An Architectural Framework for Modeling Teleoperated Service Robots

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

División de Sistemas e Ingeniería Electrónica (DSIE)

Universidad Politécnica de Cartagena (Spain)

Campus Muralla del Mar, s/n. Cartagena, E-30202, Spain

juanangel.pastor@upct.es

Abstract.[1]. Teleoperated robots are used to perform tasks that human operators cannot carry out because of the nature of the tasks themselves or the hostile nature of the working environment. Though many control architectures have been defined for developing these kinds of systems reusing common components, none has attained all its objectives because of the high variability of system behaviors. This paper presents a new architectural approach that takes into account the latest advances in robotic architectures while adopting a component-oriented approach. This 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 teleoperated robots which perform cleaning of ship-hull surfaces.

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

Keywords: Software Architectures. Distributed objects, components, containers. Commercial robots and applications.

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 these) that perform inspection and maintenance activities in hostile environments. These activities are generally complex and it is not possible to work with completely autonomous systems. Therefore, the operator is in charge of monitoring and operating the mechanisms. The system receives orders from a human operator and performs the requisite actions to execute them.

Teleoperation systems cover a broad range of mechanisms and missions, all with their own specific features and requirements. At the same time, however, they all share many common characteristics, making it possible to describe an application domain and its corresponding reference architecture. In fact, in recent years the DSIE research group at the Technical University of Cartagena has been using a reference architecture to enable a number of developments for the nuclear industry [1]:

- Teleoperation software for the Westinghouse ROSA III robot, used for maintenance operations inside steam generator channel heads in pressurized water nuclear plants.
- An IRV vehicle used to search for and retrieve fallen objects inside nuclear plant primary circuit pipes.
- A TRON system design for inspection of lower PWR vessel internals.

Despite their differences, these systems share some key characteristics in terms of their control, and they can therefore be relatively easily developed using the same architecture. The shared characteristics are:

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

None of above characteristics apply to the new developments considered in the EFTCoR project [2] which the DSIE is currently working on. The EFTCoR system comprises a family of teleoperated systems whose mission is to retrieve and confine paint, oxide and marine adherences from ship hulls. In this case:

- Working environments are not fixed, given the great variety of ship types, the number of different areas of a given ship and the differences among shipyards.
- Systems need to have a high degree of autonomy.
- Different systems may have to work cooperatively at the same time.

Because of these new characteristics, the original architecture cannot be used for the EFTCoR robots. However, the use of a common architecture for all developments is extremely useful, in that it allows rapid development of systems and the reuse of a wide variety of components, thus saving time and money. For this reason, the DSIE research group is working on a new architecture that takes these new characteristics into account and can be used for development of the robots considered in the EFTCoR project. This paper summarizes the main characteristics of this architecture and is structured as follows: section 2 gives a brief description of the main parts of the EFTCoR system; section 3 presents and justifies the methodological approach adopted; section 4 describes the limits of the system considered and

the main issues for definition of the architecture. Section 5 describes the architecture, and section 6 summarizes the conclusions.

2. The EFTCoR system

The EFTCoR system is a highly ambitious teleoperated platform for non pollutant ship hull maintenance operations in different working areas: from large dry docks free of obstacles to areas full of obstacles such as synchrolift systems. Five subsystems have been identified which cooperate to provide the required functionality. A brief description of these will give us an idea of some of the general requirements to be reflected in the architectural design:

- The **Teleoperation Console**. The teleoperation platform, which is fed with CAD data from the hull being worked on and the process parameters, is able to control and coordinate up to ten robots to optimum quality standards in order to minimize resources and operation time. The teleoperation terminal shows the status of the robots to an operator, who can remotely carry out blasting operations. In fully-automated operational mode, control subsystems use information from the Vision System to complete blasting operations. They provide a set of commands and allow the teleoperation subsystem to move the robot when the operating mode is selected.
- The **Vision System** gives the operator a real-time video image of the surface on which the cleaning head is positioned. Subsequent versions of the vision system will master the blasting operation by calculating the robot trajectory; they will also test resulting blasting quality. The Vision System will provide on-line automated path planning for spot working and assessment in quality control of the blasted surface. It will then communicate with the robot control subsystem and the monitoring system.
- The **Monitoring System**. This system is external to the teleoperating system. The monitoring system considers the scheduling information for each cleaning task. The

system will communicate with the teleoperated system by means of wireless technology. The operator will query and notify the maintenance task data produced.

- The **Positioning System**. Its purpose is to move the cleaning head up to or away from the hull surface. The positioning system may comprise primary and secondary positioning systems: the primary positioning system can be a large crane used to reach all the hull areas, and the secondary positioning system may be a robot or a manipulator capable of covering a certain area. The main reasons that led us to consider differentiated primary and secondary positioning systems were: (1) the difficulty of finding a large positioning system that meets reach, load, precision and controllability requirements at the same time; (2) the need for different positioning systems depending on the size and shape of the hull and the part of the hull to be cleaned; and (3) the existence of primary positioning systems for surface blasting that can be reused for spotting if a controllable secondary positioning system is attached to them. Depending on the nature of the tasks to be performed and on the characteristics of the areas to be treated, there could in principle be configurations where only one of these positioning systems is required. The secondary positioning system should position the cleaning head over the area to be cleaned with reasonable speed and precision. The secondary positioning system is the first candidate for automation, so it is essential that it can be operated as a robot.
- The **Cleaning System**. This system is composed of three primary parts: the blasting head (injection unit), the aspiration unit and the cleaning head. The cleaning head consists of a pan&tilt head that guides the blasting hose and allows the incident angle, air pressure and grit feeding to be adjusted to control the blasting operation. The assembly is enclosed by specially designed shrouds around the blasting heads to "seal" the units to the surface being cleaned, thus preventing dust emissions. Flexible contact between the seal and the hull is achieved by the combined use of air springs and adjusting springs. The waste

handling and recycling system eliminates the residues produced by hull blasting, allowing reuse of the grit material and adequately packaging and disposing of other wastes.

3. Methodological approach

Although many robotics architectures can be found in the literature [3], it is more difficult to find examples of a development process for defining reference architectures in the field of robotics. In our proposal to arrive at a reference architecture for service robot control applications, we followed the Architecture Based Design Method (ABD) [4] and completed 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) may 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 general rather than concrete functionality is the issue, because *the success or failure of such architectures depends on their ability to deal with the variability among the systems of the considered domain*. In this sense, use cases may be very relevant to one system and not very relevant to others. Moreover, at the level of abstraction required to deal with the variability of the systems, concrete use cases cannot be properly defined. For this reason, we have adopted another methodological approach: ABD.

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. ABD is based on:

- Functional decomposition of the problem based on the concepts of low coupling and high cohesion and on knowledge of the application domain.

- 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 decomposing the system into subsystems apply to decomposing the subsystems in other simpler subsystems.

ABD offers as a final model a conceptual view of the architecture that identifies the main subsystems and their relationships, which are described in terms of architectural styles and design patterns. Hofmeister et al [5] propose another architecture-oriented development method, which can be superimposed on ABD in the initial stages. The approach of these authors is interesting because it includes the notions of ports 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 to express such components, ports and connectors. Hofmeister's approach also makes it easier to establish the connection between the conceptual components and their implementations.

4 Domain characterization. Teleoperated service robots.

Service robots are mechatronic systems, usually designed for a concrete application that may be extended to a new functionality in the course of time. They can differ widely from a physical point of view, but they normally use similar software and share many common components, both logical and physical. The first step in defining the functional and quality

requirements that will inform the design of the architecture is to characterize the application domain. In our experience, the main features to be considered should be the following:

- A high degree of specialization and hence high variability of functionality and physical characteristics.
- Different combinations of vehicles, manipulators and tools.
- A large variety of execution infrastructures, including different kinds of processors, communication links and man-machine interfaces.
- A large 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.
- Varying degrees of autonomy, from fully operator-driven systems to semi-autonomous robots.
- Presence of hard real time requirements.
- Hardware- versus software-intensive implementations with all imaginable intermediate cases.
- And last but not least, safety is nearly always a main concern.

Considering the differences among systems as noted above, a central objective of the architecture must clearly be to deal with such variability. A more precise analysis of the differences among systems [8] reveals that most of them relate not to the components of the system but to the interactions among the components. Therefore, when designing the architecture the following points should be borne in mind:

- Very different instances of the architecture should be able to share the same *"virtual"* components.

- The designer should adopt policies that allow a clear separation between the components as such and their patterns of interaction.
- The implementation of such virtual components may be software or hardware; it is highly advisable that such components can be COST.
- It should be possible to derive concrete architectures for both deliberative (operator-driven) and reactive (autonomous intelligent) systems.

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

5. Architecture overview.

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

Once it has been defined how the components must or may be assembled, the second step is to define what components there are to be. The third architectural driver identified in the

previous section states that the components may be implemented by software or hardware, and it is highly advisable that such components can be COST. To achieve this, it is necessary to define the typical components of systems of this kind, which can be identified at different levels of granularity. At the lowest level are the actuators and sensors. A level up are the controllers for simple actuators (for example a motor controller). A further level up are 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 acquired on 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 linkage between the virtual component and its implementation can be achieved using the Bridge pattern [9].

To define virtual components the architecture identifies four levels of granularity and adopts the *hardware abstract layer* notion 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 make it possible to define libraries of components and interchange both hardware and software implementations (perhaps commercial) of the devices with minimum impact.

The last architectural driver identified was the possibility of deriving concrete architectures for both deliberative and reactive systems. For this purpose, the autonomous or programmed behavior has to be separated from the operator driven behavior, 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. However, our approach separates these concerns in a different way. As in the CLARAty architecture, the highest levels of intelligence can directly access the lowest level components: *the intelligence is a client of the functionality*. However, unlike CLARAty, where some autonomous behaviors can be added to the functional layer, in our approach the intelligence of the system is completely separate from the functionality.

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

These layers are called *hardware abstract layers* because the components defined within them may be (and frequently are) implemented in commercial hardware. The simplest components modeled 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 limit switches associated with a given joint. The actuators model the simplest active elements, for example a motor.

SUCs (Simple Unit Controllers) are the components defined at the second architecture layer. The SUC components model control over the actuators and collection of data from the 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 **controllerControl** port), the information received from the sensors that describe the state of the actuator, and its own control policy. This policy is an interchangeable part of the SUC. For example, the **ControlStrategy** of a given joint may be a traditional control (PID) or may be exchanged for a fuzzy logic strategy. SUCs usually need to satisfy hard real time requirements and are therefore generally implemented in hardware. When they are implemented in software they tend to impose severe real time requirements on operating systems and platforms.

At the third level of granularity is the Mechanism Unit Controller (MUC). The MUC component models control over a whole mechanism (vehicle, manipulator or end effector). As figure 3 shows, the MUC is a logical entity composed of an aggregation of SUCs plus a Coordinator responsible for coordinating their actions in accordance with the commands and information that it receives and their own coordination strategy. This strategy is an interchangeable part of the SUC; for example, the coordination strategy of a given manipulator may be a particular solution for its inverse kinematics, the coordination strategy for a given vehicle may be a particular navigation strategy, and so on.

Although the architecture defines the MUCs as relational aggregates, they can be 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, although MUCs may be implemented by hardware or

software, they are frequently commercial motion control cards that constrain the range of possible commands to their internal components. COSTs limit the flexibility of the approach, in that COSTs do not always provide direct access to either their inner sub-components or their inner state.

Finally, the architecture defines the RUC (Robot Unit Controller) component at the fourth layer. The RUC component models control over a whole robot, for example a robot composed of a vehicle with an arm and several interchangeable tools. As figure 4 shows, an RUC is an aggregation of MUCs with a global coordinator that generates the commands for the MUCs and coordinates their actions in accordance with the orders and the information that it receives and with its own coordination strategy. Such a strategy is an interchangeable part of the RUC. For example, the **CoordinationStrategy** of a robot comprising a vehicle with a manipulator may be a generalized kinematic solution that takes into account the possibility of moving the vehicle to reach a given target. Like MUCs, RUCs are logical components that can take the form of physical components depending on concrete instantiations. In general, the RUC is quite a complex component that comprises hardware and software components and can have a large variety of interfaces depending on the complexity of the controlled system.

Having defined SUCs, MUCs and RUCs, it would seem 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 RUCs. There is a good reason for this. The *"usual intelligence"* required to control a joint or mechanism which is an assembly of joints or to teleoperate a robot which is a combination of various mechanisms is limited, is well-known and can be encapsulated in reusable components. The intelligence required to work cooperatively usually demands a more flexible approach. This also goes for some missions

involving SUCs, MUCs and RUCs, and likewise algorithms for collision avoidance or navigation systems for vehicles. It is very difficult to define a component that will encapsulate “intelligence”. If a system or component is capable of being intelligent and taking non-trivial decisions, it will normally be complex enough to have a defined architecture of its own (for example, an artificial vision system able to determine obstacle-free paths). In that case, the approach should be different: Do not impose a structure on the *intelligent* components but find a way to integrate them into the system.

5.2 Adding autonomous behavior.

The SUC/MUC composition produces a hierarchical architecture where the decisions flow from the top down and the information flows from the bottom up. This architecture sits well with operator-driven systems, where autonomous behavior does not exist or is confined to some hardware safety actions. It also sits well with systems where the reactive or autonomous behavior responds to simple rules that can be added to controllers and coordinators so that the latter, following these rules, can take decisions and notify them to the upper level controller or coordinator. However, there are systems where the autonomous behavior is anything but simple. In such cases, the *intelligent component* needs to integrate more information and access more functionalities than those embedded in a given component. The approach in that case (see figure 5) is to *superimpose* the “*intelligent*” autonomous behavior and the operator-driven behavior while providing the means for integrating both and resolving the potential conflicts. This approach does not entail any change in the components defined so far, but simply new command sources for them. These sources are constituted by new components that have access to the global information system and are capable of deciding what to do on the basis of programmed rules, algorithms or heuristics.

Every component of a given layer can access the information and control ports of components of lower layers. In this sense, every component of a given layer is an intelligent component for the layer below it, for example from the point of view of a MUC, no matter whether the commands come from the coordinator of the RUC that controls it (see figure 5), from the operator or from some of the intelligent components defined on a level above the RUCs. Since 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 external to the component. 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 encapsulates (or even the **arbitrator** itself) can vary from system to system, during the life of a given system or even at different stages in the functioning of a system. The concept of an arbitrator derives from the notion of a *composition filter* [13] and is strongly connected to the need to separate functionality from the patterns of interaction among components.

This approach is highly flexible and makes it possible to integrate intelligence that is directly concerned not with the missions of robotic devices but with management of the application as regards 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 puts them together with a component-oriented approach. This approach focuses on the definition of a common component framework that allows the definition of components that can be reused in different systems and integrated in intelligent systems capable of driving robot behavior. Our main sources of inspiration have been 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 paint, oxide and marine adherences from ship hulls (see figure [6]). Presenting as it does a wide variety of behaviors and degrees of complexity, this family of robots is an excellent test bench for the architecture.

Our experience using the architecture has been satisfactory; however, we would note two major challenges in this respect:

- There is not enough support to express the component abstractions and model their interactions.
- Also, there are no well known techniques to cope with the variability of components from one instantiations to another.

These challenges can be met by the PRISMA approach. We are currently working on this with the Technical University of Valencia (Spain) within the framework of a nationally funded (CICYT) research project, DYNAMICA, ref. TIC2003-07804-C05. A possible first step is to use the PRISMA language to define the components and the layered architecture. A possible second step is to consider changes in the interactions among these components.

7. References

1. A Iborra., J.A. Pastor, B. Álvarez, C. Fernández and J.M. Fdez-Meroño. “Operational Experiences using Robotics during Maintenance Services in PWR Nuclear Power Plants”. *IEEE Robotics&Automation Magazine*, **Vol. 10**(4), (Dec, 2003). pp. 12-22.

2. "Environmentally Friendly and Cost-Effective Technology for Coating Removal (EFTCOR)". *Fifth Framework Programme,* European Community, Subprogram Growth ref. GRD2-2001-50004, (Oct 2002).
3. E. Coste-Manière and R. Simmons, "Architecture, the Backbone of Robotic System", *Proc. of the IEEE international conference on Robotics & Automation*, San Francisco, (Apr 2000), pp. 67-72.
4. F. Bachmann, L. Bass, G. Chastek, P. Donohoe and F. Peruzzi, "The Architecture Based Design Method", Technical Report CMU/SEI-200-TR-001, Carnegie Mellon University, USA, (Jan 2000).
5. C. Hofmeister, R. Nord and D. Soni, "Applied Software Architecture", *Addison-Wesley*. USA, (Jan 2000).
6. I. Jacobson, G. Booch and J. Rumbaugh, "The Unified Software development Process". *Addison-Wesley*, (1999).
7. B. Selic, G. Gullekson and P.T. Ward, "Real-Time Object-Oriented Modeling". *John Wiley and Sons*, New York, (1994).
8. J. Pastor, "Incremental evaluation and development of software architectures for tele-operation system using formal methods". Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales", *PhD Thesis*, Universidad Politécnica de Cartagena, Spain, (2002).
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", *Addison Wesley*, Reading Mass. (1995).
10. J. Pérez, 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, USA, (Nov. 2003), pp. 59-66.

11. The OROCOS project: Open Realtime Control versus Open Robot Control. EURON, available at www.orocos.org (last access in May 2005) .
12. I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, "CLARAty: An Architecture for Reusable Robotic Software," *SPIE Aerosense Conference*, Orlando, Florida, http://www.jpl.nasa.gov, (Apr 2003).
13. L. Bergmans, Composing Concurrent Objects, *PhD Thesis*, University of Twente, The Netherlands, (1994).

Figure 1: An abstract overview of the proposed architecture.

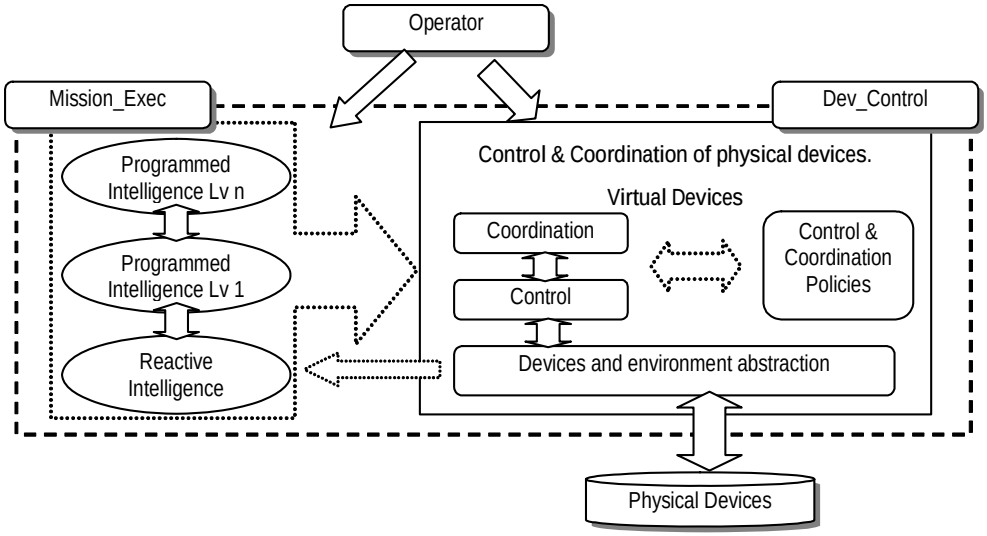
Figure 2: SUC: Simple Unit Controller.

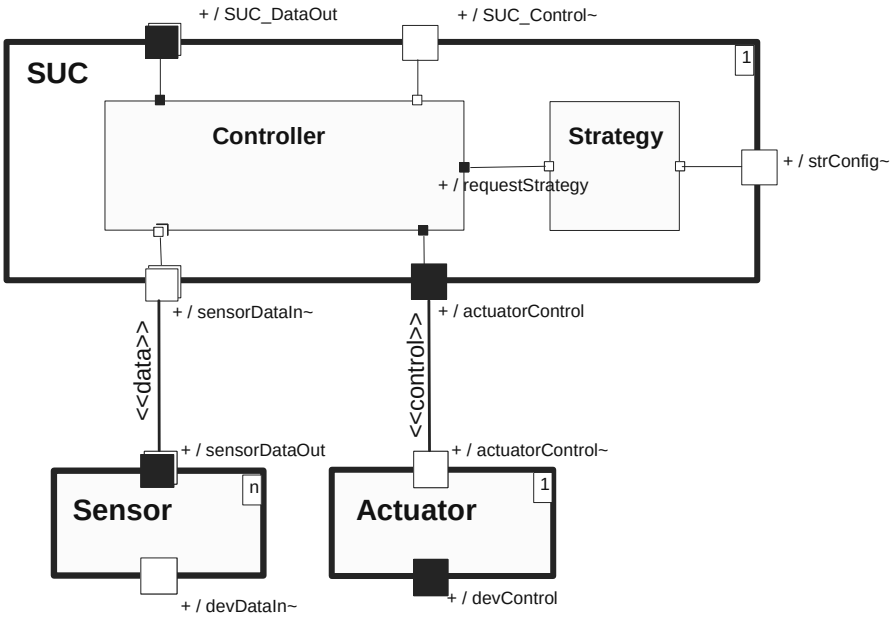
Figure 3: MUC: Mechanism Unit Controller.

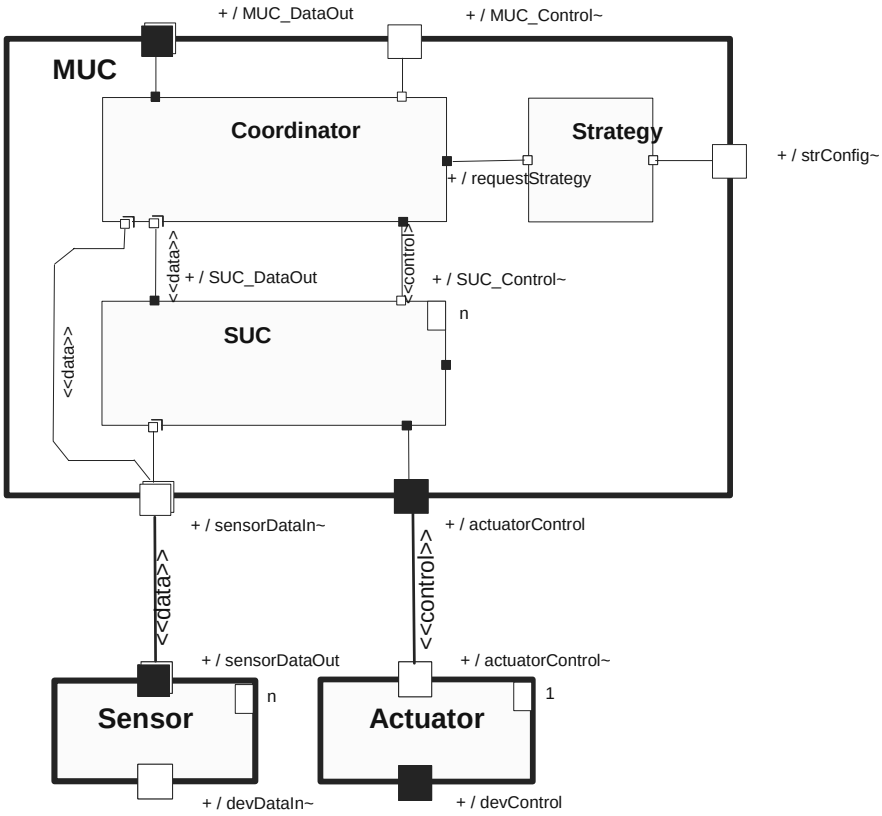
Figure 4: RUC: Robot Unit Controller.

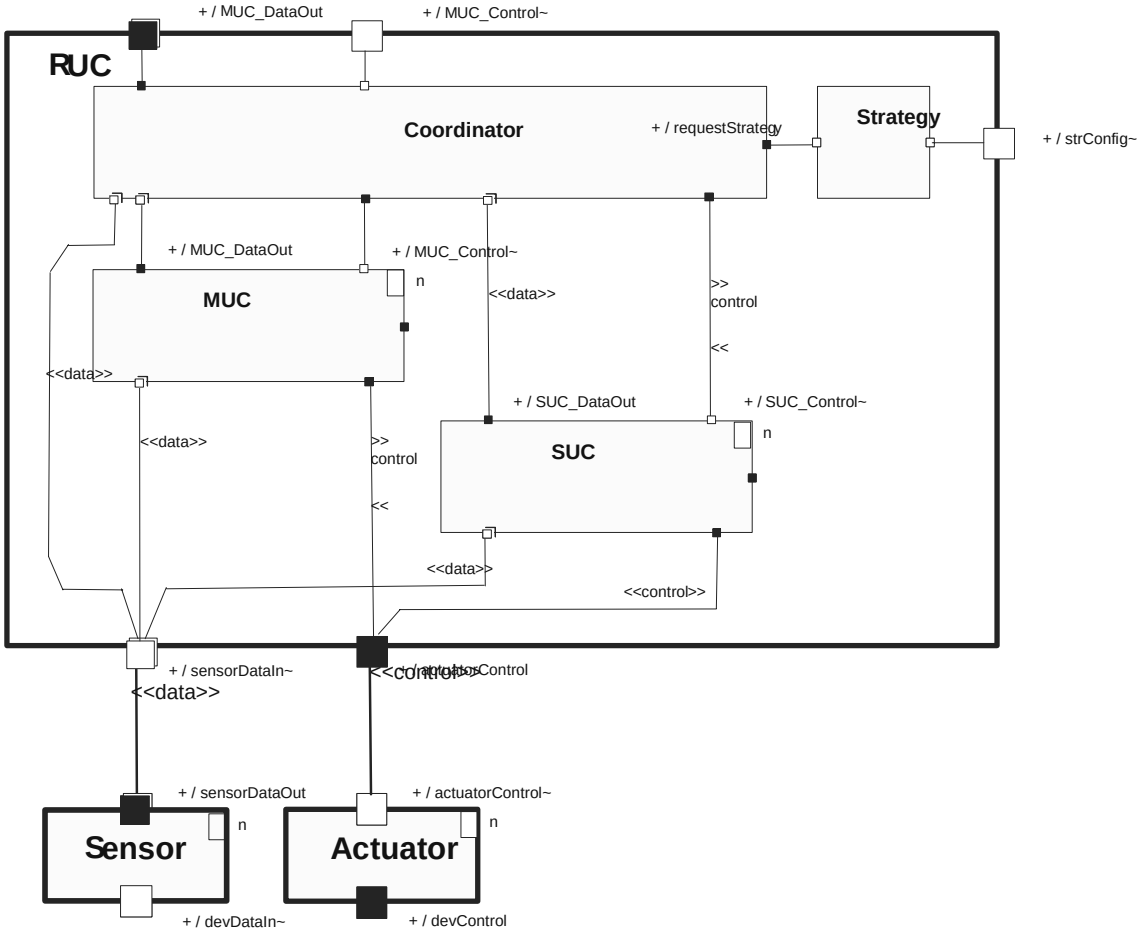
Figure 5: Superimposition of operator-driven and autonomous behavior.

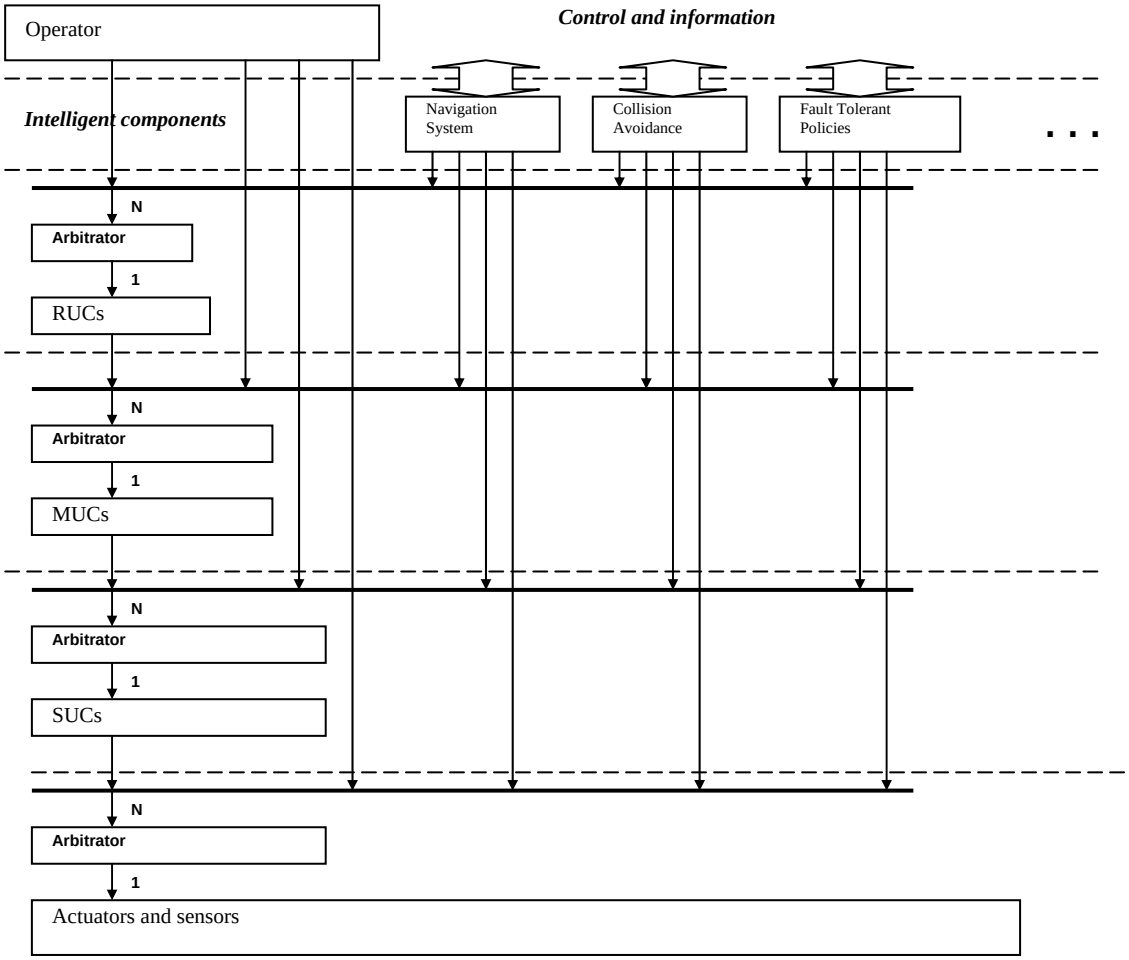
Figure 6: Three prototypes (cherry-picker model, elevation platform and mobile vehicle, respectively) of the family of robots and a ship awaiting repair.

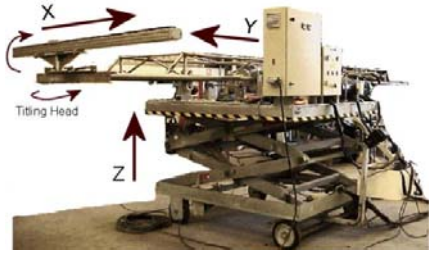

Operator
Mission_Exec
Dev_Control
Programmed Intelligence Lv n
Programmed Intelligence Lv 1
Reactive Intelligence
Control & Coordination of physical devices.
Virtual Devices
Coordination
Control
Control & Coordination Policies
Devices and environment abstraction
Physical Devices


+ / SUC_DataOut
+ / SUC_Control~
SUC
1
Controller
Strategy
+ / strConfig~
+ / requestStrategy
+ / sensorDataIn~
+ / actuatorControl
<<data>>
<<control>>
+ / sensorDataOut
+ / actuatorControl~
n
Sensor
1
Actuator
+ / devDataIn~
+ / devControl


+ / MUC_DataOut
+ / MUC_Control~
MUC
Coordinator
Strategy
+ / strConfig~
+ / requestStrategy
<<data>>
+ / SUC_DataOut
<<control>>
+ / SUC_Control~
n
SUC
<<data>>
+ / sensorDataIn~
+ / actuatorControl
<<data>>
<<control>>
+ / sensorDataOut
+ / actuatorControl~
n
1
Sensor
Actuator
+ / devDataIn~
+ / devControl


+ / MUC_DataOut
+ / MUC_Control~
RUC
Coordinator
+ / requestStrategy
Strategy
+ / strConfig~
+ / MUC_DataOut
+ / MUC_Control~
n
MUC
<<data>>
>>
control
<<
<<data>>
+ / SUC_DataOut
+ / SUC_Control~
<<data>>
>>
control
SUC
n
<<
<<data>>
<<control>>
+ / sensorDataIn~
<<data>>
<<control>>
+ / sensorDataOut
+ / actuatorControl~
Sensor
n
Actuator
n
+ / devDataIn~
+ / devControl


Operator
Control and information
Intelligent components
Navigation System
Collision Avoidance
Fault Tolerant Policies
. . .
N
Arbitrator
1
RUCs
N
Arbitrator
1
MUCs
N
Arbitrator
1
SUCs
N
Arbitrator
1
Actuators and sensors


X
Y
Titling Head
Z