Generic modular framework for robotic arm applications

Sergi Hernandez Juan, Josep M. Mirats Tur

Abstract: *Robotic manipulators are becoming a very common tool in modern industries and research laboratories. In order to successfully execute a given task a control software is necessary that sends and tracks appropriate orders to the robot arm. This paper presents the design principles of a general software framework capable to control any robot arm with any set of sensors. A possible implementation of such a general framework is provided together with experiments on a particular robot platform.*

Key words: *Computer Systems and Technologies, Software design, Robotic arm manipulators.*

1. INTRODUCTION

The use of robotic manipulators has been growing up dramatically in the last decades [2]. Common applications range from the assembling and welding of different parts in the automotive industry to the manipulation of unknown objects in space missions [3]. The broad spectrum of applications has lead to a quite big number of different and specific designs for such robotic manipulators. Main differences are found in their architectures: number and type of degrees of freedom (dof), dimensions of its links, actuators or sensors employed.

In order for the robotic device to perform a given task it is necessary to control it, that is basically, to have the capabilities of, first, sending specific orders to the manipulator, in terms of positions or velocities of its final effector, and second, sensing the real obtained position or velocity. Other sensing capabilities may be important depending on the application, such as force feedback. So, in order to implement a given control algorithm for the manipulator to solve a task, it is necessary to control its basic functionalities (moving and sensing) which implies the development of a software platform.

Usually, robotic manipulator and sensors manufacturers supply a set of functions (Application Programming Interface) to interface with the low level features of robots and sensors respectively. Those functions need to be integrated within the full software application used to solve a specific task, which means designing it to specifically interface with those particular APIs. So the application itself is platform dependant and, whenever it would be necessary to change the robot or any of its sensors, it would also imply an important modification of the control program, which is, in general, an error full and time consuming task.

Furthermore, as the computational needs of some sensing devices, such as vision based perception, increase, the need for a distributed control system become more important to keep the desired performance and real time control. If the software application is not designed to be splitted between several computers it will be almost impossible to achieve that, but even in that case, it would be difficult to modify the original communication architecture once designed.

In this paper a modular generic software framework is presented for robotic manipulators. It helps to minimize the time necessary to complete a full distributed control OS independent application when using any robotic arm and set of sensors to perform a given task. The software is designed in such a way that changes in the hardware platform used (as dramatic as using a completely different manipulator for the same task) imply a minimum reprogramming effort. Section 2 introduces the design principles of this software architecture and a possible implementation is presented in Section 3. Section 4 reports some results on using the implemented software framework on a robotic platform and finally, Section 5 concludes the paper.

2. SOFTWARE DESIGN PRINCIPLES

Given a robotic manipulator we want to control it in order to solve a specific task. The control software should be general for any kind of robot arm and set of sensors. To this aim the main considered design principles are:

- Modularity and functional separation.
- Abstraction.
- Distributed capability.
- Real time control.
- Multi-platform.

2.1 Modularity and functional separation

Modularity and functional separation [5,1], together with code implementation dependent principles such as reusability and robustness [6], are the most basic software design principles that can be found in commercial as well as non-commercial software applications. The aim of modularity is to encapsulate all the features of a physical or logical entity into some kind of structure to prevent other entities to directly modify them. Instead, a set of functions is provided for each of these entities to access their attributes. Such set of functions is usually referred to an *interface*, term that will be used all throughout the paper.

Such an *interface* should provide, first, the necessary functions to modify the entities attributes and to retrieve their values, and second, provide those functions required to carry out their specific functionality, such as acquiring data from a sensor or executing a given trajectory with a robot. Furthermore, modular design allows the software designer to limit the required changes whenever a module needs to be modified or extended.

The functional separation main issue is to isolate as much as possible the application itself from the user interface and the underlying hardware. The user interface is the set of functions that allow the interaction between the user and the software application. For this case the functional separation is easily accomplished by providing a complete enough *interface* for the highest hierarchical module, so the user interface (graphical, command line, web based, etc.) just needs to call a function or a sequence of functions from the *interface* to carry out any desired task.

On the other hand, the separation from the low level hardware is much more complicate. It will always be necessary to develop a device driver for any external device that will depend on the link type between the external device and the application (RS-232, Ethernet, PCI, etc.). In order to overcome this problem, each developed device driver should use a virtual communication device (VCD) which will be used for any input/output access to the external device. It works as a placeholder for any real communication device and implements all the features needed to provide an efficient channel of communication: input and output queues for data flux control, asynchronous notification of received data, multithreaded execution and error handling capabilities; but it not includes the specific implementation to actually open, configure, close or send and receive data to and from the external device, which must be defined for each particular implementation.

By using such a VCD it is possible to easily change the way the application interfaces with the external devices, that is, external devices may be connected either directly or remotely without modifying the application. This idea is further developed in section 2.3 where the principle of distributed control is introduced and described.

As outlined at the beginning of this section, these two basic and simple principles apply to any software application, but, in the specific case of robotic applications, new challenges arise which can be handled using other design principles as described next.

2.2 Abstraction

Each physical device (i.e. sensor, motor controller, robot, etc.) has a given functionality which is independent from the particular implementation. So, the corresponding software module has to provide a generic interface which allows any other module to access all its functionality without noticing the implementation details.

For the particular case of robotic arm applications, the set of generic modules correspond to the robot, the force and position feedback devices, the motion controller and each of the motor controllers. The abstraction of the basic functionality takes place in all of these modules, from the low level device driver interface to the high level robot control interface.

For example, a motor control module should provide the same functions to configure the motor motion (absolute or relative motion, motion blending, etc.), to load the motion parameters (position or speed) and to start and stop the motion whatever the motor controller used. A robot module should provide the same functions to carry out any arbitrary trajectories, whatever the kinematic and dynamic models of the robot and the particular feedback devices are.

Therefore, this set of generic modules should provide, first, a generic *interface* to access their specific functionality, second, a basic implementation, common to any platform, and, finally, the necessary communications between the different modules.

All the high level control algorithms need information about the robot's kinematic and dynamic models, the motors controllers and/or the feedback devices, but due to the generic *interface* provided by each module, the control algorithm will be always the same.

Thus, given a set of generic modules, and in order to adapt the software to a specific platform, the only functions that are required to be implemented are those describing the robot's architecture and the ones that interface with the device drivers. Since all the basic control algorithms and inter-module communications are already implemented, the time needed to migrate from one platform to another is highly reduced.

2.3 Distributed capability

As the computational needs of some feedback devices increase, it becomes necessary to split the control application through several computers so as to keep performance and real-time control. Also, due to the absolutely different requirements for each application, the way the control software is splitted should be flexible enough to adapt to different configurations with little or no modification at all of the program.

Given the generic modules introduced in the previous section, the easiest way to split the application is through these modules. Each of them may be local or remote to the users' computer. In the former case nothing has to be done since all the modules functionality is directly accessible through function calls. On the other hand, when a module is executing remotely in another computer several new features must be added to it. First of all, it is necessary to define a communication protocol to both, send the commands and parameters, and receive the operation results. It is required, also, to keep two different versions for each module: one working as a server, waiting for commands and actually carrying out the necessary actions, and the other working as a client, sending the commands and waiting for them to end.

Finally, it's also necessary to define a communication device to handle the data transfer between the host and the client. To this end, the VCD explained in 2.1 can be used given the many different available possibilities: Ethernet in research environments, CAN bus in industrial environments or even pipes between processes in the same computer.

A part from the great flexibility in designing distributed control systems, by using this approach, it is also possible to easily interface the application to a simulator in order to validate the control algorithms instead of trying them into the real platform.

2.4. Real control

Another design principle that must be taken into account is that most control applications must execute in real time [4]. This implies that none of the modules introduced so far can be blocked waiting for data or an event to take place. Since blocking is in general unavoidable due to the asynchronous nature of the events to wait for, and continuously polling is not efficient in terms of CPU usage, the use of multiple threads of execution becomes necessary.

Although extremely powerful, multi-threaded applications introduce a lot of new problems such as thread synchronization, shared memory control and the thread handling itself. The use of such advanced programming techniques considerably increases the application complexity. In order to simplify the use of threads, it is necessary to define a new set of modules to handle the threads, the mutual exclusion methods and the logical events.

In order to include the functionalities to create and handle multiple threads, as well as transparently handling the shared memory and the logical event notifications used for synchronization, each of the generic modules introduced in section 2.2 will have either to inherit or use the new defined thread modules. One of the main problems when using these features is that they are strongly dependent on the operating system used, issue that shall be addressed in the next section.

2.5. Multi-platform

Nowadays, both Windows and Linux operating systems (O.S.) are world wide used in most scientific and industrial environments. A control application that can run in any of these O.S. should be much more useful. There exist some programming languages specially designed to generate multi-platform applications but which are not generally suited for real time applications, since they are interpreted in run time and not compiled before execution (i.e. Java).

Even though, considering the software framework presented so far, only few of the modules actually depend on the O.S. used. These are the VCD and those used to handle threads, shared memory and logical events. If the end user interface is based on a windowed graphical scheme, it will be also necessary to use a library which is independent from the underlying O.S.

Since the software framework core modules are platform independent, any application built on top of them will also be platform independent, and even more, the designer may use advanced programming techniques such as threads and logical events without having to worry about specific O.S. implementation issues.

3. PROPOSED IMPLEMENTATION

Yet quite powerful, the described approach is difficult to implement due to the high level of generality implicit in the development of the software framework core. In this section, a possible implementation is presented using an object oriented programming language (C++). Thus, each module is implemented as a class which inherits or use the necessary features from the other classes in the way presented in Fig. 1.

Class *CModule* is the most important one since each of the generic modules inherit most of its functionality from it. To comply with the principle of section 2.3, it includes the necessary code to configure the module in order to be executed locally or remotely and to work as a client or server.

For the case it is configured as a remote client, this class generates the needed commands, sends them through the VCD (implemented in the class *CComm* and waits for the answer. On the other hand, when configured as a remote server, this class waits for commands, executes any required action and returns the necessary data. *CModule* also inherits all the necessary features to handle multiple threads from the class *CThread* to comply with the principle of real-time control (section 2.4). Class *CThread* allows any derived class to handle multiple threads, to assign a different execution function to each of them and to end threads independently using a unique finalization event (implemented as a *CEvent* object). The *CThread* class also inherits from *CMutex* class to handle any shared memory issue.

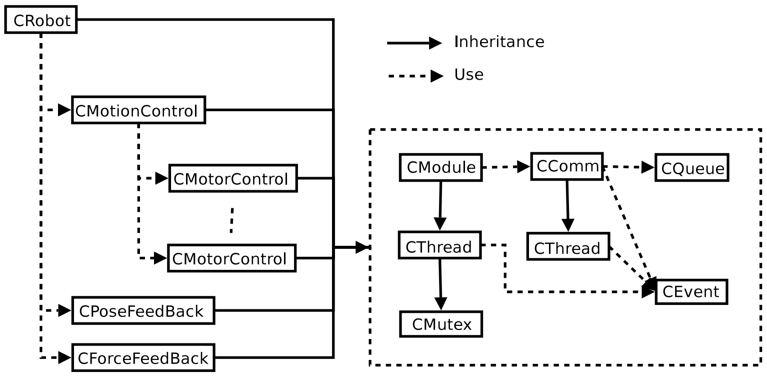

Figure 1: Class hierarchy for the software framework core.

The VCD handles the incoming and outgoing data through circular queues implemented in *CQueue*. It also inherits from *CThread* to wait for new data in an independent thread to avoid blocking the main execution thread. When new data is received, executes a user defined function and/or stores the new data into the receive queue. This VCD may be used as the base class for any communication object, so it is only necessary to implement the way to open, configure, close, read from and write to the specific communication device (Ethernet, RS-232, etc.).

As presented in section 2.2, the generic modules present are the robot module, the force and pose feedback modules and the motion and motor control modules, which correspond to the *CRobot*, *CForceFeedBack*, *CPoseFeedBack*, *CMotionControl* and *CMotorControl* classes respectively. Most of the *interface* functions on these classes are virtual or abstract allowing the derived classes to redefine or complete the basic implementation (polymorphism).

In order to be able to execute the application in both Windows and Linux as introduced in section 2.5, all the O.S. dependent classes actually provide the implementation in both O.S., and only the corresponding part is compiled to generate the final application. These classes provide a unique *interface* to the other modules whatever the O.S. is.

4. ASSESMENT OF RESULTS

The robot platform used to test the software framework presented in this paper consists of a 3 degrees of freedom (dof) modified SCARA architecture robot[1]. As shown in Fig. 2, the designed manipulator has four joints, with the particularity that two of them have been attached together (joints labelled O and O` in Fig. 2 so the movement of point T in the figure is straight along axis x. The other two degrees of freedom are used to control the pan (joint T) and tilt (joint Q) of the end effector. The physical implementation is shown in Fig. 2.

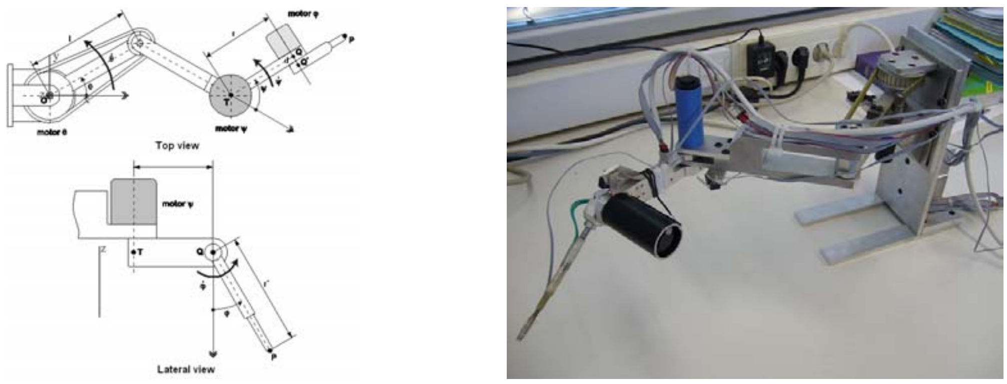

Figure 2: On the left, the top and lateral view of the robot architecture, on the right the physical implementation of the robot architecture.

Three actuators are needed to move the robot articulations. Chosen devices have been DC motors with a planetary gear box including an incremental magnetic encoder in the motor shaft in order to feedback the axis position. Additionally, a 2 dof force sensor based on strain gauges has been fixed to the end effector of the manipulator. It allows measuring the force components that are transversal to the robot end effector. The whole system is completed with an external optical position feedback sensor based on a 2D Position Sensing Device (PSD). All the external devices (motor controllers and feedback devices) communicate with the computer using a unique serial link. A specific designed serial hub is used to handle the data packets in and out of each device.

The software architecture used for this particular application is shown in Fig. 3. The classes *CAgullaMotorControl* and *CAgullaMotionControl* inherit directly from *CMotorControl* and *CMotionControl* respectively and they only extend the configuration functions of the base class to send the corresponding commands to the real controllers and define the main control functions.

The pose and force feedback classes *CAgullaPoseFeedBack* and *CAgullaForceFeedBack* inherit from *CPoseFeedBack* and *CForceFeedBack* respectively. They implement the low level data acquisition functions. *CAgulla* inherits from *CRobot* and it just defines the kinematic and dynamic models of the robot described at the beginning of this section.

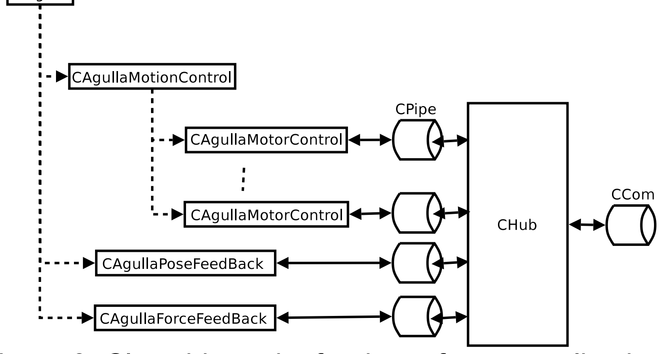

Figure 3: Class hierarchy for the software application.

Class *CHub* implements a VCD hub so any object derived from *CComm* may be used. It inherits from *CThread* to handle incoming data in any of the communication devices without blocking the main thread. In this case two communication devices are used, *CCom* to send and

[1] The robot, named Agulla, has completely been designed and built in the IRI under the research grant CeRTAP (Ref. I00544).

receive data from the low level hardware and *CPipe* to exchange data between the application modules. Both *CPipe* and *CCom,* just implement the low level physical access to the communication devices. Another communication class is used to split the application through several computers using a Local Area Network (*CSocket*) to test the distributed capability.

By using the base framework introduced in section 3 to design a software application for this particular platform, it has been possible to develop a control software using external position and force feedback in a short time. The only modules that were necessary to be implemented were the kinematic and dynamic models of the robot, adapting the motor control and feedback modules to the characteristics of the particular device drivers and the way those modules interface to the hardware through the serial hub (*CHub*).

Finally, it has been possible to split the control application through several computers using the distributed capability principle, in which the software framework is based, with minimum modification of the source code. For every remote module it was only necessary to assign a communication device (*CSocket*) and create a main program which generates the corresponding part of the control application for each computer.

5. CONCLUSIONS AND FUTURE WORK

Given a robotic arm we want to control it in order to solve a task. The control software should be general for any kind of robot arm and set of sensors. This paper introduces a set of design principles which seek to reduce robotic applications design and implementation time so reducing the errors present in any practical implementation as well. Then, a possible implementation of the general software framework proposed in section 2 is developed. Experiments show that the solution presented in this paper, although its limitations, allow the robotic applications designer to save development time while keeping the overall complexity low. Although the control algorithms always worked, due to the bandwidth requirements of some modules (i.e. motor control), the performance achieved for some configurations is severely limited. There exists open-source applications which handle similar problems [7,8] but they are not well fitted for small control applications like the one presented in section 4 due to the high complexity involved.

We have learnt that because of limitations in the programming language used to develop the application, some of the design principles presented in section 2 can not be completely fulfilled, and the final application itself is highly sensitive to implementation issues. Also, to completely verify the design principles it would be necessary to evaluate the effort required to design a control application for multiple and heterogeneous platforms.

REFERENCES

[1] F. Buschmann, Rational architectures for object-oriented software systems, Journal of Object-oriented programming, pp 30-41, 1993.
[2] P.Dario, R. Dillman, H.I. Christensen, Euron Roadmap document, 2005
[3] Hirzinger, G.; Sporer, N.; Schedl, M.; Butterfass, J.; Grebenstein, M, Robotics and mechatronics in aerospace, 7th International Workshop on Advanced Motion Control, 2002.
[4] Y. Kuo, A distributed real-time framework for robotics applications, Master's Thesis, Univeristy of Auckland, Submitted 2004.
[5] B. Meyer, Object-oriented software construction, Prentice-Hall, 1997.
[6]I.A.D. Nesnas, Toward developing reusable software components for robotic applications, Technical report JPL, 2001.
[7] ORCA, http://www.orca-robotics.sourceforge.net/, 2004.
[8] OROCOS, Open Robot Control software Open Robot Control Services, http://www.orocos.org.

ABOUT THE AUTHORS

Sergi Hernandez Juan, PhD student, Institut de Robòtica i Informàtica Industrial (IRI), Spain, Phone: +34 93 401 57 91, E-mail: shernand@iri.upc.es.
Josep M. Mirats Tur, post-doc researcher, Institut de Robòtica i Informàtica Industrial (IRI), Spain, Phone: +34 93 401 07 75, E-mail: jmirats@iri.upc.es.