YARP: Yet Another Robot Platform

Giorgio Metta[*] **Paul Fitzpatrick**[**] **Lorenzo Natale**[*]

[*]LIRA-Lab, DIST, University of Genova
Genova, Italy

[**]MIT CSAIL
Cambridge, Massachusetts, USA

Abstract

We describe YARP, Yet Another Robot Platform, an open-source project that encapsulates lessons from our experience in building humanoid robots. The goal of YARP is to minimize the effort devoted to infrastructure-level software development by facilitating code reuse, modularity and so maximize research-level development and collaboration. Humanoid robotics is a "bleeding edge" field of research, with constant flux in sensors, actuators, and processors. Code reuse and maintenance is therefore a significant challenge. We describe the main problems we faced and the solutions we adopted. In short, the main features of YARP include support for inter-process communication, image processing as well as a class hierarchy to ease code reuse across different hardware platforms. YARP is currently used and tested on Windows, Linux and QNX6 which are common operating systems used in robotics.

1. Introduction

YARP is written by and for researchers in humanoid robotics, who find themselves with a complicated pile of hardware to control with an equally complicated pile of software. Achieving visual, auditory, and tactile perception while performing elaborate motor control in real-time requires a lot of processor cycles. The only practical way to get those cycles at the moment is to have a cluster of computers. Every year the capabilities of an individual machine grows, but so also do our demands – humanoid robots stretch the limits of current technology, and are likely to do so for the foreseeable future. Moreover, software easily becomes entangled with the hardware on which it runs and the devices that it controls. This limits modularity and code reuse which, in turn, complicates software development and maintainability. In the last few years we have been developing a software platform to ease these tasks and improve the software quality on our robot platforms. We want to reduce the effort devoted to infrastructure-level programming to increase the time spent doing research-level programming. At the same time, we would like to have stable robot platforms to work with. Today YARP is a platform for long-term software development for applications that are real-time, computation-intensive, and involve interfacing with diverse and changing hardware. It is successfully used on several platforms [1, 2, 3, 4, 5, 6, 7] in our research laboratories (see Table 1).

Robot	Laboratory	size	OS
Babybot	LIRA-Lab	13	Win/QNX6
Eurobot	LIRA-Lab	11	Win/QNX6
RobotCub	LIRA-Lab	3	Win
Obrero	MIT-CSAIL	4	Linux/OSX
Mertz	MIT-CSAIL	4	Linux
Domo	MIT-CSAIL	6	Linux
COG	MIT-AILab	30	QNX4/Linux
Kismet	MIT-AILab	12	Linux/Win/QNX4

Table 1: Robots using YARP, and the number of computers they use (this count does not include processors such as DSPs, often used for low-level motor control).

We begin the paper by summarizing the lessons we have learned over the years while working on various robots, some of which are software engineering commonplaces and some of which are more specific to long-term robotic research. The bulk of the paper discusses the communication model supported by YARP. We then briefly mention other components of the library, in particular image processing and device drivers.

2. Motivation

Let us now introduce YARP by describing the high-level lessons we have learned and applied within it.

One processor is never enough. Designing a robot control system as a set of processes running on a set of computers is a good way to work. It minimizes time spent wrestling with code optimization, rewriting other people's code, and maximizes time spent actually doing research. The heart of YARP is a communications mechanism to make writing and running such processes as easy as possible. Even where mobility is required this is not a limiting factor if tethers or wireless communication are acceptable.

Modularity. Code is better maintained and reused if it is organized in small processes, each one performing a simple task. In a cluster of computers some processes are bound to specific machines (usually when they require a particular hardware device), but most of the time they can run on any of the available computers. With YARP it is easy to write processes that are location independent and that can run on different machines without code changes. This allows us to move processes across the cluster at runtime to redistribute the computational load on the CPUs or to recover from a hardware failure. YARP does not contain any means of automatically allocating processes as in some approaches like GRID [8]. We deliberately assign this task to the developer. The rationale is that: i) the link between hardware and corresponding control software is subject to constraints understood by the developer but cumbersome to encode, particularly in a continually-changing research environment, and ii) in an heterogeneous network of processors, faster processors might need to be allocated differently from slower processors. The final behavior is that of a sort of "soft real-time" parallel computation cluster without the more demanding requirements of a real-time operating system.

Minimal interference. As long as enough resources are available, the addition of new components should minimally interfere with existing processes. This is important, since often the actual performance of a robot controller depends on the timing of various signals. While this is not strictly guaranteed by the YARP infrastructure, the problem is in practice alleviated computationally by allowing the inclusion of more processors to the network, and from the communication point of view by the buffer policy (see Section 5.).

Stopping hurts. It is a commonplace that human cycles are much, much more expensive than machine cycles. In robotics, it turns out that the human cost of stopping and restarting a process can be very high. For example, that process may interface with some custom hardware which requires a physical reset. That reset many need to be carefully ordered with respect to when the process is stopped and started. There may be other dependent processes that need to be restarted in turn, and other dependent hardware. These ordering constraints are time-consuming to satisfy. YARP does its part to minimize dependencies between processes. Communication channels between processes can come and go without process restarts. A process that is killed or dies unexpectedly does not require processes to which it connects to be restarted. This also simplifies cooperation between people, as it minimizes the need to synchronize development on different parts of the system.

Humility helps. Over time, software for a sophisticated robot needs to aggregate code written by many different people in many different contexts. Doubtless that code will have dependencies on various communication, image processing, and other libraries. Even the operating system on which the software is developed can pose similar constraints. This is especially true with code that relies heavily on the services offered by the operating system (such as communication, scheduling, synchronization primitives, and device driver interfaces). Any component that tries to place itself "in control" and has strong constraints on what dependencies are permissible will not be tolerated for long. It certainly cannot co-exist with another component with the same assumption of "dominance". Although YARP offers support for communication, image processing, interfacing to hardware etc., it is written with an *open world* mindset. We do not assume it will be the only library used, and endeavor to be as friendly to other libraries as possible. YARP allows interconnecting many modules seamlessly without subscribing to any specific programming style, language interface, or demanding specifications as for instance in CORBA [9] or DCOM [10]. Such systems, although far more powerful than YARP, require a much tighter link between the general algorithmic code and the communication layer. We have taken a more lightweight approach: YARP is a plain library linked to user-level code that can be used directly just by instantiating appropriate classes. Finally, other programming languages can access YARP as well, provided they can link and call C++ code. We have successfully used YARP from within Matlab and L [11].

Exploit diversity. Different operating systems offer different features. Sometimes it is easier to write code to perform a given task on one OS as opposed to another. This can happen for example if device drivers for a given board are provided only on a specific platform or if an algorithm is available open source on another. We decided to reduce the dependencies with the operating system. For this we use ACE [12], an open source library providing a framework for concurrent programming across a very wide range of operating systems. YARP inherits the portability of ACE and has indeed been used and tested on Windows, Linux and QNX 6.

YARP's core communication model was the survivor from an early humanoid robot (called Kismet) controlled by a set of Motorola 68332 processors, an Apple Mac, and a loose network of PCs running QNX, Linux, and Microsoft Windows. Communication was a hodge-podge of dual-port RAM, QNX message passing, CORBA, and raw sockets. At one point, three incompatible communication protocols layered over QNX message passing were in use simultaneously. This variety was a consequence of organic growth, as developers added new modules to the robot. YARP began as one of the communication protocols built on QNX message passing. A key, defining, feature of YARP was that it was *broad-minded*: it was implemented in the form of a library which placed minimal constraints on user code; communication resources did not need to be allocated at any particular time or place in a program; reading messages could be blocking, polling, or callback based, etc. This meant it could be easily added without disturbing existing code, and communication could be moved across to the new protocol piece by piece.

3. Communication

Communication in YARP follows the *Observer* pattern [13]. The state of special *Port* objects can be delivered to any number of observers, in any number of processes distributed across any number of machines. YARP manages these connections in a way that insulates the observed from the observer and, just as importantly, insulates observers from each other. For example, if one observer reads a data source slowly and infrequently, this does not force other observers to slow down.

In YARP, a port is an active object managing multiple connections for a given unit of data either as input or output (see Figure 1). Each connection has a state that can be manipulated by external commands, which manage the connection or obtain state information from it. Ports can behave either as input or output. An input port can receive from multiple connections at different data rates "speaking" different protocols (e.g. TCP, UDP, multicast). An output port can send data to many destinations reading at different rates on different protocols. Service channels are also temporarily created to perform the handshaking between ports; in this case the protocol of choice is TCP for reliability. The use of several different protocol allows us to exploit their best characteristics:

- ▷ TCP: reliable, it can be used to guarantee the reception of a message;
- ▷ UDP: faster than TCP, used for point to point connections;
- ▷ multicast: used for creating one to many connections, efficient for distributing the same information to many targets;
- ▷ shared memory: employed for local connections (selected automatically whenever possible, without the need for programmer intervention);
- ▷ QNet: a fast and synchronous protocol used under the QNX real-time OS.

Ports can be connected either programmatically or at runtime. Communication is fully asynchronous and as such messages are not guaranteed to be delivered unless special provisions are made. The default behavior of YARP ports is targetted at dealing with recurrent messages, updated and sent often, where losing one message does not compromise the integrity of the system. This is a characteristic of sensor data such as images and sound, where it is far more important to keep up with the present than to process every bit received. A typical application is, for example, the acquisition of images, and delivery to many machines performing the processing in parallel. Slower processes might simply not use all the available frames in the stream of data and rather skip some of them. Details of the port API are reported in the next section (Section 4.). Message delivery can be guaranteed, but at a cost or introducing a subtle coupling between processes, as discussed later in this paper (see Section 5.).

Ports are located on the network by symbolic names which are managed by a name server. The name server maps symbolic names (strings) into the triplet composed of the IP address, port number, and interface name. This information is all that is required to establish socket communication between two endpoints. A description of the network topology is stored statically in the name server tables (a cluster might have multiple separate networks) and used to reply to registration or connection requests by the clients. The first operation each port must perform is the registration of its name with the name server. Registration is typically followed by connection to a peer of the same data type. When the user is done with the port, it can be stopped, unregistered, and eventually destroyed.

Ports can deal with any data type. For simple data types (i.e. not containing pointers) the port class is already equipped with the appropriate communication code.

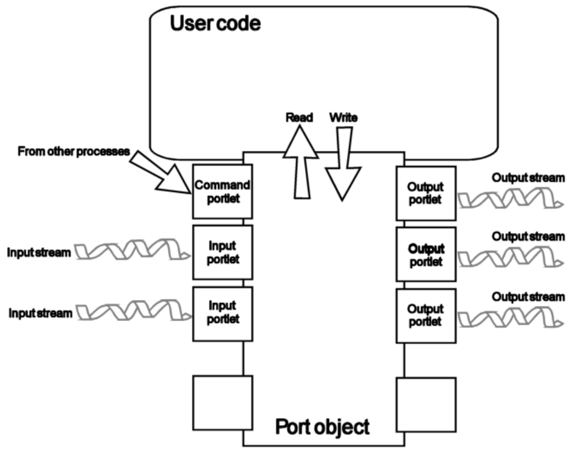

Figure 1: The port internal structure: in practice either input or output connections are used for a given instance of a port object.

Complex data types are dealt by specializing the port C++ template for the new complex data type and providing the serialization and deserialization functions. Serialization is done by providing lists of memory blocks, to minimize copies (crucial for bulky types such as images). Support for marshalling is not built into the library. Ports are implemented as C++ templates and specialized to the type of the data to be transmitted or received. This creates a very clean and consistent client interface.

4. Port API

For concreteness, we will show an example of the idiom used in YARP for communication. As mentioned in the previous section, the Port class is the key abstraction used. Ports are typically instantiated with a specific type. For example, if we wish to receive integers, we can create an input port at any point in the program and in any thread, as follows:

```
YARPInputPortOf<int> in_port;
in_port.Register ("/my_in_port");
```

This creates a port for receiving integers with the default buffering provided by the communication layer (see Section 5. for a discussion of alternatives). If we are in a heterogeneous network, it would be wiser to use `YARPInputPortOf<NetInt32>`, where `NetInt32` is a standard integer type that is the same size and byte order on all platforms. The next statement instructs the port to register with the name server with the arbitrary name "/my_in_port". A hypothetical sender should conversely create a port as in the following example:

```
YARPOutputPortOf<int> out_port;
out_port.Register ("/my_out_port");
```

This is an output port employing the default protocol (TCP, or shared memory whenever possible); alternatives can be easily requested. The protocol type is determined by the output port since the input port can receive any of the available protocols. Again, the port has to register with the name server by calling *Register*. As described earlier, the port is a template with the argument of the template being the type of the data being sent.

The next step is to wait for data from the input port and send data through the output port Waiting or polling for data can be done in several ways; here's one way:

```
if (in_port.Read()) {
    int datum = in_port.Content();
    cout << datum << endl;
}
```

This shows how to read from the port with a blocking *Read* and acquire the received data through *Content*. If the call to *Read* succeeds, then the object returned by any subsequent call to *Content* will be the received data, and is guaranteed not to change or be overwritten until the next call to *Read*. If new data is sent to the port in the meantime, the appropriate action will be taken based on the port buffering policy (see Section 5.). For example, the data may be stored in an alternate buffer and then queued up to become the *Content* after the next call to *Read*. On the sending side we will have something like:

```
out_port.Content() = 42;
out_port.Write ();
```

This fills the content (a simple integer in this case) by accessing the buffer through *Content* and sends it by calling *Write*. The use of *Content* is important to avoid unnecessary copies while still maintaining an abstraction barrier between the port and the user. We can now connect the two processes (one receiving, one sending) by, for example, using the YARP command-line utility *yarp-connect*:

```
yarp-connect /my_out_port /my_in_port
```

When done with the communication the user can detach the ports in a similar way. The ports are not destroyed by detaching them and in fact can be connected and disconnected freely. When done with the ports the user code can call *Unregister* to remove the ports from the name server, and finally destroy them by invocation of the C++ destructor (perhaps implicitly when exiting the port scope).

The *Write* method abstracts over a great deal of complexity. An output port may be connected to many input ports, all of which may read data at different rates. By default, when *Write* is called, a reference to the buffer is passed to every free output connection (to block and wait for all sends to finish before trying the next one, *FinishSend* can be called). The buffer will be retained until it is no longer needed by any output connection, and then given back to the port to be recycled.

This short example shows all the main features of the port classes including the strong typing of the communication channels, the independence of the connected processes, and the use of an external utility to command ports. Port creation, connection, and communication can occur in one part of a much larger program without, for example, having to place special initialization steps in some particular phase of start-up. This makes it very easy to add YARP-style communication incrementally to existing code.

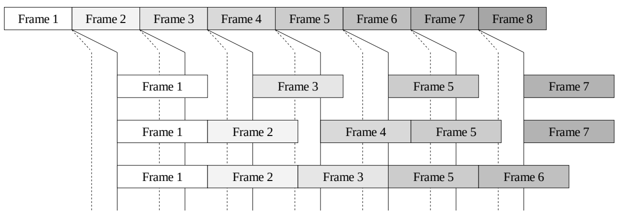

Figure 2: The top row represents an output port configured for *no-wait*; dashed and solid lines show (exaggerated) start and end times of sending an update to three observers (input ports), configured as *single-buffer*, *double-buffer*, and *triple-buffer* respectively. For the scenario shown, the processing time of the client is greater than that of the server.

5. Decoupling timing

A very useful feature of YARP is that "observers" (input ports) can be connected to an "observable" (output port) with minimal impact on existing observers. A "slow" observer, which takes time to process each update it received from the observable, does not force a "fast" observer of the same observable to slow down. To achieve this requires either buffering of messages for bursty sources, or simply dropping messages for observers that can't keep up. The second approach is the default behavior in YARP, since it is important to minimize latency.

Let us assume we have a "server" process which contains an observable (an output port), and a "client" process which contains a corresponding observer (an input port). The server process can update the observable in one of three ways:

▷ The default mechanism is ***no-wait***. When the server process calls the observable's update method (*Write*), then the current state of the observable is made available to be sent to every free observer, and the server can continue without delay. Free observers are ones not currently in the process of reading a previous state of the observable.

▷ An alternate mechanism is ***wait-after***. After the same steps as *no-wait* are taken, the server can choose to wait for all communication to cease before continuing (by calling *FinishSend*). This guarantees that all observers will be notified and free to receive the next update.

▷ The final mechanism is ***wait-before***. The server can choose to wait for all communication to cease before updating (by calling a blocking version of *Write*). This guarantees that all observers will be free, and the update will be sent to all of them. The difference between this and *wait-after* is that, if the processing time of the server (the time between updates) is greater than the time taken to send the update to all observers, then the server will never actually need to wait.

To insulate the server from the details of implementing all this, the state associated with an observable is made logically distinct from the observable itself, and once an update is requested (by a call to *Write*) the state becomes the property of the communication system, while the server is given a replacement object to prepare for the next update. The communication system manages a pool of such state objects which grows to whatever size is necessary based on the speed of the various observers. On the client side, there are some choices in how the observer behaves:

- ▷ ***triple-buffer*** behavior: an observer becomes free for another update immediately after having received one, before any processing is done by the client. If updates arrive faster than processing occurs, then updates will be lost from time to time (where "lost" means "never processed"), but the most recent update received will always be available to the client immediately when processing is completed.
- ▷ ***double-buffer*** behavior: same as above, but if an update is currently arriving, then no new content will be available to the client until the update arrives. This is good if it is better to minimize latency of non-dropped updates than to maximize throughput.
- ▷ ***single-buffer*** behavior: the arrival of updates is delayed until the client completes processing. No updates will ever be lost on the client side.

The default behavior for YARP is *no-wait* for the observable (server side) and *triple-buffer* for the observer (client side). This choice minimizes the time spent waiting for communication to occur by the server and the client, and permits updates to be lost (either by never sending them, or discarding them on the client side) if the client is not keeping up. This is generally a good choice for real-time performance (See Figure 2).

The default of *no-wait* on the server side is particularly important, since it minimizes coupling between observers of the same observable. If it is important that updates are never lost, then inevitably there will be coupling, since a slow client can then force the server to slow down the rate at which it serves all clients.

The default of *triple-buffer* on the client side insulates the server from the client's behavior by default. Even if the server is configured to wait, default clients will only delay the server with the time taken to communicate with them, and not the time they take to process the update. Clients which absolutely need a guarantee of zero update loss can choose *single-buffer* behavior.

6. Image processing

Support for visual processing is a mandatory requirement for a software library designed to be used in humanoid robotics. Efficiency is very important in real-time image processing, so we choose an approach which interfaces particularly well with popular optimized libraries, but which is still capable of good performance in their absence.

To help developers write efficient visual processing routines, Intel released the Image Processing Library (IPL). This library is optimized to provide high performance on machines which employ Intel processors, especially if equipped with MMX^{TM}technology. The IPL library is a

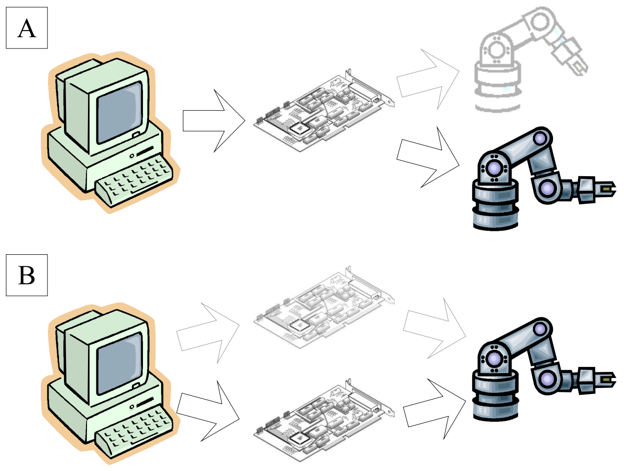

Figure 3: Changes in hardware make code reuse challenging. YARP makes a distinction between *proximate* devices, such as control boards and framegrabbers which are used to talk to *distal* devices such as arms or sensors. The same proximate device may be used to interface with different distal devices (A). Conversely, a given distal device may be interfaced with using a choice of proximate devices (B). Taking care to disentangle these two devices aids code reuse.

set of C functions which implement basic operations on images, from simple algebraic operations on pixels to color conversions and convolutions. The library consists of different modules optimized for different CPUs. For better performance at run-time the library automatically detects the CPU type and loads the module that is most suitable. Another advantage of using the IPL is that it is at the core of the OpenCV library [14] which provides sophisticated routines for image processing such as filtering, face tracking, optic flow, and much more.

Our basic image class has an internal structure that is compatible with the IPL library. This allows any user to take full advantage of the IPL and/or OpenCV libraries; if these libraries are not used, then a core set of functions are available through YARP. Furthermore, the image class can act as a *proxy* to image data stored in a foreign format. This is useful to prevent unnecessary copies when using other image processing libraries, or interfacing with image sources (e.g. framegrabbers) and sinks (e.g. a graphic display). YARP also provides support for transmitting images across the network.

7. Device drivers

A frequent problem encountered during development in robotics is that it is very hard to reuse code on different platforms. For example, two mechanically similar platforms may have different electronics – different frame grabbers, different control boards, etc (see Figure 3). In these situations it is not possible to reuse code written for one platform on the other unchanged. However something can be done to reduce the differences and localize them to specific components by minimizing the degree to which high level software modules are concerned with the low level details of the underlying hardware platform.

Another problem occurs when two identical boards are used on setups that are mechanically different. Experience shows that in these situations code reuse is very difficult. Consider for instance the example of two robotic arms controlled by identical boards. The calibration of the joints might be different if indexes are available in the encoders or if hardware limits are presents in the joints. Likewise, the procedure required to activate the amplifiers might differ in the two cases. These dissimilarities cannot be handled by different configuration files as they imply the execution of different routines.

The ensemble of these routines are grouped in an *adapter*. This class is in general responsible for implementing methods to correctly initialize and shut down the device, but it can implement other functionalities as well – it is the place where all the peculiarities of each particular piece of *distal* hardware (arms, sensors, etc.) are mapped onto the *proximal* device (control boards, framegrabbers, etc) used to interface with it. As such it collects all and the only routines specific to each hardware device.

Finally, the driver for the proximal device and the adapter for the distal device are aggregated together by a single class. The interface between higher level software modules and the hardware occurs through this class and is thus independent of the device driver or the actual hardware underneath. Code changes required to use different boards or mechanical devices are localized to the device driver and the adapter respectively.

8. Conclusions

To operate in natural, unengineered environments, we need perceptive robots. We hope that humanoid robots will ultimately be able to operate productively in such environments. That means that between now and when that happens, we can only expect the sensor density and computational burden on our robots to grow. Real-time operation under this burden is challenging enough, but we must also expect the hardware we work with to change continually. The YARP library has grown organically to face this challenge. Somewhat similar projects have evolved from other domains in robotics such as mobile navigation (Carmen/IPC [15]) and commercial/industrial robotics (Orocos [16]), and we expect that there will be further development as the *perceptual* component of robotics grows in importance.

Acknowledgements

YARP makes heavy use of, and is itself, F/OSS software – thank you world. This work was partially funded by DARPA DABT 63-00-C-10102, and by NTT under the NTT/MIT Collaboration Agreement, and by European Union grants RobotCub (IST-2004-004370) and ADAPT (IST-2001-371173).

References

[1] Lorenzo Natale. *Linking Action to Perception in a Humanoid Robot: A Developmental Approach to Grasping*. PhD thesis, DIST, University of Genoa, Italy, February 2005.

[2] C. Beltran-Gonzalez and G. Sandini. Visual attention priming based on crossmodal expectations. In *Accepted for publication at the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Edmonton, Alberta, Canada, August 2005.

[3] Eduardo Torres-Jara, Lorenzo Natale, and Paul Fitzpatrick. Tapping into touch, 2005. Accepted for publication at the Fifth International Workshop on Epigenetic Robotics.

[4] Lijin Aryananda and Jeff Weber. Mertz: A quest for a robust and scalable active vision humanoid head robot. In *Proceedings of the IEEE-RAS/RSJ International Conference on Humanoid Robots*, Los Angeles, CA, November 2004.

[5] Aaron Edsinger-Gonzales and Jeff Weber. Domo: A force sensing humanoid robot for manipulation research. In *Proc. of the IEEE International Conf. on Humanoid Robotics*, Los Angeles, November 2004.

[6] C. Breazeal and B. Scassellati. How to build robots that make friends and influence people. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Kyonjiu, Korea, 1999.

[7] R. A. Brooks, C. Breazeal, M. Marjanovic, and B. Scassellati. The Cog project: Building a humanoid robot. *Lecture Notes in Computer Science*, 1562:52–87, 1999.

[8] Internet Site. Grid computing research. http://www.grid.org.

[9] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2):46–55, February 1997.

[10] Internet Site. Com: Component object model technologies. http://www.microsoft.com/com/.

[11] R. A. Brooks. The behavior language; user's guide. Technical Report AIM-1227, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, April 1990.

[12] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, 2003.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[14] Gary Bradski. Open source computer vision library. In Gerard Medioni and Sing Kang, editors, *Emerging Topics in Computer Vision*, pages 521–582. Prentice Hall, 2004.

[15] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 3, pages 2436–2441, Las Vegas, NV, October 2003.

[16] Peter Soetens. The Orocos open realtime control services: Open RObot COntrol Software 0.20.0, 2005. Available online at http://people.mech.kuleuven.be/ psoetens/orocos/ doc/orocos-manual.html.