A layered architectural component model for service teleoperated robots

Juan A. Pastor, Bárbara Álvarez, Pedro Sánchez, Francisco Ortiz

juanangel.pastor@upct.es
Universidad Politécnica de Cartagena (Spain)

Abstract.[1]. The teleoperated robots are used for performing works that human operators can not carry out due to the very nature of their tasks or the hostile nature of the working environment. Though many control architectures have been defined for developing such kind of systems and reuse common components, none of them have reached all its objectives due to the great variability among systems behaviors. The purpose of this paper is to present a new architectural approach for the development of these systems that takes into account the current advances in robotic architectures as well as the component-oriented approach. The described architectural approach provides a common framework for developing robotized systems with very different behaviors and for integrating intelligent components. The architecture is currently being used, tested and improved in the development of a family of robots, teleoperated cranes and vehicles, which perform an environmentally friendly cleaning of hull-ship surfaces (the EFTCoR project). This paper summarises the methodological approach taken, the features of the systems that constitute its design drivers and finally the main characteristics of the proposed architecture.

Keywords: software architecture, teleoperated robots, component-oriented approach.

1. Introduction

The purpose of this paper is to present a reference system architecture for service robot control applications. These applications are used to teleoperate mechanisms, such as robots, vehicles and tools (or a combination of them), which perform inspection and maintenance activities in hostile environments. In general, these activities are complex and it is not possible to work with completely autonomous systems. Thus, the operator is in charge of monitoring and operating the mechanisms. The system receives orders from a human operator and performs the corresponding actions for executing them.

[1] This work was partially supported by the CICYT (with reference TIC2003-07804-C05-02) and the Regional Government of Murcia (Séneca Programmes with reference PB/5/FS/02)

Teleoperation systems cover a broad range of mechanisms and missions, each of them with their specific features and requirements. However, at the same time, all of them share many common characteristics making possible to describe an application domain and its corresponding reference architecture. In fact, during the last years the research group DSIE of the Technical University of Cartagena has been using a reference architecture to afford several developments for the nuclear industry [1]:

- The teleoperation software of the ROSA III robot of Westinghouse, used for maintenance operations inside the channels heads of the steam generators of the pressurized water nuclear plants.
- The vehicle IRV used for the search and retrieval of fallen objects inside the pipes of the primary circuit of nuclear plants.
- The TRON system design for the inspection of the lower internals of the PWR vessels.

Despite their differences, these systems share some key characteristics from the point of view of their control and therefore it is relatively easy to use the same architecture to develop them:

- The working areas are fixed and well known.
- The behaviour is operator driven. Reactive behaviour is limited to some simple safety actions.
- The applications control a single system.

However, none of these characteristics stand in the new developments considered in the EFTCoR project [2], where the DSIE is currently working on. The EFTCoR system comprises a family of teleoperated systems whose mission is to retrieve and confine the paint, oxide and marines adherences of hull ships. In this case:

- The working environments are not fixed due to the great variability among the different kind of ships, the different areas of a given ship and the differences among shipyards.
- The systems should have a great degree of autonomy.
- It is possible that different systems have to work cooperatively at the same time.

These new characteristics make impossible the use of the original architecture for the EFTCoR robots. However, the use of a common architecture for all the developments is extremely useful. It allows the rapid development of the systems and the reuse of a great variety of components, saving time and money. For this reason, the DSIE research group is working on a new architecture that takes into account these new characteristics and can be used for the development of the robots considered in the EFTCoR project. This paper summarises the main characteristics of this architecture and it is structured as follows: section 2 presents and justifies the methodological approach taken; the section 3 describes the limits of the considered system and the main issues for the architecture definition. The sections 4 and 5 describe the architecture and section 6 summarises the conclusions.

2. Methodological approach

Although many robotics architectures can be found in bibliography [3], it is more difficult to find examples of a development process for defining reference architectures in the robotics domain. In our proposal to reach a reference architecture for service robot control applications, the Architecture Based Design Method (ABD) [4] has been followed, completing it with the 4 views of Hofmeister [5] with their notation based on UML for components.

The development methods based on use-cases (mainly RUP [6] and others derived from RUP) could be appropriate for defining the architecture of a given system, but they are not suitable to define reference architectures. The use cases define concrete functionality, however in the design of reference architectures the issue is not the concrete, but the general, because *the success or failure of such architectures depends on its ability to deal with the variability among the systems of the considered domain*. In this sense, use cases that could be very relevant to one system are negligible for others. Furthermore, at the level of abstraction required to manage the variability of the systems, concrete use cases can not be properly defined. For this reason, we have adopted another methodological approach: the ABD.

The ABD is a methodology proposed by the SEI (Software Engineering Institute of The Carnegie Mellon University) to design software architectures for a given application domain or product family. The ABD is based on:

- The functional decomposition of the problem based on the concepts of low coupling and high cohesion and on the knowledge of the application domain.
- The realization of the functional and quality requirements by means of a correct choice of architectural styles and design patterns.
- The notion of software templates that define the elements and responsibilities common to a group of components, such as their interactions with the infrastructure.

ABD decomposes the system into subsystems recursively. Thus, the same rules that apply to decompose the system into subsystems apply to decompose the subsystems in other simpler subsystems.

ABD offers as final model a conceptual view of the architecture, identifying the main subsystems and their relationships described in terms of architectural styles and design patterns. Hofmeister et al [5] propose another architectural oriented development method, which can be superposed to ABD in their initial steps. The approach of these authors is interesting because it includes the notions of port and connectors among components, using a ROOM inspired notation [7]. In this case, the UML notation has been extended with stereotyped classes and special symbols, for expressing such components, ports and connectors. The Hofmeister's approach also makes easier the connection between the conceptual components and their implementations.

3 Domain characterisation. Teleoperated service robots.

The service robots are mechatronic systems, usually designed for a concrete application that could be extended with new functionality along the time. Though they could be very different from a physical point of view, they use to be logically very similar sharing many common components, both logical and physical ones. The characterisation of the application domain is the starting point to define the functional and quality requirements that guide the architecture design. In our experience, the main features to be considered should be the following:

- High degree of specialisation and therefore a great variability of functionality and physical characteristics.
- Different combinations of vehicles, manipulators and tools.
- A great variety of execution infrastructures, including different kinds of processors, communication links and man-machine interfaces.
- A great variety of sensors and actuators.
- Different kinds of control algorithms, from very simple reactive actions to extremely complex algorithms and navigation strategies, depending on the applications.
- Different degree of autonomy, from completely operator-driven systems to semi-autonomous robots.
- Presence of hard real time requirements.
- Hardware versus software intensive implementations with all the imaginable intermediate cases.
- And last, but not least, safety is nearly always a main concern.

Considering the differences among systems described above, it is clear that the main objective of the architecture is to deal with such variability. A more precise analysis of the differences among systems [8] reveals that most of them refer not to the system components, but to the components interactions. Therefore, when designing the architecture the following points should be taken into account:

- It should be possible that very different instantiations of the architecture can share the same *"virtual"* components.
- The designer should adopt policies that allow a clear separation between the components and their interaction patterns.
- The implementation of such virtual components could be software or hardware, being highly advisable that such components could be COTS.
- It should be possible to derive concrete architectures for both, deliberative systems (operator-driven systems) and reactive systems (autonomous intelligent systems).

Following the ABD terminology, these four points constitute the *architectural drivers* of the architecture.

4. Architecture overview.

Since it should be possible that very different systems use the same components, the first issue is to define the rules and common infrastructure that allow components to be assembled or connected. To achieve this aim the key concepts are: component, container, port and connector, as well as the Composite pattern [9]. The concept of port provides a regular way of interchange data and control and therefore of connecting and assembling components, independently of their functionality and granularity. The concept of connector allows separating the functionality of the components from their interaction patterns (choreographies [10]), because they are included inside the connectors. The Composite pattern provides the means to deal with complex and simple components in the same way, hiding the inner complexity of the large components resulting of the assembly of many other components.

Once it has been defined how the components have to be or can be assembled, the second point is to define what components have to exist. The third architectural driver identified in the previous section states that the implementation of the components could be software or hardware, being highly advisable that such components could be COTS. To achieve this, it is necessary to define the typical components of this kind of systems, which can be identified at different levels of granularity. At the lowest level we can find the actuators and sensors. At a higher level we find the controllers for simple actuators (for example a motor controller). At the next higher level, we find the controllers for groups of actuators (for example a motion card capable of controlling the joints of a mechanism) and so on. Many of these components can be found in the market either as hardware devices and control cards or software packages for a given platform. To facilitate the use of COST components, the most usual COST should have its virtual counterpart. The attachment between the virtual component and its implementation can be done using the Bridge pattern [9].

To define virtual components the architecture identifies four levels of granularity and adopts the notion of *hardware abstract layer* described in the OROCOS framework [11]. The hardware abstract layers model the features of the physical components of the system, defining virtual sensors, actuators, motion controllers, etc. The hardware abstract layers allow defining libraries of components and interchange both hardware and software implementations (perhaps commercial) of the devices with a minimum impact.

Finally, the last architectural driver identified was the possibility of deriving concrete architectures for both deliberative and reactive systems. For this purpose, it is necessary to separate the autonomous or programmed behaviour from the operator driven behaviour, as shown in figure 1. This scheme also appears in the CLARAty (Coupled Layered Architecture for Robotic Autonomy) architecture [12] used for the development of the Mars rovers. CLARAty distinguishes a Functional Layer, where the components of the system are defined, and a Decision Layer that encapsulates the subsystems responsible for planning and executing the missions. Unlike CLARAty, where some autonomous behaviours can be added to the functional layer, in our approach the system's intelligence is completely separate from its functionality because it has been designed for the domain of teleoperated robots rather than autonomous systems, so the conception of the intelligence as another user of the functionality, like the teleoperator, was considered a more important issue.

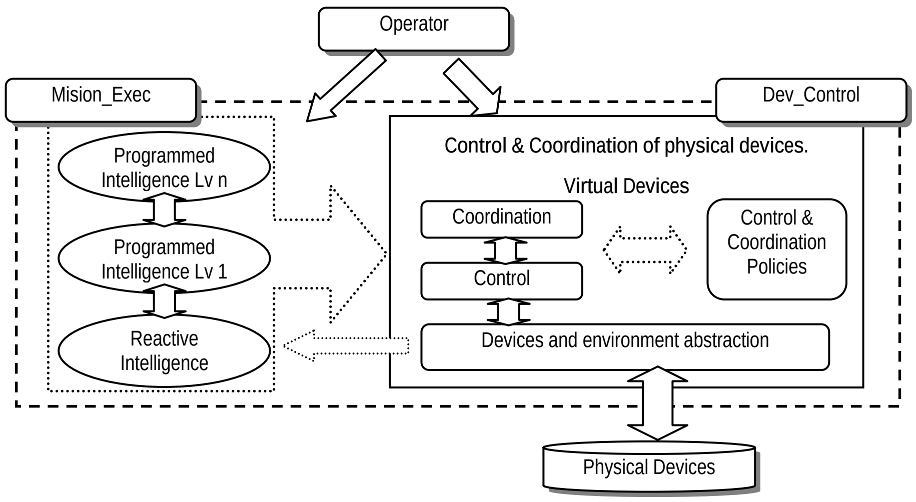
Operator
Mision_Exec
Dev_Control
Programmed Intelligence Lv n
Programmed Intelligence Lv 1
Reactive Intelligence
Control & Coordination of physical devices.
Virtual Devices
Coordination
Control
Control & Coordination Policies
Devices and environment abstraction
Physical Devices

Figure 1: An abstract overview of the proposed architecture.

5. An overview of the architecture layers and components.

The architecture proposed in this paper identifies four layers of granularity at which the components can be defined:

- Layer 1: Abstract the characteristics of atomic components, such as sensors and actuators.
- Layer 2: Simple Unit Controllers (SUCs).
- Layer 3: Mechanisms controllers (MUCs).
- Layer 4: Robot controllers (RUCs).

These layers are called *hardware abstract layers* because the components defined inside them could be (and frequently are) implemented in commercial hardware. The simplest components modelled by the architecture are the sensors and actuators, which are defined at the lowest architectural layer. The sensors are components that provide the information required for controlling a given active element, for example, the encoder and switch limits associated to a given joint. The actuators model the simplest active elements, for example a motor.

The SUCs (Simple Unit Controllers) are the components defined at the second architecture layer. The SUC components (figure 2) model the control over the actuators and the collection of data from sensors. For example, there will be SUCs defined for controlling the joints of a given mechanism. The SUC generates the commands for the actuator according to the order that it receives from another component (through the port **`controllerControl`**), the information received from the sensors that describe the state of the actuator as well as the control policy that comprises. This policy is an interchangeable part of the SUC. For example, the **`ControlStrategy`** of a given joint could be a traditional control (PID) or be changed for a fuzzy logic strategy.

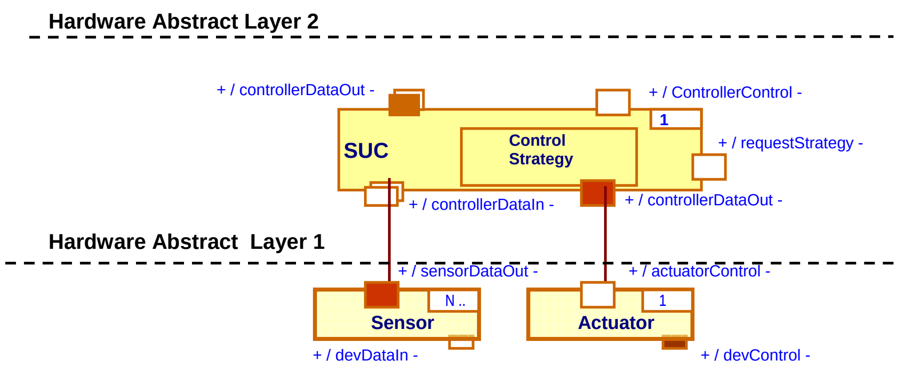

Figure 2: SUC: Simple Unit Controller.

The SUCs usually need to accomplish hard real time requirements and therefore they are generally implemented in hardware. When they are implemented in software they use to impose severe real time requirements on operating systems and platforms.

At the third level of granularity it is defined the Mechanism Unit Controller (MUC). The MUC component models the control over a whole mechanism (vehicle, manipulator or end effector). As it is shown in figure 3, the MUC is a logical entity composed of an aggregation of SUCs and a Coordinator responsible of coordinating their actions according to the command and information that it receives, as well as the coordination strategy that comprises. This strategy is an interchangeable part of the SUC. For example, the **`CoordinationStrategy`** of a given manipulator could be a given solution for its inverse kinematics, the coordinator strategy for a given vehicle could be a given navigation strategy, etc.

Although the architecture defines the MUCs as relational aggregates, they can become inclusive components (hard or soft) when the architecture is instantiated to develop a concrete system. Whether or not the interfaces of the inner SUCs are directly accessible is a decision of the architecture instantiation. In fact, though MUCs could be implemented in hardware or software, it is very usual that they are commercial motion control cards that constrain the range of possible command over its internal components. The COTS limit the flexibility of the approach, in the sense that COSTs do not always provide direct access to their inner sub-components neither their inner state.

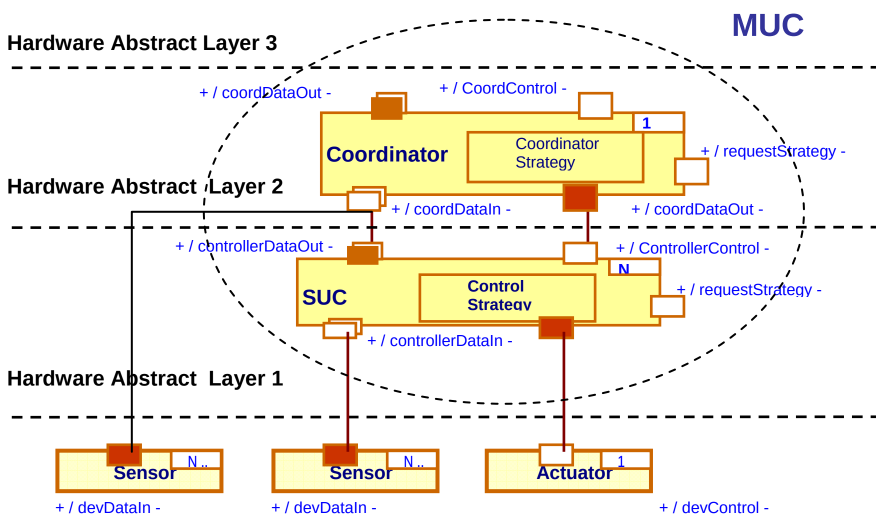

Figure 3: MUC: Mechanism Unit Controller.

Finally, at the fourth layer, the architecture defines the RUC (Robot Unit Controller) component. The RUC component models the control over a whole robot. For example, a robot composed of a vehicle with an arm and several interchangeable tools. As figure 4 shows, the RUCs are an aggregation of MUCs and a global coordinator that generates the commands for the MUCs and coordinates their actions, according to the order and the information that it receives and the coordination strategy that comprises. Such strategy is an interchangeable part of the RUC. For example, the **`CoordinationStrategy`** of a robot composed of a vehicle with a manipulator could be a generalised kinematics solution that takes into account the possibility of moving the vehicle to reach a given target. As the MUCs, the RUCs are logical components that can become into physical components depending on the concrete instantiations. In general, the RUC is a rather complex component that comprises hardware and software components and can expose a great variety of interfaces depending on the complexity of the controlled system.

Having defined the SUCs, MUCs and RUCs, it seems logical to define a Group Unit Controller (GUP) capable of managing and coordinating a group of cooperative robots. However, the architecture does not go beyond the RUCs. There is a good reason for this. The *"usual intelligence"* required for controlling a joint or the mechanism that results from the assembly of joints or to teleoperate the robot that results from the combination of various mechanisms is limited, well-known and can be enclosed inside reusable components. The intelligence required to work cooperatively usually demands a more flexible approach. It is also true for some missions that concern the SUCs, MUCs and RUCs, likewise the algorithms for collision avoidance or the navigation systems for vehicles. It is very difficult to define a component to encapsulate "intelligence". If a system or component is capable to offer intelligence and to take non-trivial decisions it uses to be complex enough to have defined their own architecture (for example, an artificial vision system capable to determine paths free of obstacles). Thus, the approach should be different: Let's the *intelligent* components are as they want and provide a way to integrate them into the system.

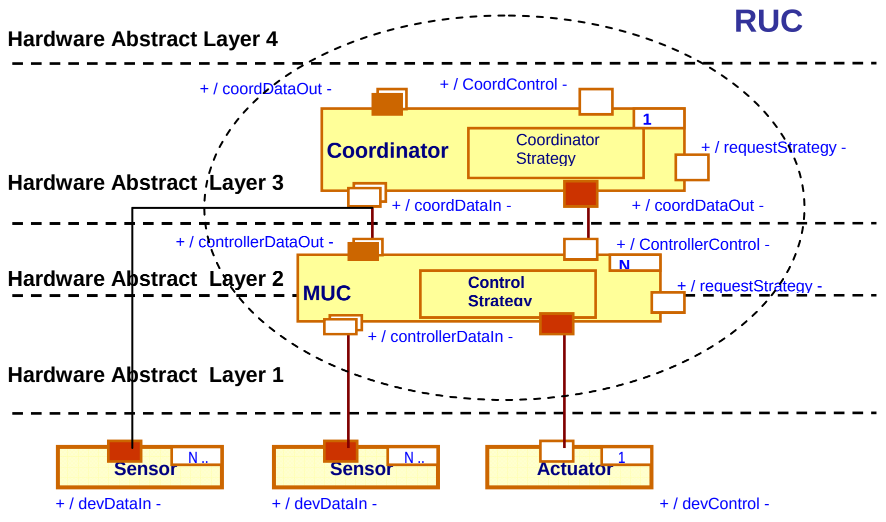

Figure 4: RUC: Robot Unit Controller.

6. Adding autonomous behaviour.

The composition of SUCs and MUCs results in a hierarchical architecture where the decisions flow from top to down and the information flows from bottom to up. This architecture fits well with operator-driven systems, where the autonomous behaviour does not exist or it is reduced to some safety hardware actions. It also fits well with systems where the reactive or autonomous behaviour responds to simple rules that can be added to controllers and coordinators which, following these rules, can take decisions and notify them to the upper level controller or coordinator. However, there are systems where the autonomous behaviour is anything but simple. In such cases, the *intelligent component* needs to integrate more information and access more functionality than those embedded inside a given component. The approach taken (showed in the figure 5) consists of *superimposing* the "*intelligent*" autonomous behavior and the operator-driven behavior, providing the means for integrating both and resolving the potential conflicts. This approach does not imply any change in the components defined until now but new sources of commands for them. These sources are constituted by new components that have access to the global system information and are capable of deciding what to do according to some programmed rules, algorithms or heuristics.

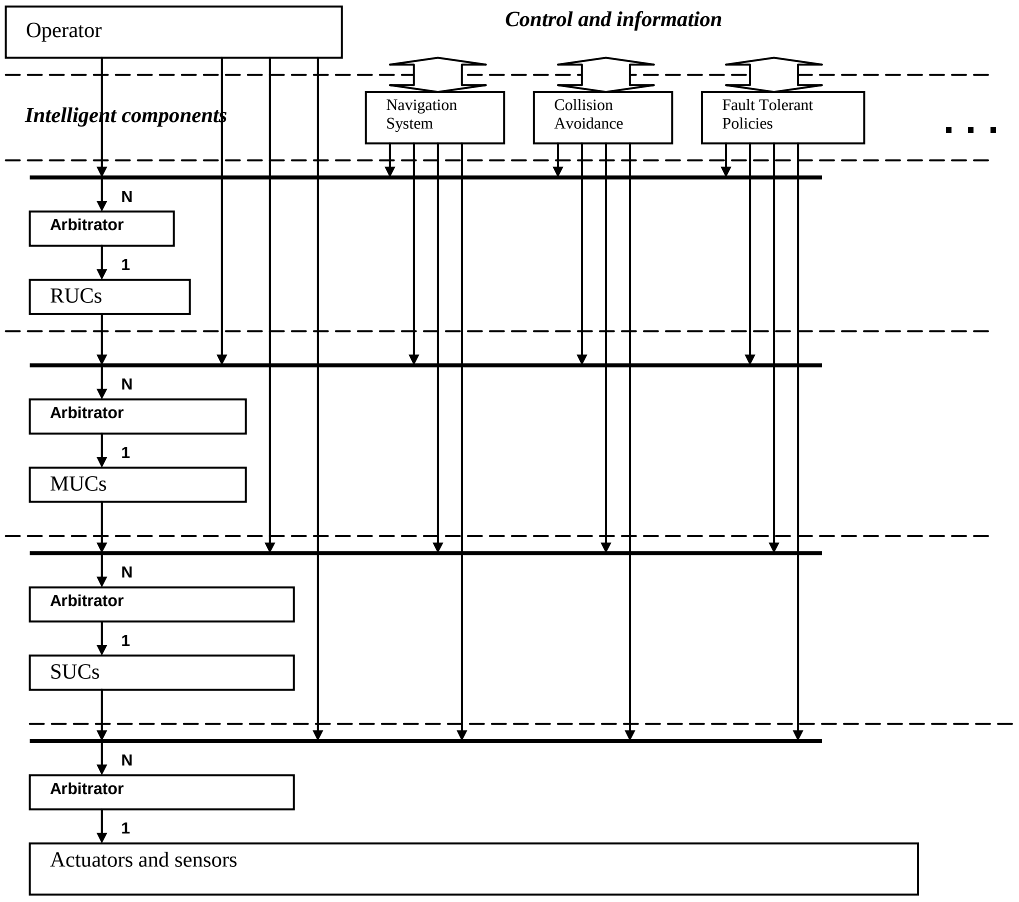

Figure 5: Superimposition of operator-driven and autonomous behaviour.

Every component of a given layer can access the information and control ports of the lower layers components. From this point of view, every component of a given layer is an intelligent component for the lower layer. For example, from the point of view of a MUC, no matter whether the commands come from the coordinator of the RUC that comprises it (see figure 5), from the operator or from some of the intelligent components defined over the RUCs. As a component can receive commands from more than one source, it is necessary to decide what command to perform. The logic for this decision is outside the component. The figure 5 shows a new type of component: the **`arbitrator`**. Arbitrators encapsulate the rules that determine which command should be delivered to a given component. The **`arbitrator`** is separately defined because the rules that it encloses (or even the **`arbitrator`** itself) can vary from system to system, during the life of a given system or even at different stages of the functioning of a system. The concept of arbitrator is inspired in the idea of *composition filter* [13] and is strongly connected to the need of separating the functionality from the interaction patterns among components.

This approach is very flexible and allows integrating intelligence that does not concern directly with the missions of the robotic devices, but with the management of the application, such as fault tolerance policies or a meta-layer for reconfiguring the application.

6. Summary and future work

The architecture described in this paper takes the most promising architectural advances in the domain of teleoperation and put them together with a component-oriented approach. The taken approach focuses in the definition of a common component framework that allows the definition of components that can be reused in different systems, as well as in the integration of intelligent systems capable to drive the robot behaviour. Our main sources of inspiration have been both OROCOS [11], CLARAty [12] (robotic architectures) and the PRISMA approach [10] (component and aspect oriented approaches).

The architecture is currently being used in the development of a family of robots whose mission is to retrieve and confine the paint, oxide and marines adherences of hull ships (see figure [6]). This family of robots shows a broad variety of behaviours and complexity degrees, being an excellent test bench for the architecture.

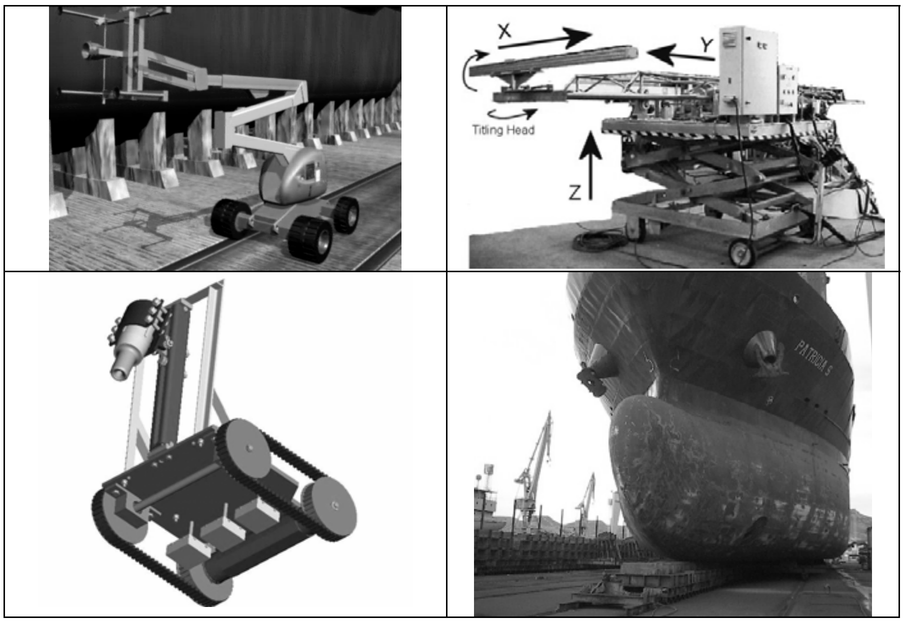

Figure 6: Three prototypes (cherry-picker model, elevation platform and mobile vehicle, respectively) of the family of robots and a ship to be repaired.

Our experience using the architecture is satisfactory, however it is necessary to remark, among others, two important challenges:

- There is no enough support for expressing the component abstractions and modelling their interactions.

- Although there are well known approaches to cope with the variability [14], they do not offer a concrete way to solve evolution of architectural components and between their relationships.

These challenges can be afforded by means of the PRISMA approach. In this way, we are currently working with the Technical University of Valencia (Spain) in the frame of the national funded (CICYT) research project DYNAMICA with ref. TIC2003-07804-C05. The first step could be the use of the PRISMA language for defining the components and the above layered architecture. A second step could be the consideration of changes in the interactions between these components.

7. References

1. Iborra A., J.A. Pastor, B. Álvarez, C. Fernández, J.M. Fdez-Meroño. "*Operational Experiences using Robotics during Maintenance Services in PWR Nuclear Power Plants*". IEEE Robotics&Automation Magazine, ISSN 1070-9932, December 2003.
2. *Environmental Friendly and Cost-Effective Technology for Coating Removal* (EFTCOR). Fith Framework Program, Growth (GRD2-2001-50004).
3. Coste-Manière E., R.Simmons, "Architecture, the Backbone of Robotic System", Proc. of the 2000 IEEE international conference on robotics & Automation, San Francisco, abril 2000.
4. Bachmann F., L. Bass et al, "*The Architecture Based Design Method*", Technical Report CMU/SEI-200-TR-001, Carnegie Mellon University, USA. January 2000.
5. Hofmeister C., R. Nord, D. Soni, "*Applied Software Architecture*", Addison-Wesley. ISBN 0-201-32571-3. USA. Enero 2000.
6. Jacobson I., G. Booch, J. Rumbaugh, "The Unified Software development Process". *Addison-Wesley 1999, ISBN 0-201-57169-2.*
7. Selic B., G. Gullekson, P.T. Ward, "Real-Time Object-Oriented Modeling" (ROOM). John Wiley and Sons, New York. 1994.
8. Pastor J., "Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales", ph Thesis, Universidad Politécnica de Cartagena (Spain), 2002.
9. Gamma E., R. Helm, R. Johnson, J. Vlissides, "*Design Patterns: Elements of Reusable Object Oriented Software*", Addison Wesley, Reading Mass. 1995.
10. Pérez J., I. Ramos, J. Jaen, P. Letelier and E. Navarro. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, Nov. 2003.
11. Bruyninckx H., B. Konincks, P. Soetens, "A Software Framework for Advanced Motion Control", Dpt. of Mechanical Engineering, K.U. Leuven. OROCOS project inside EURON. Bélgica. Febrero 2002.
12. Nesnas I.A., A. Wright, M. Bajracharya, R. Simmons, T. Estlin, Won Soo Kim, "CLARAty: An Architecture for Reusable Robotic Software," SPIE Aerosense Conference, Orlando, Florida, April 2003.
13. Bergmans L., Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands, 1994.
14. Svahnberg, M, Van Gurp J., Bosch J. On the Notion of Variability in Software Product Lines. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), IEEE Computer Society, pages 45-54, 2001.