Design and implementation of modular software for programming mobile robots

Alessandro Farinelli, Giorgio Grisetti, Luca Iocchi

Dipartimento di Informatica e Sistemistica - Università di Roma "La Sapienza"

Via Salaria 113 00198 Roma Italy

E-mail: <lastname>@dis.uniroma1.it

Abstract: *This article describes a software development toolkit for programming mobile robots, that has been used on different platforms and for different robotic applications. We address design choices, implementation issues and results in the realization of our robot programming environment, that has been devised and built from many people since 1998. We believe that the proposed framework is extremely useful not only for experienced robotic software developers, but also for students approaching robotic research projects..*

Keywords*: Robotic Application Development.*

1. Introduction

Research on developing autonomous agents, and in particular mobile robots, has been carried out within the field of Artificial Intelligence and Robotics from many different perspectives and for several different kinds of applications, and the development of robotic applications is receiving increasing attention in many laboratories. Moreover, robotic competitions (e.g. AAAI contexts, RoboCup, etc.) have encouraged researchers to develop effective robotic systems with a predefined goal (e.g. playing soccer, searching victims in a disaster scenario, etc.). Moreover, mobile robots are also used for teaching purposes within computer science laboratories and often students are required to work and develop robotic applications on them[1]. This increasing population of robots in the research laboratories and the consequent

need for developing robotic applications have started a process of design and implementation of robotic software, that aims at having a design methodology and a software engineering approach in the development of such applications. Furthermore, companies producing and selling mobile robots make available to their users development libraries and software tools for building and debugging robotic applications (e.g., Saphira/ARIA for Pioneer robots (Konolige et al., 1997), OPEN-R SDK for Sony AIBO[2], etc.). These tools are obviously platform dependent and thus they cannot easily be used for building multi-platform robotic systems. Moreover, they usually lack some features that are required from a general purpose robot development toolkit. For instance, the OPEN-R SDK completely lacks facilities for remote monitoring the behavior of the robot. It just supports wireless network communication among processes and all the remote information exchange must be explicitly

---

[1] e.g. CMRoboBits Course at CMU
     www.andrew.cmu.edu/course/15-491/

[2] Open-r SDK, www.jp.aibo.com/openr/

coded. On the contrary, the Saphira/ARIA environment, although it is specifically implemented for the Pioneer robots, has several facilities for building robotic applications and debugging them also by using a simulator and allowing for a graphical display of the robot status. Finally, a number of open source multi-platform robotic development environments have been realized. For example, OROCOS (Open RObot COntrol Software)[3] is an European project that has recently started with the objective of realizing a framework for developing robot control software under Real Time Linux. This project has many general goals, like independence to architectures used for connecting the components together, to robot platforms, to robotic devices, to computer platforms. The OROCOS project has a long time target and it is currently under development. Player/Stage (Gerkey et al., 2003) is also a general framework for controlling a robotic system. Player supports a wide range of devices, algorithms and viewers, that can be tested through Stage, a simulator able to work on complex multi robot scenarios. Each of these devices can be either a server or a client, allowing for a great flexibility in spreading the computation on different machines. However, Player/Stage provides only limited support for high level specification of user-defined modules and their interaction. CARMEN[4] comprises a set of independent utilities, that communicate with each other through UNIX inter process communication. This framework has been used for implementing a set of interesting algorithms, but it is mainly suited with the low level activities of the robots (such as navigation and exploration). MARIE[5] is a development tool and an integration environment for mobile robot applications. It is well suited for fast and easy connection of high level modules among thems and with hardware components. However, MARIE does not provide infrastructures for dynamic information sharing and for remote inspection of the application. Also the works in (Utz et al., 2002; Wang et al., 2001) are focused on proposing robot middle-ware that are not specific to a given platform or to a particular application domain. In particular, the system presented in (Wang et al., 2001) is explicitly focused on the realization of soccer applications, while in (Utz et al., 2002) mostly low level interface issues are addressed. In this paper we describe

a Robot Development Toolkit (RDK) for modular programming of mobile robots. We will use the term task to denote a basic functionality of the robot and module to refer to its software implementation. The toolkit we have realized includes a middle-ware that implements all the basic functionalities for the development of a typical robotic application, a set of modules implementing the basic capabilities of the robot, and a set of tools that are useful for developing, monitoring and debugging the entire application. In particular, the middle-ware implements an infrastructure for: task management, interfacing with the robot hardware, representation of the status of the robot, remote monitoring and debugging. The main difference with other approches discussed above is the support that our middle-ware provides for task development, in terms of hardware abstraction, dynamic information sharing among modules and remote inspection that are useful for efficient development of robotic applications. Our development toolkit is currently named SPQR-RDK. We have used our framework for developing different kinds of robotic applications: i) RoboCup soccer (Kitano et al., 1998) ii) RoboCup Rescue (Tadokoro et al., 2000) iii) RoboCare (Bahadori et al., 1995) - a project for developing a multi robot system for assistance of elderly people in a health care house. The development of these applications has given us a real testbed for evaluating the proposed RDK and, by a comparison with the development of similar applications by using a different development environment (in particular, we refer to the robotic soccer application with Sony AIBO robots by using OPEN-R SDK), we have experimented the effectiveness of our toolkit.

2. Design Choices

During the development of our RDK, we have identified a set of fundamental functionalities and a set of software requirements needed for our framework. As our applications have been developed through the years by different people which were able to work at the related projects only for a limited period of time, modularity and re-usability appear to be the main issues to address: the proper division of the code in independent modules exchanging data inside a clear framework ensures to have a coherent software generation, resulting in high modularity and re-usable code. Efficiency is also a

---

[3] Orocos project, www.orocos.org

[4] Carmen project, www-2.cs.cmu.edu/~carmen/

[5] MARIE project, marie.sourceforge.net

primary requirement: the middle-ware needed for running the modules must have a minimum overhead with respect to the entire application. Moreover, the hardware computational capabilities must always be considered, posing strict constraints on the implementation choices for our middle-ware; therefore most of the design choices that we have done (e.g. language, operating system, shared memory for information exchange) are motivated by this requirement. As for functionalities we have identified three main issues to be addressed: i) *Remote Inspection Capability,* ii) *Information Sharing,* iii) *Common Robot Hardware Interface*. Remote Inspection is a fundamental functionality for every robotic application and is extremely important for effective development of a complex system. The Remote Inspection mechanism, should allow the developers to use a general mechanism for remotely inspecting the internal status of each component of the application and to dynamically chose what to monitor and when, with limited network bandwidth and minimum computational overhead with respect to the normal execution of the robotic application. Another important problem that we have faced during our past developments has been the exchange of data among components. The use of shared memory without any data access policy, is not satisfactory because the management of all the shared data in the program can become very complex. Similarly, the use of message exchanging typically arises the same problems and may also affect modularity of the system, when a module is implemented by including the details of other interacting modules. Therefore, an important functionality for the RDK is an Information Sharing mechanism providing a uniform interface and a policy for dynamic data sharing among modules. Finally, when dealing with several different types of mobile bases and sensing devices the independence of the application from the low level details of platforms and devices becomes an important issue. Hence, the development of a Robot Hardware Interface has been identified as another important functionality: a uniform interface has to be defined between robot devices and user modules, and hardware configuration is described in a configuration file.
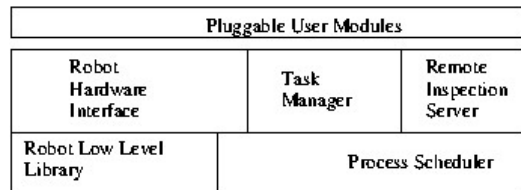

Fig. 1 Middle-ware Architecture Layered View

3. Software Architecture and Implementation of the Middle-Ware

The RDK we are presenting in this article is based on a middle-ware that provides the basic functionalities for the development of robotic applications. This middle-ware is composed by a minimum set of modules, common to all the applications that can be developed within this framework. The middle-ware is made up by the following modules (as shown in Figure 1): i) The Robot Hardware Interface is a library that defines an abstraction layer on the specific robot hardware, providing a common interface to the higher level modules. ii) The Task Manager is a library that defines a template for all the user modules and provides both a set of services for dynamically loading the user modules in the application and a mechanism for data exchange among them. iii) The Remote Inspection Server is a library that allows for remotely monitoring the robot activities, by implementing a publish/subscribe mechanism for the data produced by the running modules that can be selectively gathered at run-time.

A Robot Hardware Interface
The Robot Hardware Interface (RHI) module encapsulates the functionalities for accessing the mobile base and the on board devices and provides an abstraction for: i) mobile robot kinematics, by implementing the functions for reading odometry and for controlling the motion that are specific to a mobile platform kinematics model (for example, distinguishing holonomic[6] mobile bases from unicycle-like[7] ones); ii) mobile base connection, by providing a standard way to access the mobile base and its specific control functions. Each mobile base is generally equipped with various kinds of sensors and actuators like sonar rings, laser scanners, cameras, kickers (in the case of our soccer robots) and so on, that are generically defined as Device.

---

[6] An holonomic robot has three degrees of freedom in its motion.
[7] A unicycle robot has translational and rotational velocity bounded by a given kinematic law.

These devices are connected to the robot and grouped in a set of hierarchical classes for convenience (see Figure 2). In the following we provide a short description of the class hierarchy: i) Robot: is the base class of the hierarchy, that defines primitives for getting/setting the absolute robot position, for enabling/disabling the motors, for synchronizing the internal variables with the underlying hardware, etc. A Robot may have one or more connected Devices. ii) HoloRobot and UnicycleLikeRobot: define the interface for controlling a generic holonomic or unicycle-like robot, by defining the interface of the commands for setting/getting the rotational and translational speeds of the mobile base. Their subclasses redefine control functions for specific kinds of robots. iii) Device: is an abstraction for a device which is connected to the mobile base. The sensor devices produce information that are exploited by user modules (e.g. images form a camera), while the actuator devices export commands that are used by user modules for executing some action (e.g. moving the camera motor). Note that the mobile platform is not explicitly modeled as a device, since it is integrated with the robot and thus it is considered in the specific robot class by using the specific control library. Each specific robot or device driver class is compiled into a different shared object, that can be loaded by the application at run time. This allows great flexibility in switching among mobile bases or devices, which is useful for developing the single application subsystems. Both devices and robot drivers can be replaced by simulators or players of real data streams recorded before, allowing for off-line application development and debugging.
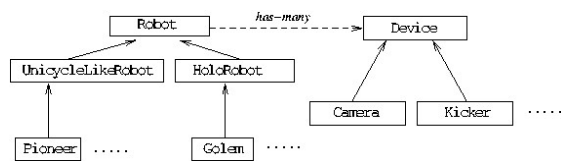

Fig. 2 Robot Hardware Interface Class Hierarchy

B. Task Manager

 The Task Manager has been designed in order to allow the user to dynamically load his/her modules, to specify their execution features (i.e. execution period, scheduling policy, priority and so on) and to export the information to be shared among them. A user module is modeled within the Task Manager as a single thread. Although there are several sophisticated C++ thread libraries

available, like the Boost thread library[8], as well as some implementation of process schedulers that are used in building mobile robotic applications (Piaggio et al., 2000), since we need only simple features, we chose to implement a simple C++ wrapper for the Linux threads, instead of using external libraries. Basically the wrapper defines the following kind of tasks, that differs each other for the scheduling policy: Asynchronous Tasks: are a classical threads, whose execution policy is delegated to the Linux scheduler; it is useful for implementing modules that do not interfere with the executional flow of other modules, like a network monitor. Periodic Tasks: are asynchronous threads, re-spawned at fixed time intervals; they are used for tasks that require periodic execution, like vision, localization, etc.
Serial Tasks: are tasks whose execution is serialized with respect to other serial tasks in the same group; since all the serial tasks in the group do not preempt each other, they are used for modeling operations that have a strict time or data dependence. Another important feature of the Task Manager is to allow for the exchange of information among modules. When modules need to directly exchange information each other, the simplest solution is to couple those modules. For example, if a module a needs the information provided by another module b, it is an obvious choice to allow a and b to know each other since they have to interact. However, this simple solution has the effect of limiting the software modularity since a modification in the implementation of b may need a modification of a; Therefore, besides the mechanism of directly coupling two modules, the Task Manager offers another possibility to exchange information, by abstracting on the type of information. In fact, if a module needs data provided by some other module, it only needs to know where to read such data and when the data are available. On the other hand, a module that produces information can easily declare the kind of such information without knowing which user module will use it. This solution grants a complete independence among modules sharing data and it is possible to substitute a module with another, by only ensuring that the two modules produce the same kind of data. This mechanism has been used for sharing information among user modules, as well as between a device and a user module. More in depth, this information exchange mechanism makes use of a Shared

---

8   The boost libraries, www.boost.org

Information Register (SIR) which is a sub-component of the Task Manager. The information producers submit the data to share to the SIR, while the information consumers can decide when to retrieve the data, without knowing who published the information. The published data can be written only from the module that produces it, and read from all the other modules. As an example of use of this mechanism, suppose we want to develop a localization module for a robot equipped with a laser range finder. In this case we have to develop a module that reads data from the range finder and produces data associated to a known label (e.g. RangePoints). In order to read the range finder output, the localization module only has to get from the Task Manager the reference of the information named RangePoints. If we further want to test the developed localization method to work with points extracted from a camera image, rather than provided by a range finder, we just have to define a vision module that produces the same kind of information, with the same label, and thus we can replace the laser device with this vision module, without affecting the localization module. Observe that, since no one of the three involved modules (localization, vision, scanner reader) knows the others, in the proposed implementation there is no coupling among modules, thus data flow can be dynamically activated at run-time.
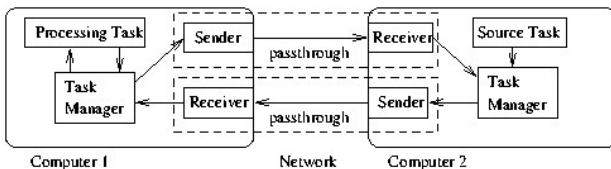

Fig. 3 Work of the Pass-through Task

Another aspect that must be considered is that the reliance on the shared memory constrains the user to execute all of the communicating tasks on a single machine. If the computational requirements of the modules are high, then such a constraint can be limiting. In order to spread the execution of heavy tasks on more machines, we have defined a pass-through mechanism that is similar to the one used in Player/Stage (Gerkey et al., 2003) and simply acts as a bridge in the network: on one side it is seen as a sink, on the other side it is seen as a source see Figure 3. All of the predefined data-types that can be used in our RDK are serialaizable, and in fact there is no great programming effort in using this kind of mechanism.

C. Remote Inspection

The problem of remote monitoring is very important in developing a robotic application and its realization may not be simple when considering that: i) the wireless network connection between a robot and its monitoring host is usually very noisy; ii) network latency must not affect the robotic application; iii) the information to inspect must be selected at run time, i.e. during normal operation of the robot, and when no information are requested there should be no overhead in the robotic application; iv) it should be avoided to differentiate a debug version from a release one. In order to devise such functionalities, the first design choice is the network transmission protocol. In fact, while wireless network devices are less reliable than the wired ones, remote monitoring requires to collect all the data transmitted by the robot (and in the correct order). Therefore, a reliable protocol, like TCP, must be used, since an unreliable one, like UDP, does not guarantee the retransmission of lost packets. However, since a reliable protocol grants packet delivery by retransmitting lost packets and this may be very frequent with wireless networks in noisy environments, the amount of data to be transmitted must be minimized in order to avoid network overhead. The second implementation choice, that has been made in order to avoid locks and minimize delays to the robotic application due to network latency, has been to perform this transmission in a separate thread with respect to the robotic application. The third design choice has been a publish/subscribe mechanism for allowing the monitoring clients to subscribe for receiving specific information published by the user modules. In this way the network bandwidth is determined only by the amount of information actually requested by the connected clients. The Remote Inspection Server (RIS) defined in our middle-ware exports facilities for the user modules requiring to publish information that can be monitored by a remote client and manages the connection with the clients. The information update is performed in two steps: the first one is refresh, where the RIS copies in a local buffer the information produced by user modules which have been requested by at least one client; and the second step is transmission, in which the RIS performs the transmission of the buffered information to the clients. The refresh step, which has to interact with other modules in the same machine, typically takes a very short time; while the transmission step, which has to interact with a remote host, can take a long time and thus it runs as a separate thread. In this way network latency only affects the communication of the information to the

remote host and not the efficiency of the publishing module on the robot. During the normal operation, when it is not needed to monitor the robot behavior in such a deep way, and clients do not request information to the robot, there is no overhead at all, since the Remote Inspection Server detects this situation and avoids useless computation. Moreover, in our middle-ware we have defined several data types that can be useful in robotics, like points, set of points, trajectories (i.e. sequences of points), bitmap images, vectorial images, etc., and other types may be easily defined. Each data type is identified by means of a key mechanism that allows also for serialization, and for each of these data types a graphical viewer is defined in the remote graphical client application in order to display the status of the robots during their missions. This allows developing remote monitoring graphical tools with a very small overhead. Observe also that the publish/subscribe mechanism that we have implemented allows for simultaneously connecting more clients to a robot. This is very useful in order to monitor different behaviors of a robot application with appropriate client tools. As an example, consider a situation in which we want to debug a navigating robot equipped with a camera: we want to be able to inspect both navigation and vision processing. With the RIS mechanism it is not needed to develop a debug tool that is specific for this task combination, but it is possible to use two clients connected to the robot: one that analyzes the camera image processing, and the other that controls the robot motion. Although there exist efficient alternatives to the proposed approach that provides for interoperability among modules, such as CORBA implementations, that can be suitable for robotic applications, we have chosen a simple remote inspection mechanism in order to implement a small subset of specific features. In fact, the Remote Inspection Server that we have developed has been specifically devised for a robotic application in order to provide a minimal set of specific facilities, instead of a wide range of general ones that has to be specialized in order to become useful.

4. Applications

The toolkit described in this article is designed to be a useful programming tool to develop applications for autonomous mobile robots. In this section we present

and discuss some specific applications developed using the described toolkit. In particular, we focus on two main domains: 1) Soccer Mid-size Robots (Nardi, 1999); 2) Exploration and Mapping in rescue environments (Bahadhori et al., 2005). For each of those issues we highlight the desired goals, and the results obtained.

Soccer Robots: The goal of the soccer robotic application is to build a team of autonomous robotic soccer players for the RoboCup middle size league competition (Nardi, 1999). Our middle size team comprised four different platforms: a customized ActivMedia Pioneer 1 platform, a customized ActivMedia Pioneer 2, a Golem Robot and a robot completely developed within our group. The same code runs on all the robotic platforms, and a configuration file is used in order to load different libraries and set different parameters for each robotic base. Figure 4 reports the pluggable modules involved in the robotic soccer application and the data flow among them. All modules have been realized using our framework and the SIR is used for data exchange.
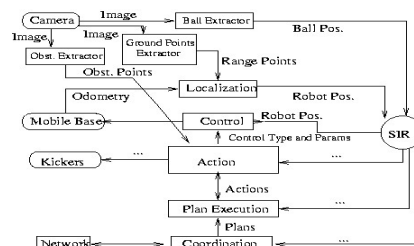
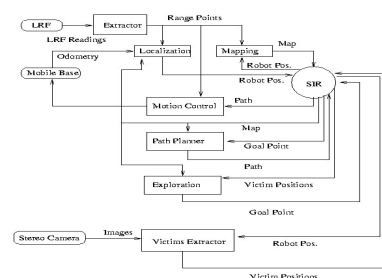Fig. 4. Robotic soccer application

Fig. 5. Expl. and Mapping in Rescue Environments

Exploration and Mapping in Rescue Environments: We are currently involved in several projects regarding rescue robots (RoboCup Rescue, SRSOES[9]). The main aim of these projects is to develop robotic platforms with

---

[9]   Simulation and Robotic Systems for Operation in emergency   Scenario www.dis.uniroma1.it/~multirob/SROES.html

high level capabilities to assist human rescuers during emergency operations. Figure 5 reports the modules realized for the rescue applications. In particular, we added a mapping module to build the map of the environment (Bahadori et al., 2004), a victim detection module to detect and locate victims (Bahadori et al., 2004) and an exploration module which is in charge of finding a strategy to explore the environemnt (Calisi et al., 2005). Moreover the motion control module has been substituted to negotiate narrow passage and deal with unstructured obstacles. The results of the use of our robotic development toolkit in the described applications has been a rapid, modular and effective development by many people that have interacted each other with minimum overhead and high productivity. Moreover, the possibility of reusing a large part of what we have developed so far in future projects without fundamental changes, provides an evidence that the design choices made in the development of our toolkit were reasonable and adequate for these kinds of applications.

5. Conclusions

In this article we have presented a toolkit (SPQR-RDK) for developing modular multiplatform robotic applications, that has been designed for providing modularity, effectiveness and efficiency. Such a framework has been tested in different contexts: robotic soccer, robot navigation and mapping, and it is currently used in some other robotic projects. This RDK allows a group of programmers to design and implement the modules composing a multi-platform robotic application, having both remote control and remote debugging capabilities, with a very small effort, by using a software engineering approach and by focusing on the semantics of the information exchanged among the modules. The main use of our toolkit is for people (mainly students) that want to develop a solution for a single topic or for a specific application (e.g. localization in an office-like environment, path planning with moving obstacles, multi robot coordination in a soccer domain, etc.), by using available modules for all the other capabilities of the robot. Our RDK provides these programmers with an easy methodological tool for implementing the robotic application and also it allows for easily evaluating the specific application developed under different environment conditions and in comparison with different solutions implemented by other people. The presented RDK has several advantages with respect to other robotic development libraries distributed by robot producing companies (e.g. Saphira/ARIA (Konolige et al., 1997), OPEN-R SDK, etc.), since it has been specifically designed for multi-platform applications and provides for easy and efficient implementation of modular solutions to a specific robotic problem, remote control and debug, abstraction with respect to the mobile base and the connected devices, and a set of useful tools for developing typical robotic applications. Furthermore, besides providing facilities for robot hardware abstraction, module interaction as in (Utz et al., 2002; Wang et al., 2001), MARIE, CARMEN and Player-Stage, our RDK integrates at the same time two other important mechanisms: dynamic information sharing and remote inspection, that are very important in the realization of a robotic application.

As for the OROCOS project, while the general objectives are similar to our approach, our framework is specifically targeted toward a particular kind of robotic applications and has been developed with specific goals (e.g. minimizing the computational overhead of the infrastructural layer, reducing performance decrease due to communication failures, etc.) that were driven by experience gained developing robotic applications. The SPQR-RDK is continuously increasing in the number of modules that are realized for the different applications that are currently under development within our group, but always maintaining the same middle-ware. This is an important achievement for our group since having several modules that can be combined for building different robotic applications with a minimum effort, allows for developing different solutions to common robotic problems and to evaluate them in several scenarios and in general to increase over time the quality and the effectiveness of the robotic applications developed.

6. References

S. Bahadori, D. Calisi, A. Censi, A. Farinelli, G. Grisetti, L. Iocchi, and D. Nardi. (2005) Autonomous systems for search and rescue. In A Birk, S. Carpin, D. Nardi, Jacoff A., and S. Tadokoro, eds. Rescue Robotics. Springer-Verlag.

S. Bahadori, D. Calisi, A. Censi, A. Farinelli, G. Grisetti, L. Iocchi, and D. Nardi. (2004) Intelligent systems for search and rescue. In Proc. of IROS Workshop Urban search and rescue: from Robocup to real world applications.

S. Bahadori, A. Cesta, G. Grisetti, L. Iocchi, R. Leone, D. Nardi, D. Oddi, F. Pecora, and R. Rasconi. (1995) Robocare: an integrated robotic system for the domestic care of the elderly. In Proc. of Workshop on Ambient Intelligence AI*IA-03, Pisa, Italy.

D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi. (2005) Autonomous navigation and exploration in a rescue environment. In Proc. of 2nd European Conference on Mobile Robots, Ancona, Italy. pp. 110-115 June.

B. P. Gerkey, R. T. Vaughan, and A. Howard. (2003) The player/stage project: Tools for multi-robot and distributed sensor systems. In Proc. of the Int. Conf. on Advanced Robotics (ICAR 2003), pages pp. 317-323, Coimbra, Portugal, July.

H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. (1998) Robocup: A challenge problem for ai and robotics. In Lecture Note in Artificial Intelligence, volume 1395, pages 119.

K. Konolige, K.L. Myers, E.H. Ruspini, and A. Saffiotti. (1997) The Saphira architecture: A design for autonomy. Journal of Experimental and Theoretical Artificial Intelligence, 9(1):215235.

D. Nardi et al. (1999). ART-99: Azzurra Robot Team. In RoboCup-99: Robot Soccer World Cup III, pages 695698. Springer-Verlag.

M. Piaggio, A. Sgorbissa, and R. Zaccaria. (2000) A programming environment for real time control of distributed multiple robotic systems. Advanced Robotics, 14(1):7586.

Tadokoro et al. (2000) The robocup rescue project: a multiagent approach to the disaster mitigation problem. IEEE International Conference on Robotics and Automation (ICRA00), San Francisco.

H. Utz, S. Sablatng, S. Enderle, and G. K. Kraetzschmar. (2002) Miro - middleware for mobile robot applications. IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures, 18(4):493497.

Hui Wang, Han Wang, C. Wang, and W. Y. C. Soh. (2001) Multi-platform soccer robot development system. In RoboCup 2001: Robot Soccer World Cup V, pages 471476.