A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle *

Francisco Ortiz, Diego Alonso, Bárbara Álvarez and Juan A. Pastor

{francisco.ortiz, diego.alonso}@upct.es
Universidad Politécnica de Cartagena (Spain)

Abstract. Teleoperated robots are used to perform hazardous tasks that human operators cannot carry out. The purpose of this paper is to present a new architecture (ACROSET) for the development of these systems that takes into account the current advances in robotic architectures while adopting the component-oriented approach. The architecture is currently being used, tested and improved in the development of an heterogeneous family of robots in the context of the EFTCoR project. It is also presented the Ada'95 implementation of ACROSET for a climbing robot.

1 Introduction

Teleoperated robots are used for extending human capabilities in hazardous and inaccessible environments. Recent progress in mechatronics, perception and computing is opening up a number of new application domains for tele-robotics, but at the same time, the complexity of the applications grows due to the domain characteristics: high variability of functionality and physical characteristics, large variety of execution infrastructures, sensors, actuators, control algorithms, degrees of autonomy, etc.

Despite these differences, teleoperated systems are normally similar from a logical point of view, having many common requirements in their definition and many common components, both logical or physical, in their implementation. As stated in [1], one way of dealing with this complexity is *to use architectural frameworks and tools that embody well defined concepts to enable effective realization of systems to meet high level goals.* Such an architectural framework allows rapid development of systems and reuse of a large variety of components, with concomitant savings in time and money. There are numerous efforts to provide developers with architectural frameworks of this nature, such as [2,3,4].

The objects of this paper are twofold: to present an architectural approach to the development of control units for these kind of systems and to present an example of its use in the development of a real system. The architectural

* This work has been partially supported by European Union (GROWTH G3RD-CT-00794) and the Spanish Government programs CICYT (TIC2003-07804-C05-02) and Seneca (PB/5/FS/02).

approach, ACROSET, is based on the latest advances in robotic architectures and adopts a component-oriented approach. ACROSET offers a way to re-use the same components in very different systems by separating the functionality from the interaction patterns. It also provides a common framework for developing robot systems and for integrating intelligent behaviours. ACROSET has been implemented and tested in different systems, such as a PLC of Siemens (series 300) and a small FPGA (*Field Programmable Gate Array*), which is a kind of re-programmable hardware. Actually, it is being implemented in Ada'95.

2 A Climbing Vehicle in the EFTCoR Project

The *Environmental Friendly and Cost-Effective Technology for Coating Removal* (EFTCoR) project [5,6] is part of the European Industry current effort to introduce environmental friendly ship maintenance. It addresses the development of a solution to the problem of retrieval and confinement of the subproducts obtained from the ship maintenance operation (oxide, paint and adherences mainly). A glance at Fig. 1 shows the difficulty of designing a general purpose system, or even defining a common body of general requirements that could be applied to all systems because: hull dimensions and shapes differ widely, the different areas of any given hull impose very different working conditions for robotic devices, working areas differ in different shipyards or even within the same shipyard and the particular businesses and cultures of shipyards impose different requirements and priorities.

a) Vertical Surfaces b) Bottoms c) Bows

Fig. 1. Different hull shapes in the operational domain

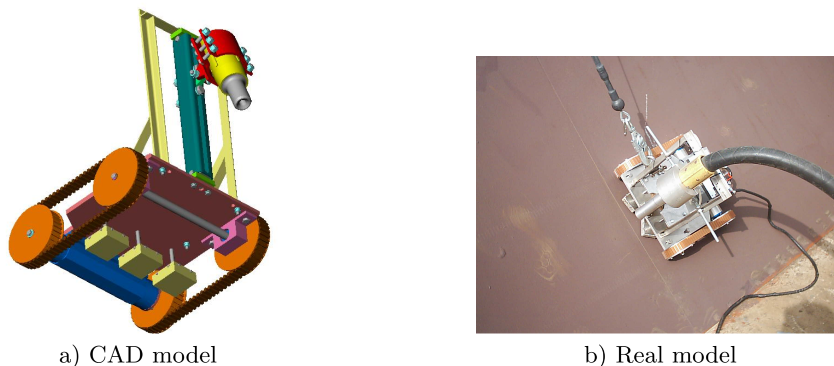
This tremendous variety generates very different problems, which require different robotic systems, each adapted to the specific problem. To solve these problems, a common design pattern has been followed for every robot of the

EFTCoR family: they generally consist of a primary positioning system capable of covering large hull areas and a secondary positioning system mounted on the primary system that can position a tool over a relatively small area (4 to 16 m^2). Different combinations of primary/secondary/tool have been considered and tested. Finally, it is important to stress that the EFTCoR is an industrial project and as such should use components that are common in industrial facilities (PLCs rather than work-stations, field buses rather LANs).

One of the members of the EFTCoR family is the Lázaro vehicle. Lázaro is a caterpillar vehicle capable of scaling a hull thanks to permanent magnets (see Fig. 2), carrying a manipulator that holds a cleaning tool. Like all members of the EFTCoR family, the vehicle can be driven by a human operator and also performs some autonomous tasks, such as obstacle avoidance and simple pre-programmed sequences.

The execution platform is an on-board embedded PC with a PC/104 expansion bus. Its based on an Intel, ultra low voltage Celeron microprocessor. The PC/104 bus is a widely used industrial standard with many advantages, such as vibration-resistance, modularity, mechanical robustness, low power consumption, etc., so its an excellent bus for embedded systems. The expansion system is formed by an analog and digital I/O board featuring 8 analog inputs, 4 analog outputs, 3 timer/counter and 24 general pourpose digital lines, and a PCMCIA expansion interface.

The Lázaro robot has two servomotors to move along the ship hull. The control of each servomotor is performed with the help of both incremental encoders. Besides this, the robot also has a ring of bumpers and infrared sensors to stop in case it gets near an obstacle or collides with it.



a) CAD model

b) Real model

Fig. 2. Different views of the Lázaro vehicle

The chosen operating system is Real-Time Linux [7], which makes it possible to have a real-time application running while retaining all the power of a Linux distribution (though with some restrictions) underneath. The Industrial

IT Group at the Universidad Politécnica de Valencia has made a GNAT Ada compiler port for the RTLinux operating system [8].

3 The ACROSET Reference Architecture

Considering the differences among systems as noted in sections 1 and 2, a central objective of the proposed architecture must clearly be to deal with such variability. A more precise analysis of the differences among systems [9] reveals that most of them relate not only to the components of the system but to the interactions among these components. Therefore, when designing the architecture the following points (architectural drivers, AD) should be borne in mind:

- **AD1:** Very different instances of the architecture should be able to share the same virtual components.
- **AD2:** The designer should adopt policies that allow a clear separation between the components as such and their patterns of interaction.
- **AD3:** The implementation of such virtual components may be software or hardware; it is highly advisable that such components can be *Commercial off the Shelf* (COTS) components.
- **AD4:** It should be possible to derive concrete architectures for both deliberative (operator-driven) and reactive (autonomous intelligent) systems.

ACROSET (*Arquitectura de Control para Robots de Servicio Teleoperados*[1]) was designed as a solution to the variability problem found on the EFTCoR project that takes account of the already mentioned architectural drives. It aims to be a reference architecture for teleoperated service robot control units. The architecture emerged from previous works at the DSIE (*División de Sistemas e Ingeniería Electrónica, Universidad de Cartagena, Spain*) [10,11] and is currently being used, tested and improved in the EFTCoR project.

ACROSET (see Fig. 3) is supposed to make it possible for very different systems to use the same components. Therefore, the first step was to define the rules and common infrastructure that would allow components to be assembled or connected. To that end, the concepts of component, port and connector were adopted as defined in [12] and they are the keypoints of ACROSET. A brief description of these concepts is given in section 5.1. The notation followed to describe components, ports and connectors is inspired by the 4 views of Hofmeister [12] and ROOM [13], which extends the UML notation with stereotyped classes and special symbols.

The first subsystem of the architecture is the *Coordination, Control and Abstraction Subsystem* (CCAS). The CCAS abstracts and encapsulates the functionality of the physical devices of the robot. The CCAS is composed of *virtual* components, which can be implemented in either software or hardware. The CCAS breaks down into several components, distributed in hierarchical layers (see 3.1). Many of the components used in a robot control unit can be found on

[1] Control Architecture for Teleoperated Service Robots

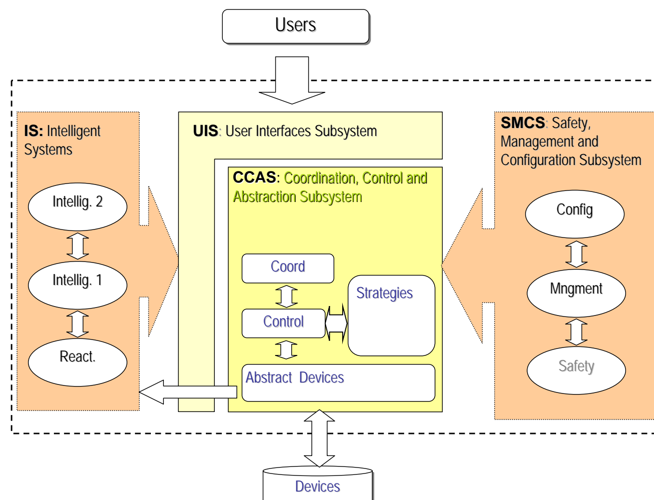


Fig. 3. An overview of ACROSET subsystems

the market either as hardware devices and control cards or software packages for a given platform. Where COTS components are used, ACROSET offers the designer two possible solutions: he can define its *virtual* counterpart or he can use the *Bridge* pattern to map an existing virtual component to the actual COTS interface.

To deal with operator-driven and semi-autonomous systems an *Intelligence Subsystem* (IS) is proposed. This way, autonomous intelligence can be added if necessary, to act as another user of the CCAS functionality. This separation of intelligence and functionality enhances the modifiability and adaptability of the system to new missions and behaviours. The intelligence can be combined with the operator commands depending on the application or mode of operation.

A *User Interaction Subsystem* (UIS) is proposed to interpret, combine and arbitrate between orders that may come simultaneously from different users of the system functionality (CCAS), since the system does not concern itself with the source of the order.

Other important aspects besides the functionality or the intelligence of the system include the safety and the possibilities of configuration and management of the application. To differentiate between functionality *per se* and the monitoring of such functionality, a *Safety, Management and Configuration Subsystem* (SMCS) is proposed. Another function of this subsystem is to manage and configure the initialisation of the application.

A complete description of ACROSET and one of its instantiations is too extensive to be included in this paper, so only details from one of the subsystems will be presented in the remaining sections. The CCAS has been the selected subsystem because its the most representative and complex subsystem of ACROSET, since it abstracts the robot functionality.

3.1 Components of the Coordination, Control and Abstraction Subsystem (CCAS)

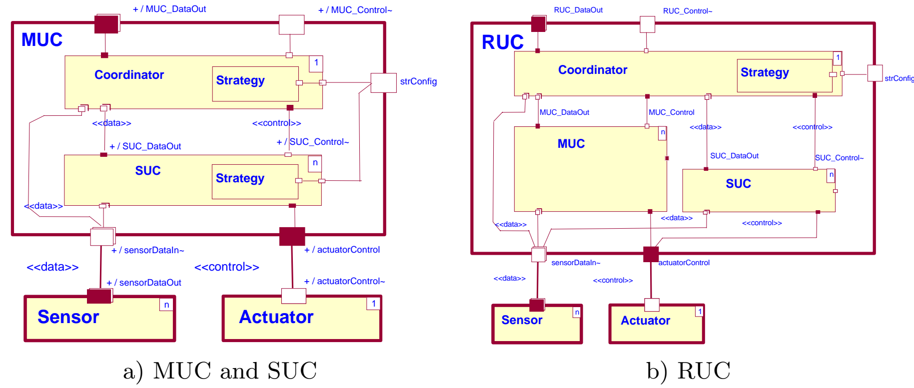
Figure 4 depicts the diagrams of the main components of the CCAS, which are defined and grouped in four layers of granularity:

Layer 1: Composed by the abstraction of the characteristics of atomic components, such as sensors and actuators.

Layer 2: *Simple Unit Controller* (SUC). SUCs model the control over one actuator of the robot (e.g. a joint).

Layer 3: *Mechanism Unit Controller* (MUC). MUCs model the control over a whole mechanism (e.g. vehicle, manipulator, end effector).

Layer 4: *Robot Unit Controller* (RUC). RUCs model the control over a whole robot (e.g. a vehicle with an arm and several interchangeable tools).



a) MUC and SUC b) RUC

Fig. 4. CCAS components diagrams

Every component of the CCAS is composed of two similar objects. On the one hand, it contains a *statechart manager*. This statechart manager decides, depending on the component current state, whether a recently issued command should be executed or not or if the state of the component should change in response to an external signal. It also controls every task created by the component. On the other hand, the object that carries out the component main purpose is interchangeable. This object follows a *Strategy* pattern, so the component behaviour can be modified even at runtime to adapt to a new state and new behaviours can be added later.

Figure 4-a depicts the *SUC* component. SUCs are meant to control actuators, so the `ControlStrategy` is the interchangeable object in this case; for example, the `ControlStrategy` of a given joint may be a traditional control algorithm (PID) or may be changed for a fuzzy logic one. SUCs usually need to accomplish hard real-time requirements and are therefore generally implemented

in hardware. When they are implemented in software, they impose severe real-time constraints on operating systems and platforms. In such case, SUCs also need a task to periodically generate the control signal according to the algorithm present in the `ControlStrategy` object.

Figure 4-a also depicts the *MUC* component. MUC components are logical entities composed of an aggregation of SUCs and a coordinator, which coordinates SUC actions according to the commands and information it receives. The interchangeable object of the MUC is the `CoordinationStrategy`; for example, the `CoordinationStrategy` of a given manipulator may be a particular solution for its inverse kinematics. Although the architecture defines MUCs as relational aggregates, they can actually become components (hard or soft) when the architecture is instantiated to develop a concrete system. ACROSET allows the designer decide whether the MUC interface provides access to its inner components or not. In fact, although MUCs may be implemented in either hardware or software, they are very commonly commercial motion control cards that constrain the range of possible commands to its internal components.

Finally, Fig. 4-b shows the *RUC* component. RUCs are an aggregation of MUCs and a global coordinator that generates the commands for its MUCs and coordinates their actions. As in the case of MUCs, the `CoordinationStrategy` is the interchangeable object. For example, the `CoordinationStrategy` of a robot composed of a vehicle with a manipulator could be a generalised kinematics solution that contemplates the possibility of moving the vehicle to reach a given target. Like MUCs, RUCs are logical components that can become physical components depending on the concrete instantiation. In general, RUCs are quite complex, comprise both hardware and software elements and can expose a wide variety of interfaces.

4 Instantiation for the Lázaro Vehicle

Figure 5 shows the CCAS instantiated for this system. As can be seen in the figure, two different MUCs have been implemented: one to control the vehicle and another to control the manipulator. The first contains one SUC to control each of the electrical motors that move the vehicle. The manipulator MUC coordinates two SUCs, one for each manipulator axis. The vehicle uses a tool that consists of an enclosed nozzle for making the blasting and recovering of residues.

The motion controllers have been implemented by means of Ada packages that implement the interfaces defined by ACROSET. In this case, the implementation allows direct access to the hardware without the mediation of any SUC. Two different intelligent behaviours have been added to the IS: obstacle avoidance and simple pre-programmed sequences. The components of the IS that implement these behaviours obtain the information they need from the vehicle sensors and generate commands to the CCAS. Integration between these commands and the operator commands is resolved by an arbitrator in the UIS.

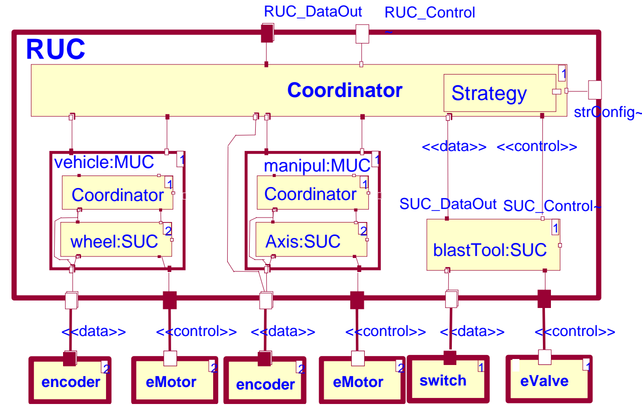


Fig. 5. Components of CCAS in climbing vehicle Control Unit

5 Implementation of the Architecture

In the implementation phase, the conceptual view must be mapped to a module view [12]. We have chosen the object-oriented paradigm and Ada95 to implement the architecture, so components, ports and connector will be mapped to classes, objects, associations and dependencies. In this section, some important aspects of the implementation will be presented.

5.1 Ports and Connectors

As mentioned in section 3, ACROSET follows a component-oriented approach. Two important concepts when talking about components are the concepts of *port* and *connector*. Connectors communicate two compatible ports of two components. Only the functionality offered by the ports of the component can be invoked, using for that the communication protocol encapsulated by the connector. This way, the content of the message is separated from how it is sent.

The concept of *port* is similar to that of interface, but with two differences: ports involved both the operations offered and required by the component and they implement the necessary services to fulfil the communication protocol appropriate to its connector. *Connectors* allow the flow of information between components, and can be as simple as pipes and events or as complex as the client-server protocol.

Changing a connector basically implies the change of the communication protocol between the ports it connects. This variation should be reflected in the modification of the port services referred to the communication, but not of those referred to the component functionality, which is accessed through the port. To separate these concepts (separation of concerns strategy [12]), ACROSET defines as many port types as possible communication ways exist. Port types are defined according to the functionality that they must fulfil (Control Input

Port for a SUC, Out Data Port and Input Data for a sensor, etc). Defining the communication protocol for these last ports is as simple as inheriting from the desired protocol port, as showed in Fig. 6.

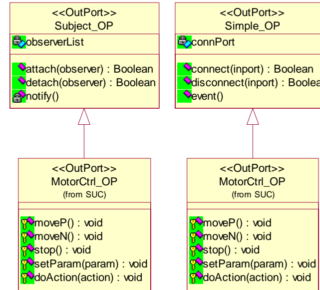


Fig. 6. Ports and connectors implementation

5.2 SUC Implementation

For the same reason that a complete description of ACROSET cannot be described in this paper, only an example of a component implementation is presented in this section. The chosen component is a SUC to control one motor as a representative part of the system (see Fig. 7).

The Motor_SUC class contains the ports showed in Fig. 6 with stereotypes ≪*InPort*≫ and ≪*OutPort*≫, to get data (`Data`) or produce control (`Ctrl`) and to configure the SUC (`Config`). Ports belong to the component and they are created and destroyed with it, so they have a composite relation, as Fig. 7 shows. The operations offered by the control ports match with the events sent by other components to the SUC. Data ports are implemented as generics, showing the same interface to any component. Besides ports, class Motor_SUC contains the interchangeable `ControlStrategy` object (the control algorithm).

In case of a Tool_SUC, all the classes shown in Fig. 7 remain with the same interface, excepting control ports (`Ctrl`). These ports must be adapted to the control events related to this particular tool.

The rest of components of the instantiation of ACROSET for Lázaro have been built in a similar manner, extending their interfaces to the needs of the system. Notice that the SUC interface remains similar in every component thanks to the method `processCommand()`, which process any incoming event in its particular control inport, of course, the implementation of that method is different for SUC, MUC and RUC.

5.3 Execution View

Following the 4 views of Hofmeister [12] notation, the execution view describes the structure of a system in terms of its runtime platform elements (tasks, pro-

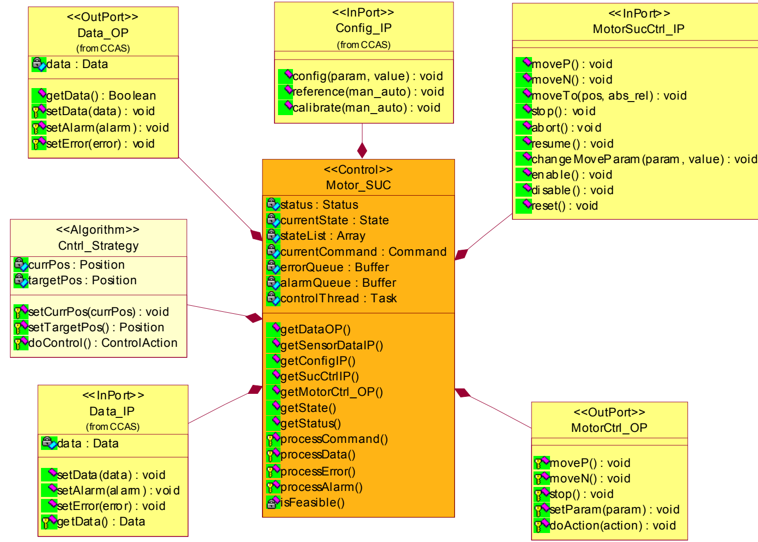


Fig. 7. Class diagram of the Motor_SUC

cesses, address spaces, etc). In this view, the objects identified in the module view of the system are mapped to a concurrent tasking architecture, where concurrent tasks, task interfaces and interconnections are defined. The driving forces behind the decisions for designing the execution architecture view are performance, distribution requirements and the runtime platform, which includes the underlying hardware and software platforms.

Too many tasks in a system can unnecessarily increase its complexity because of greater inter-task communication and synchronisation, and can lead to increased overhead because of additional context switching. The system designer has to make tradeoffs between introducing tasks to simplify and clarify the design and keeping their number low so that the system is not overloaded.

To help the designer determine the concurrent tasks and to make these tradeoffs, the COMET method [14] provides a set of heuristics which capture expert designer knowledge in the software design of concurrent and real-time system, so called *task structuring criteria*.

Hofmeister proposes as a good starting point to begin by associating each high-level conceptual component with a set of execution elements. Considering that the main objects of the system have been proposed as an instantiation of ACROSET (see Fig. 5), the task structuring criteria might be applied to determine which of the components in the IS, CCAS and SMCS may execute concurrently, and which need to execute sequentially and therefore are grouped into the same task. In a second stage, the task clustering criteria are applied,

with the objective of reducing the number of tasks. Figure 8 depicts the task diagram obtained by applying the following task clustering criteria:

Temporal clustering. Since the tasks involved in controlling the I/O show no sequential dependency and their activation periods are multiples, they have been grouped in one task.

Task inversion. Instead of using a task for each SUC, all identical tasks of the same type have been replaced by one task that performs the same service. Each object state information is captured in a separate protected object. Although SUCs have been grouped, the tasks that perform their concurrent control activities (e.g. a periodic control algorithm) remain as separate tasks, with the task type `Cmd_Control`. The same happens with MUC and SUC command control.

Sequential clustering. To avoid overloading the system, all the **UC* state control tasks have been grouped in the CCAS task because the information flows up and down always in a sequential order.

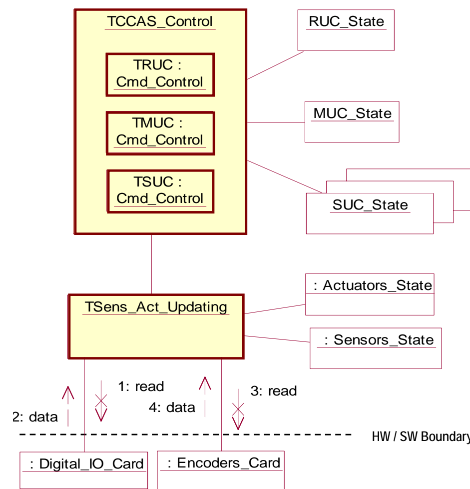


Fig. 8. Task diagram after clustering

6 Conclusions

The use of a common architecture for a domain or family of systems allows rapid developments and the reuse of components. This paper has presented a common architectural framework for the development of teleoperated service robots control units (ACROSET) and also an application example in the context

of the EFTCoR project (the Lázaro vehicle), that shows the ability of ACROSET to cope with the needs and requirements of very different systems.The separation of the conventional functionality of the systems (CCAS) from the intelligent behaviours (IS) greatly facilitates the addition of new functionalities and the maintenance of applications.

Perhaps, the main contribution of ACROSET to the current state of the art is the conceptual component oriented approach, which makes the components independent to the implementation language or hardware/software partition. This has allowed implementing those components as PLC blocks, not only as objects and classes, e.g. as CLARAty [2] does.

References

1. E. Coste-Manière and R. Simmons. Architecture: the Backbone of Robotics Systems. In *Proc. of the 2000 IEEE International Conference on Robotics & Automation*, pages 67–72, April 2000. ISBN: 0780358864.
2. I. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W.S. Kim. CLARAty: An Architecture for Reusable Robotic System. March 2003. Jet Propulsion Laboratory, NASA.
3. K.U. Scholl, J. Albiez, and B. Gassmann. MCA – An Expandable Modular Controller Architecture. In *4th Real-Time Linux Workshop*, 2001.
4. H. Bruyninckx, B. Konincks, and P. Soetens. A Software Framework for Advanced Motion Control. Draft version, January 2002.
5. C. Fernández, A. Iborra, B. Álvarez, J.A. Pastor, P. Sánchez, J.M. Fernández-Meroño, and N. Ortega. Co-operative Robots for Hull Blasting in European Shiprepair Industry. *IEEE Robotics & Automation Magazine (Special Issue on Industrial Robotics Applications & Industry-Academia Cooperation in Europe - New Tendencies and Perspectives)*, November 2004. ISSN: 1070-9932.
6. EFTCoR Official Site. http://www.eftcor.com/.
7. M. Barbanov. *A Linux-based Real-Time Operating System*. PhD thesis, New Mexico Institute of Mining and Technology, June 1997.
8. M. Masmano, J. Real, I. Ripoll, and A. Crespo. Running Ada on Real-Time Linux. In *Reliable Software Technologies - Ada-Europe 2003*, volume LNCS 2655, pages 322–333. Springer-Verlag, June 2003.
9. J.A. Pastor Franco. *Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales*. PhD thesis, Technical University of Cartagena (Spain), 2002.
10. F. Ortiz, A. Iborra, F. Marín, B. Álvarez, and J.M. Fernández-Meroño. GOYA: A teleoperated system for blasting applied to ships maintenance. In *3rd International Conference on Climbing and Walking Robots*, October 2000. ISBN: 1-86058-268-0.
11. A. Iborra, J.A. Pastor, B. Álvarez, C. Fernández, and J.M. Fernández-Meroño. Robots in Radioactive Environments. *IEEE Robotics and Automation Magazine*, 10(4):12–22, December 2003. ISSN: 1070-9932.
12. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, January 2000. ISBN: 0-201-32571-3.
13. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling (ROOM)*. John Wiley and Sons, 1994. ISBN: 0471599174.
14. H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Object Technology. Addison-Wesley, 2000. ISBN: 0-201-65793-7.