenRTM-aist.

This integrated approach to inter-operation is rapidly spreading. The YARP framework [6] has recently gained experimental ROS protocol support, although it is incomplete at the present time. Differing component models do not prevent it being used; all that matters is that the data is available at each end of a connection when necessary. The local framework is responsible for providing it to components with the correct semantics after transport is complete.

4.1 Attaining the ideal

The two disadvantages mentioned in the previous section are not insurmountable. Creating the ideal situation requires flexible serialisation systems and common introspection interfaces.

Flexible serialisation is necessary to allow a transport to choose its serialisation method. For example, if OpenRTM-aist could choose its serialisation scheme, it would be able to transmit data types specified in ROS IDL or CORBA IDL. This would remove the need to specify data types twice. ROS and OROCOS 2 [3] both feature flexible type systems that can handle any serialisation method the developer chooses to use, while OpenRTM-aist is limited to just one. This is a problem with OpenRTM-aist.

Standard introspection systems is a more wide-spread issue. OpenRTM-aist, ROS and the aforementioned OROCOS 2 all feature introspection. They also all implement it differently, with different capabilities. If the introspection of transports were at least common, it would be easier to create a tool that could manage the connections between components from multiple systems.

5 Conclusions

Previously, creating a wrapper was the most common approach to allowing interaction between robot software frameworks for use in a single robot system. This approach has several limitations. It requires considerable developer effort to create and maintain a wrapper, it introduces inefficiencies into the system, and multiple wrappers are typically required to adapt all the desired messages of the secondary framework to the primary framework.

A better approach is to separate the transport from the framework, and allow multiple transports to be used. This can then be exploited by utilising transports from other frameworks, and so allow a framework to speak the language of other frameworks and communicate with them. This approach is more flexible than wrappers, requires less effort, and is more efficient.

We have developed such a transport for OpenRTM-aist. It provides native communication with the ROS framework, allowing components for either framework to communicate seamlessly with components for the other. The new transport brings the benefits of ROS's channel-based communication to OpenRTM-aist, and OpenRTM-aist's hard real-time support to ROS. The two frameworks can be utilised in the appropriate parts of a single robot system.

Acknowledgements

This research was partially supported by the Grant-in-Aid for the Developing Intelligent Technology for Next-Generation Robots Project (FY2007-2011) of the New Energy and Industrial Technology Development Organization of Japan.

References

1. Ando, N., Suehiro, T., Kotoku, T.: A software platform for component based rt-system development: Openrtm-aist. In: SIMPAR '08: Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots. pp. 87–98. Springer-Verlag, Berlin, Heidelberg (2008)
2. Biggs, G., Makarenko, A., Brooks, A., Kaupp, T., Moser, M.: GearBox: Truly reusable robot software (Poster). In: Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on (September 2008)
3. Bruyninckx, H.: Open robot control software: the OROCOS project. In: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on. vol. 3, pp. 2523 – 2528 vol.3 (2001)
4. Collett, T., MacDonald, B., Gerkey, B.: Player 2.0: Toward a practical robot programming framework. In: Proceedings of the Australasian Conference on Robotics and Automation. University of New South Wales, Sydney, Australia (December 5–7 2005)
5. Cote, C., Letourneau, D., Michaud, F., Valin, J.M., Brosseau, Y., Raievsky, C., Lemay, M., Tran, V.: Code reusability tools for programming mobile robots. In: Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. vol. 2, pp. 1820–1825 (2004)
6. Fitzpatrick, P., Metta, G., Natale, L.: Towards long-lived robot genes. Robotics and Autonomous Systems 56(1), 29 – 45 (2008), `http://www.sciencedirect.com/science/article/B6V16-4PT296V-1/2/4009fd97b8597d84d20457d2fc7d9db0`
7. ROS Wiki. `http://www.ros.org` (2010)
8. The Robotic Technology Component Specification - Final Adopted Specification. `http://www.omg.org/technology/documents/spec_catalog.htm` (2010)
9. RTSystemEditor. `http://www.openrtm.org/OpenRTM-aist/html-en/Documents2FRTSystemEditor.html` (2010)