TOWARDS A GENERIC SOFTWARE ARCHITECTURE FOR A SERVICE ROBOT CONTROLLER

B. Álvarez, F. Ortiz, A. Martínez, P. Sánchez, J.A. Pastor, A. Iborra

Universidad Politécnica de Cartagena,
División de Sistemas e Ingeniería Electrónica
Campus Muralla del Mar,s/n. Cartagena, E-30202, Spain
E-mail address: Barbara.Alvarez@upct.es

Abstract: Embedded computer control systems rely on software for performing their functions. These systems are essentially real-time systems and traditionally the research on these systems has been focused on functionality and performance. However, quality attributes as modifiability, portability, etc. are very important for developing high quality computer control software. One way of rising high quality is based on the study of software architectures. In this work, we present the followed process to obtain a reference architecture of a robot controller for a teleoperated service robot. To achieve this goal, we have followed an architecture-based development process using UML, a standard notation for the design of static and dynamic properties of systems. *Copyrigh ã2002 IFAC.*

Keywords: Architectures, computer control, robot control, teleoperation, real-time systems.

1. INTRODUCTION

Embedded computer control systems heavily rely on software for performing their functions. Software manages the operations of underlying hardware and makes possible to achieve other advanced functions such as diagnostics, human-machine interface, etc. However, due to the fact that there are physical processes under control, embedded control software differs from general-purpose software in several aspects:

- It must support the execution of automatic control algorithms as well as other related functions. This means that the software together with hardware must support both logical determinism and temporal determinism. In other words, these systems are essentially real-time systems, where a too late completion of control task or a missed deadline may impact control performance and stability negatively, and may lead to total system failure.
- It must be at least as dependable as mechanical solutions. This means that careful considerations with respect to fault avoidance and fault tolerance at different levels of computer hierarchy are needed.

Traditionally, the research on computer control systems has been focused on these aspects: functionality, performance and dependability. However, control systems become increasingly more complex and widely used, and other characteristics become critical as well. Currently, quality attributes as modifiability, portability, reusability, etc. are very important to develop high quality computer control software and reducing the cost of products. The consideration of software architecture contributes to developing high quality systems. The architectural design is critical for the success of system development because the qualities of large software systems are largely determined by system structuring (Bass, *et al*., 1998).

Robot teleoperation systems are a typical area of application of control computer software (figure 1). In particular, our work is centralized for service robots. In these systems, an operator is in charge of monitoring and operating the robot according to the information provided by the teleoperation system. This system receives commands from the operator and performs the corresponding actions for executing them. For this purpose, it communicates with the remote robot control unit, which physically actuates on the robot to move it. The robot control unit makes some sensing from the robot in order to evaluate its global state and sends this information to the teleoperation system, which uses it to represent graphically the state of the robot and ensure the correctness of its behaviour.

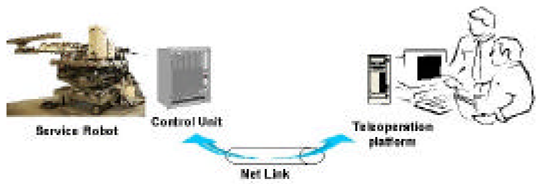


Fig. 1. Teleoperated service robot system

Different tools are attached to the robot so as to perform the maintenance operations. The tools are operated in a similar way to the robot.

In Alvarez, *et al.* (2001) a reference architecture for teleoperation platforms (reused for different applications) is described. In this work, we present the process followed to obtain a reference architecture for the control unit (ROC). In the next section, basic characteristics of these applications are described as well as desired quality attributes. In section 3, some design aspects are commented. An important aspect for managing software complexity is the description of the system structures under consideration through a formal language. As section 4 shows, we have used UML for requirements elicitation and designing of the system architecture. Section 5 describes a particular application of this architecture and finally conclusions are included.

2. SYSTEM REQUIREMENTS

One conceptual illustration of the remote control unit for service robot applications is given in figure 2, partially based on a conceptual description of automotive control systems given by Axelsson (1999). Discrete time signals are shown with dashed lines and continuous signals with solid lines, e.g., y(k) is the discrete-time value of y(t) at the sampling instants (i.e., t_k). From a functionality point of view, software at the application level can divided into the following functions:

- *Motion control functions* are needed in order to actuate on the mechanical parts according to a given reference (e.g., acceleration, speed and position) even in the presence of unexpected environmental disturbances. The control is commonly based on the concept of feedback or closed loop control. The basic operations of feedback control include sampling, computing, and actuating. For the ease of control design, the states of the system under control are sampled or discretized periodically.
- *Discrete event control functions.* These functions are responsible for determining the sequence of actions that should be performed to get desired behaviours. They are often implemented as state machines, which may run periodically or not. The desired timing mainly depends on the discrete dynamics under control.
- *Mode switch function.* This function is responsible for determining and managing the local and teleoperated system operational mode. Inputs to the control unit come both from the teleoperation system (teleoperated mode) and from an operator close to the control unit (local mode).

These functional requirements refer to requests that can be mapped into and implemented by one or several subsystems. Functionality itself is not sensitive to system structures. In contrast, other functional requirements (as performance and reliability) are concerned with system operations and depend on the architectural design. The interdependence of system constituent units must be properly structured to achieve the following non-functional requirements or non-run-time quality attributes:

- *Modifiability.* The system must allow making changes such as extending capabilities for working with several controllers for each axis of the same robot or different robots.
- *Portability.* The system must be implemented on different target platforms. This can be ensured through introducing a portability layer that encapsulates all platform specific considerations.
- *Reusability.* Different parts of the system need to be reused for future applications. Consequently, generic components must be separately developed to work together.
- *Interoperability.* The system must work with teleoperation systems. For this reason, communication subsystems must guarantee the completeness and the consistency of their interface specifications.

In order to reach these functional and non-functional requirements, an architecture-based development process has been considered, as the next section shows.

3. DESIGNING ARCHITECTURE

Traditionally, robotic software programs have followed the semantics defined by control design, and coupling and cohesion have been considered important structuring criterions. It is easier to keep consistency and completeness of information with a structured design approach. In this work, coupling and cohesion criteria (Peterson and Stanley, 1994) have been considered for mapping functional requirements into components.

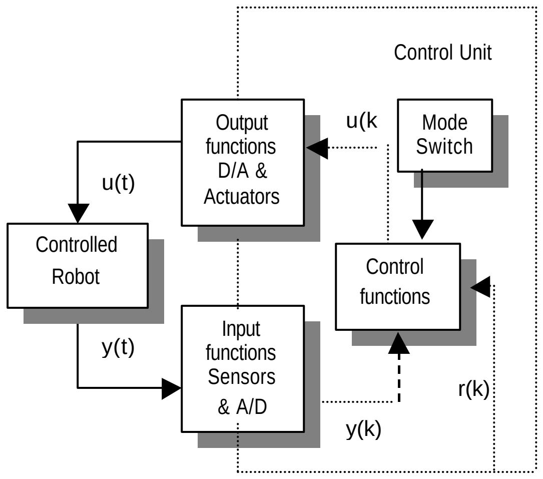


Fig. 2. Conceptual illustration of controller.

Moreover, for domain systems, a reference architecture represents a domain specific way of structuring software systems through decomposing the problems into parts and **their relationships**, and mapping the decomposition results onto software units and **their interactions**. It is shown that the qualities of a software system are largely determined by its structures. For this reason, in order to determine the structures of the system, functional and non-functional requirements as well as architecture styles and patterns need to be considered. Architecture styles denote well-known ways of structuring. This issue can be found in the architecture-based development process proposed by Bass and Kazman (1999). This process is based on a top-down and iterative process and it can be characterized by the following steps:

1. *Developing subsystems for the requirements*: A set of subsystems is generated from functional and non-functional requirements, based on architectural styles and pattern as well as experiences. The considered choice is such that satisfies more requirements simultaneously.
2. *Determining an actual architecture:* These subsystems can be seen as components in a larger subsystem. Thus, functional view is described and transformed into a process view based on the considerations of parallelism.
3. *Validating the solution:* The architecture solution is validated using the quality scenarios, e.g., change scenario for modifiability, use scenario for performance, etc.

4. ARCHITECTURAL DESCRIPTION

One of the most important issues around software architecture is the description of the system structures under consideration. It is the basis for all design activities including comprehending, communicating, analysing, trading-off, as well as for modification, maintenance, and reuse. Similar to other models, the description can be based on mathematical, textual, or graphical notations, but in order to manage the complexity of a system, a complete architecture description should be divided into multiple views.

Often, each architectural view includes a set of models that describes one aspect of a system. One well-known and widely used approach to multi-viewed architectural description is the 4+1 *View Model of Architecture* proposed by Kruchten (1995). This model has also been adopted in the development of *Unified Modeling Language* (UML) (Booch, *et al.*, 1998; OMG, 2001). UML has emerged as a standard notation for conceptual modeling using the object-oriented paradigm. Taking into account the benefits of blending object-oriented concepts with concurrency aspects, it is essential to successfully use

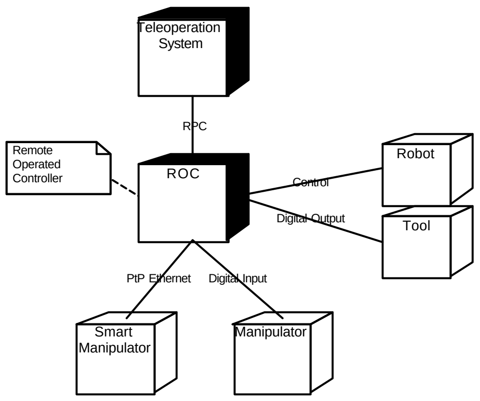


Fig. 3. Deployment diagram.

the UML notation when designing distributed and real-time applications. The UML notation provides several diagrams that allow us to represent static and dynamic properties of real systems and integrate them following the previous 4+1.

In order to obtain a reference architecture we have followed the COMET methodology (Concurrent Object Modeling and Architectural Design Method with UML) proposed by Gomaa (2000). It is a design method for concurrent applications based on the USDP (Unified Software Development Process) and the spiral model of Boehm. Starting from the system Use Cases, a static and dynamic design of the classes in the architecture can be derived until reaching the final implementation. Our goal is to reach a reference architecture for the design of control units in teleoperated service robots. In this kind of systems, a teleoperation system sends commands to the robot control unit, which controls the electromechanical elements composing the robot. In turn, this control unit returns the state of the robot to the teleoperation system. In figure 3, a possible deployment diagram of the whole system is presented, where different modules – nodes are included.

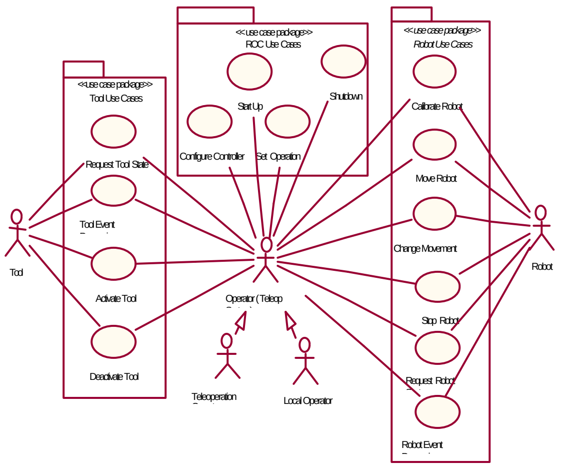


Fig. 4. Use Cases diagram.

Following the development process, once the requirements of the system are collected (functional and no functional), we create a detailed tabular specification of the system functionality. It is divided into categories where attributes (as time response, fault tolerance, etc) are included. From such specification, the use cases of the system are extracted (figure 4).

A system context class diagram (figure 5) is derived from use case diagram by considering the actors and which devices they utilize to interface with the system. In that diagram, the ROC object is composed of several other objects, and receives inputs from external objects and actuates over them. Several of these objects are device interface objects that interface to external I/O devices, mainly the robot sensors and actuators. It also receives inputs from the manipulator and interacts with the teleoperation system.

4.1 Discovering classes.

After the previous step, every Use Case is studied in order to obtain the objects that take part in it and the exchanging messages between objects. This is the most complicated phase in the development process and it needs a bug creativity effort from the designer. Several collaboration diagrams are consequence of this study. Once the different objects of the system are extracted from collaboration diagrams, the classes of the system can be proposed as a generalization of objects.

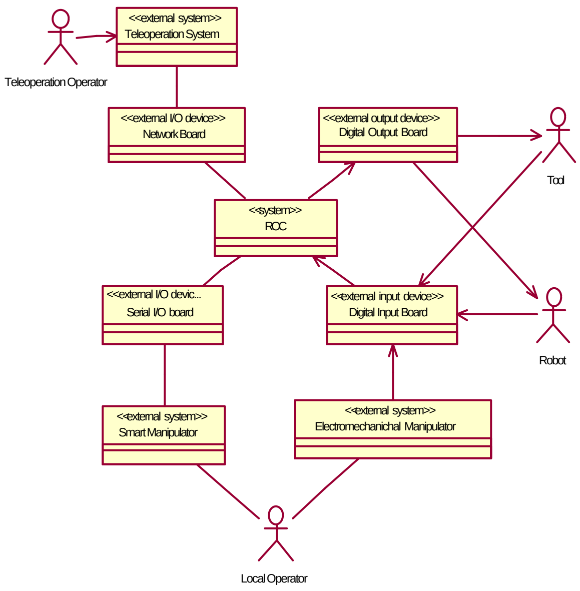
One of the main objects composing the control unit is the *Joint_Controller*, which has to implement several methods as *move_to*, *stop*, etc. Therefore, the control architecture is based on the class *Joint_Controller*, defined as interface or abstract class (figure 6). Each controller could be different, so it will be an implementation of *Joint_Controller***,** giving the same interface to the rest of the system. It will be as many controllers as joints the robot has, one for each joint. Each of them implements its own control algorithm, which could be only software or an interface to a hardware control board. It is clear then, that if a coordinated movement is needed, there should be a coordinator of controllers, as shown in figure 7. This figure represents the class diagram of the architecture. The class *Joints_Coordinator* offers different basic methods of coordination between joints.

The class *Tool_Controller* is similar to *Joint_Controller*, excepting the object to control. In the last case it is dedicated to the tool, implementing a different controller for each possible tool that could be managed by the robot. The same remark could be done for *Tools_Coordinator.*

The process coordinator establishes the highest level in this architecture. Although the domain of the application is teleoperated service robots, there are several process that can be performed in an autonomous manner. *ProcN_Coordinator* implements one of these processes. For each one there should be a different Process Coordinator, changing in running time depending on the process.

4.2 Concurrent tasks structuring.

In any real-time concurrent system is necessary to establish the active objects, that is to say, the concurrent tasks in the application. During the task - structuring phase, the task architecture is developed.



Teleoperation Operator
<<external system>> Teleoperation System
<<external I/O device>> Network Board
<<external output device>> Digital Output Board
Tool
<<system>> ROC
<<external I/O devic... Serial I/O board
<<external input device>> Digital Input Board
Robot
<<external system>> Smart Manipulator
<<external system>> Electromechanichal Manipulator
Local Operator

Fig. 5. Context class diagram.

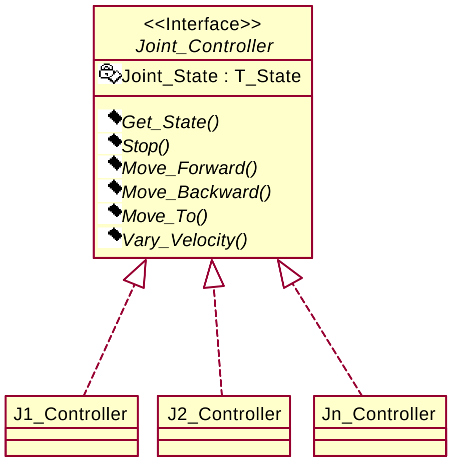


Fig. 6. *Joint_Controller* implementation diagram

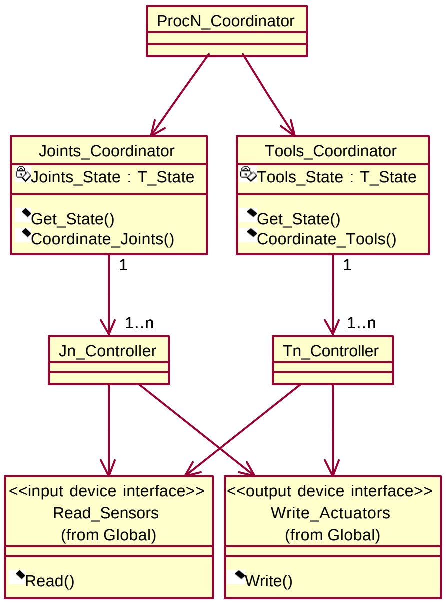


Fig. 7. Proposed architecture class diagram

As a consequence, the system is structured into concurrent tasks and the task interfaces and interconnections are defined. To help determine the concurrent tasks, task-structuring criteria is provided by COMET to assist in mapping an object-oriented analysis model of the system to a concurrent tasking architecture. For instance, depending on the characteristics of the I/O devices (asynchronous, passive, etc), one or more task will be chosen to read them. That is to say, if the sampling rate of two passive devices differs we should chose two different tasks, but if it is similar, it could be simplified in one task depending on the computational necessities of the system.

5. AN APPLICATION OF THE ARCHITECTURE: GOYA SYSTEM

Recently, the obtained architecture has been implemented for GOYA (Ortiz, *et al.*, 2000) system: a teleoperated system for blasting applied to hull cleaning in ship maintenance (figure 8). The approach taken for the teleoperation platform is based on the generic architecture described by Alvarez, *et al.*, (2001). In this case, the teleoperation platform is a O2 Silicon Graphics workstation. The architecture for the control unit have been implemented over an industrial PC with Linux.

There are control systems that have not such stringent safety and time requirements that justify the use of real time operating systems. A failure in the system execution or time requirement sporadically missed does not imply an immediate threat. This is the case for a numb er of control applications which are supervised or teleoperated by human, such as robots for ship hull blasting. In these cases, the robot speed is low and in the case of failure in the computer control system, the operator has the ability for manually stopping the robot.

In this case, the architecture has been implemented of the following manner. The Goya robot has three freedom degrees (xyz) and one tool. Then, four controllers are necessary, one for each freedom degree and one for the tool. In figure 7, a class diagram is shown with *Jn_Controller* and multiplicity 1..n; the implementation in an object diagram for this particular robot leads to: J1_Controller for the elevation platform (z-axis), J2_Controller for positioning arm (y-axis) and J3_Controller for tool positioning cart (x-axis) mounted on the titling head.

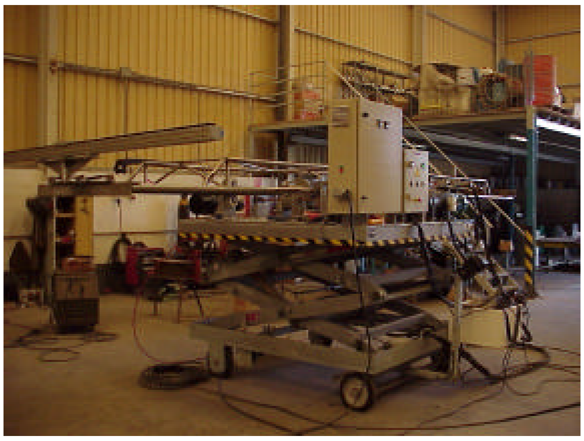
In this robot has only one tool so the multiplicity of *Tn_Controller* will be 1: T1_Controller for the blasting tool. Over this joints controllers there is a coordinator object (Joints_Coordinator) that is required to coordinate movements. This abstract class is implemented with the appropiate procedure *Coordinate_Joints* for this robot. The Tools_Coordinator is not necessary in this application because we have only one tool, but finally it is implemented to respect the architecture, offering the same interface to the rest of the application in prevention of later modifications and improvements of the robot and antic ipating possible tool interchanging.

The top layer is the Process_Coordinator. In this application, an object has implemented a state machine performing the automatic sequence for blasting a complete hull panel. The interface offered by Process_Coordinator is the same for any layer that accesses to the controllers , so every control order, not only coordinated ones, but even control for individual joints pass through the Process_Coordinator. The same could be said for Joints_Coordinator. We have created layers with the same interface to the upper layer.

Ada 95 has been employed as programming language and the communication with teleoperation platform is based on the use of GLADE (Tardieu, *et al.*, 2001) which is an implentation of DSA (Distributed System Annex) of Ada.

6. CONCLUSIONS

The success of reusing a reference architecture for teleoperation platform in several applications has motivated us for developing an architecture for the robot control unit. A key factor for a successful construction of software systems is the use of patterns in architecture design. In order to reach this goal, the architecture-based development process



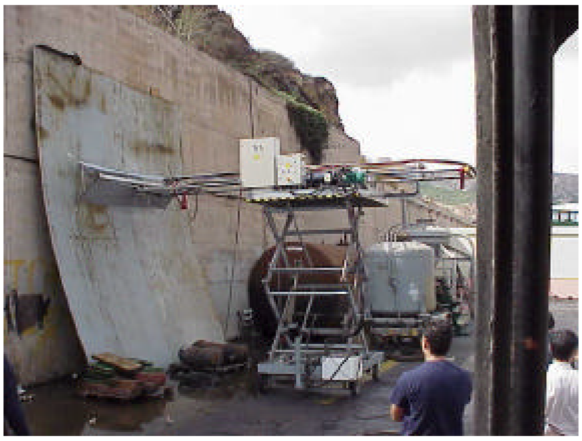


Fig.8. GOYA system.

proposed by Bass and Kazman (1999) have been followed. Furthermore, UML notation provides a semi-formal description that promotes rigorous properties verification with tool support. Besides, the COMET development process, based on USDP, is followed to analyse and design the system with the UML notation.

Recently, the above architecture has been implemented for GOYA system. In this case, a real-time operating system has not been necessary, although the architecture can be implemented in other platforms and over other operating systems.

ACKNOWLEDGEMENTS

This work has been supported by the Spanish Government and European Union Programmes for Research: Electrical Power Research Program (project PIE-041049), Technological Actuation in Industry (PAUTA projects 753/95 and 53/96), EUREKA-MAINE program (TRON-EU1565), and GOYA (CICYT-FEDER 1FD97-0823).

REFERENCES

Alvarez B., A. Iborra, A. Alonso and J.A. de la Puente (2001). Reference architecture for robot teleoperation: Development details and practical use. *Control Engineering Practice,* vol.9, pp.395-402.

Axelsson, J (1999). Holistic object-oriented modelling of distributed automotive real-time control applications. In: *Proceedings 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*.
Bass,L., P. Clements, and R. Kazman (1998). Software Architecture in Practice, Addison-Wesley. ISBN 0-201-19930-3.

Bass L., and R. Kazman (1998). Architecture-Based Development. *Technical Report, Carnegie Mellon University*, CMU/SEI-99-TR-007.

Booch, G., I. Jacobson, and J. Rumbaugh (1998). *The Unified Modeling Language User Guide,* Addison-Wesley Pub Co.

Gomaa, H.(2000).,*Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley Object Technology Series (Booch, Jacobson, Rumbaugh), ISBN 0-201-65793-7

Kruchten, P.B (1995). The 4+1 View Model of Architecture. IEEE Software, vol.12, issue:6, pp. 42-50. ISSN: 0740-7459.

OMG (2001) Object Management Group. *OMG Unified Modeling Language Specification (version 1.4)*, 2001. http://www.omg.org

Ortiz, F., A. Iborra, F. Marin, B. Álvarez, and J.M. Fernandez (2000) *GOYA* - A teleoperated system for blasting applied to ships maintenance. In: *3rd International Conference on Climbing and Walking Robots*, Spain.

Peterson, A.S. and Jay L. Stanley Jr (1994). Mapping a Domain Model and Architecture to a Generic Design. *Technical report, Carnegie Mellon University (CMU)*.

Tardieu Pautet, Lauren and Samuel (2001). *GLADE user´s guide.* Technical report version 3.14a. ACT.