A Method of Integrating Robot Programming Frameworks

Piotr Trojanek[‡] and Cezary Zieliński[‡]
[*‡] Institute of Control and Computation Engineering, Warsaw University of Technology, Warsaw, Poland
P.Trojanek@elka.pw.edu.pl, C.Zielinski@ia.pw.edu.pl

Abstract There is an ongoing quest for producing an ultimate robot programming framework that can be utilised in the design of controllers for any types of robots and moreover executing any tasks. Although many solutions have been presented since the early eighties of the XX century and much progress has been attained, the ultimate solution is still far away. The designed frameworks excel in certain areas, while exhibit significant drawbacks in some other. This paper focuses on an alternative solution to this problem – on fusion of programming frameworks. However, this shifts the problem to another area, namely the communication between the software originating with different frameworks. Combining the capabilities of `MRROC++` and `Player` is presented as an example.

1 Introduction

One of the first precursors of robot programming frameworks were `PasRo` (Blume and Jakob, 1986) and `RCCL` (Hayward and Paul, 1986). `RCCL` later evolved into `KALI` (Hayward et al., 1989). At the time that they were defined they were simply called robot programming libraries, as the benefit of associating a programming pattern was not evident. Over the years not only the base programming language changed, but also new programming paradigms have been employed and the necessity of a programming template became obvious. Initially `Pascal` (`PasRo`) and `C` (`RCCL`) were used and thus procedural programming paradigm was employed. Currently `C++` or even `Java` are used and object oriented or component based programming paradigms are utilised. Newer frameworks (e.g., `MRROC++` (Zieliński, 1999; MRROC++, 2007), `OROCOS` (Bruyninckx et al., 2003; OROCOS, 2007), `Player` (Vaughan et al., 2003; Gerkey et al., 2003; Player/Stage, 2007)) come with a pattern according to which the programmer assembles the building blocks into a program that controls the hardware and solves the task at hand.

There are many robot programming frameworks, thus it is impossible to mention all of them. However, at least some ought to be characterised. `OROCA` (Makarenko et al., 2006; Brooks et al., 2007) produces distributed component-based systems. Its emphasis is on: cross platform operation involving different operating systems, i.e. is a form of middleware. `OROCA` was initially `CORBA` based, but due to the complexity of the latter

[*]This work has been supported by MNiSW grant no. 3 T11A 009 29

Ice was chosen as a task communication management tool, thus many programming languages can be supported (e.g., C++, C#, Java, Python). However, the assumed method of communication limits the utilisation of OROCA to mobile robots, where the communication delays are not that significant.

OROCOS (OROCOS, 2007) applications are built of components, which form a network. For this purpose application templates are provided, e.g., a pattern for motion control contains components for: device access, position control, path planning and data reporting. The granularity of control components may vary from an algorithm computing inverse kinematics to controlling a whole robot. OROCOS uses five methods of interfacing components, that is through: properties, events, methods, commands and data flow ports. The emphasis of this framework is on configurability, which is achieved through XML.

Simplicity of code generation is at the focus of $\mathtt{G^{en}oM}$ (Generator of Modules) (Alami et al., 1998; Fleury and Herrb, 2001), which is a tool for automatic generation of entities providing services upon request. The system builder specifies the interfaces between modules and provides codels (code elements) that are responsible for providing the functionality of the module. Control requests influence the execution of a service (e.g. parameterize, abort, interrupt it), whereas execution requests start services. A running service, called an activity, exchanges data with other modules through posters. Services are provided by execution tasks. Besides one or more execution tasks, a module contains a single control task that is responsible for asynchronous communications with the clients of the module, checking the validity of the incoming requests and for initiating the service execution. Execution tasks and the control task contain codels.

One of the main problems with diverse robot programming frameworks is that each one of them was tested on different hardware, thus the specific software (device drivers, algorithms) exists only in some of them. The bigger the popularity of a certain framework the larger the chance that it will support the required hardware. However, quite frequently, not all of the hardware and not all data processing capabilities are provided by one framework. In such a situation either specific drivers have to be developed separately and appended to the chosen framework or mechanisms enabling the utilisation of both frameworks have to be developed. In this paper we advocate the latter solution. As an example we present the merging of MRROC++ and the Player frameworks.

2 Structure of the MRROC++ based controllers

MRROC++ provides a library of software modules (i.e. classes, objects, processes and threads) and design patterns according to which any multi-robot system controller can be constructed. This set of ready made modules can be extended by the user by coding extra modules in C++. The structure of MRROC++ is due to formal considerations presented in (Zieliński, 2001, 2006).

MRROC++ based controllers assume a hierarchical structure with a single coordinator and many effectors (manipulators, mobile platforms *etc.*) (fig. 1). Each of the effectors can use as many exteroceptors (sensors gathering the information from the environment) as necessary. The state of the effector is monitored by proprioceptors. The resulting system is thus composed of a single coordinator, many effectors with each one of them possessing many exteroceptors. Such a system can be treated as an embodied agent. This

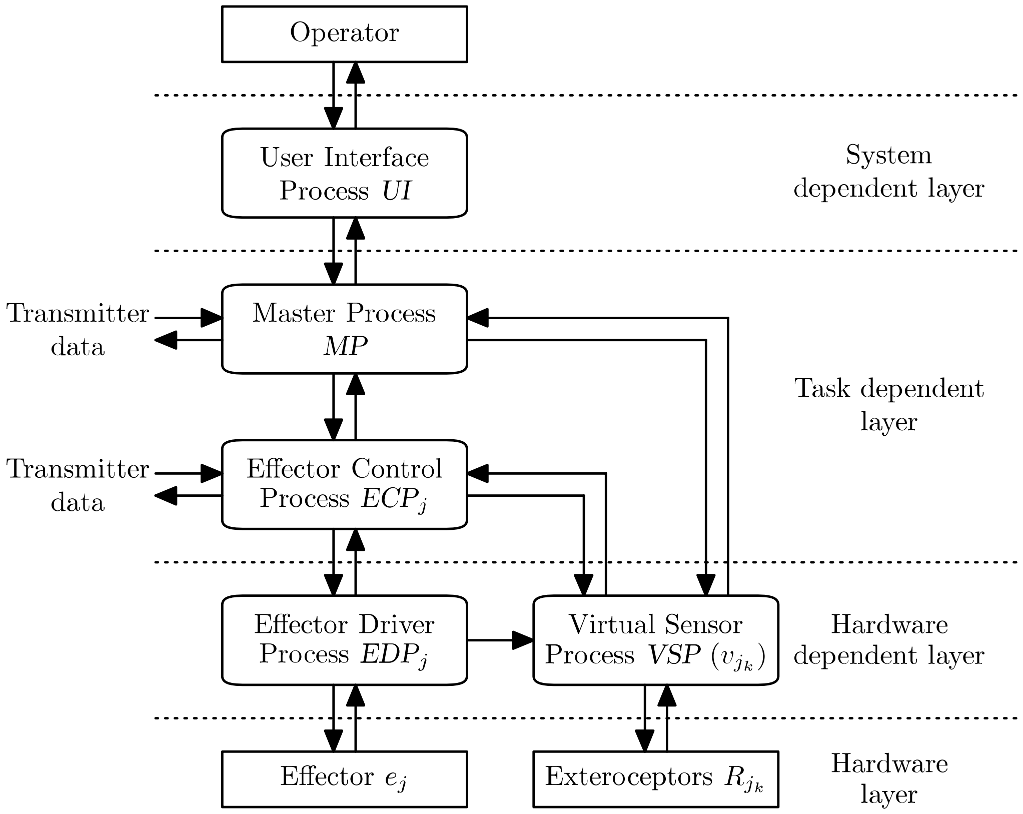

Figure 1. General structure of a `MRROC++` based system/agent ($j = 1, \ldots, n_e$, where n_e is the number of effectors, and $k = 0, \ldots, n_r$, where n_{j_r} is the number of virtual sensors associated with effector e_j)

agent has extra communication capabilities with other such agents through transmitters. If the coordinator of a single-agent is dormant, independent operation of effectors results. Effectors can also be divided into groups, with each group having its own coordinator – several agents thus result. One-member groups are also possible.

A `MRROC++` based control system is a set of processes (fig. 1):

UI – User Interface Process – a single system configuration dependent process (it can be absent, if the system is meant to be fully autonomous),

MP – Master Process – a single process representing the coordinator,

ECP – Effector Control Process – responsible for the execution of the task allotted to the effector – there are as many such processes as there are distinct effectors,

EDP – Effector Driver Process – responsible for controlling the hardware associated with the effector (proprioceptors are used by the effector control algorithm) – there are as many *EDP*s as there are *ECP*s,

VSP – Virtual Sensor Process – responsible for performing data aggregation on exteroceptor readings and thus producing a virtual sensor reading – zero or more of those processes can be associated with *MP* or any of the *ECP*s.

Each of the above processes may consist of several threads enabling concurrent realization of their tasks, e.g. acquiring data, interpreting and processing it and dispatching it to other components of the system.

From software engineering perspective process contains two elements: its shell and its kernel. The shell is responsible for the communication with the other processes and possibly other agents, and for error handling. The kernel executes of the user's task. In the case of *MP* and the *ECP*s the kernel is usually composed of the `Move` instructions producing motion of the effector or causing the effector to exert forces and torques on the environment. The shell contains the communication buffers responsible for contacting the *VSP*s, the transmitters that communicate with other agents, and buffers for contacting other processes that are either higher up or lower down in the control hierarchy of the agent. Error monitoring is done by standard exception handling mechanisms, so it is active during the execution of the `Move` instructions, as well as any other instructions of `C++` language. The generated exceptions are caught and handled at the topmost level of each process.

The `Move` instructions of the *MP* pertain to all of the effectors while the `Move` instructions of the *ECP*s deal with single effectors. In both cases those instructions cause the computation of a position/force command that is being transmitted to the lower control layers, i.e., *MP* sends this data to the *ECP*s while *ECP*s to the *EDP*s. All of this communication takes place through buffers that hold data for one control period – the macrostep. Typically a macrostep is defined as several steps, where each step is executed by the *EDP* at a servo sampling rate, e.g. 1 ms. The main argument of the `Move` instruction is a motion generator. A motion generator computes the next effector position taking into account the current state of the effector, the associated *VSP* readings, the data obtained via transmitters from the other agents and the data obtained from the higher layers of the control system (if they exist). It generates the set value for the next macrostep. It is the responsibility of the user to deliver appropriate motion generators for the execution of their task. Some of them are readily available in the framework. The generated exceptions, due to the detection of errors, are caught and handled at the topmost level of each process.

3 Structure of the `Player` based controllers

The general structure of the `Player` based controllers is composed of two main components connected together by a TCP/IP network. The first one is the `Player` server and the second one is the client application (fig. 2). The `Player` server is a robot device interface, which provides unified access methods to common hardware found in mobile robot systems. `Player` software is a collection of: device drivers, the server (the goal of which is to provide network transport mechanisms) and the client libraries (which communicate with the drivers through well defined interfaces). The drivers are coded in `C++`, but the server can provide access to their methods by any network mechanism. The robot high level control can be coded in a variety of programming languages, as long as the client library for a specific language is available.

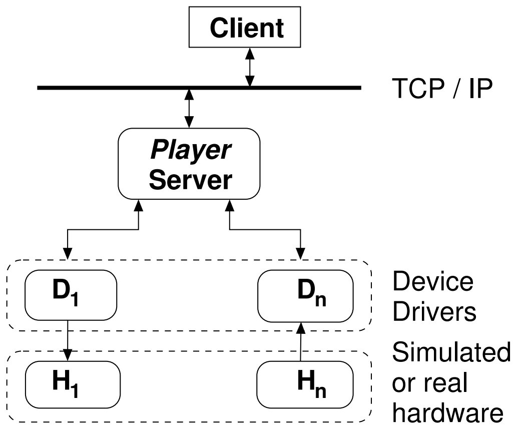

Figure 2. Control structure of `Player` software

The `Player` based controllers do not assume any structure and leave much more flexibility for the programmer. In most common scenario there is an instance of `Player` server running onboard a mobile platform and one client application, which communicates with the robot over wireless. The client continuously polls for new data and after processing it sends a new command to be executed by the drivers. Typically this loop runs at a frequency of 10 to 100 Hz, which is enough to control relatively not demanding mobile platforms and is in the range of most common sensor update rates (e.g.: camera, rangefinder). The `Player` server with its device drivers runs on Linux operating system, while with portable client libraries it is possible to control the robot from virtually all common operating systems through TCP/IP network.

4 Integration of `MRROC++` and `Player`

From the point of view of `MRROC++` controller designer the `Player` server can be seen as a remote repository of effector and sensor devices, similar to the native *VSP* and *EDP* processes. Usually a hierarchy of drivers is utilized while programming within the `Player` framework. One of the reasons for this is sensor data fusion, e.g., when mobile robot position is estimated using both odometry, which often provides unreliable data due to wheels slipping, and laser range finder readings, which are utilized for corrections. To keep the hierarchical use of drivers it is more convenient to let them run inside the original `Player` server shell code and to communicate with the most external interface through TCP/IP client library instead of wrapping each driver in the `MRROC++` process shell. Using this approach, it is preferred to integrate `Player` server as an external agent and communicate with it on the `MRROC++` side using the transmitter concept.

The transmitter is an abstract class representing the inter–agent data exchange both in the *MP* and the *ECP*. They are used for storing the incoming and outgoing data from/to the other agents. Both the MP and the ECP may need to communicate with many agents. Thus each of those processes will have many transmitters holding the

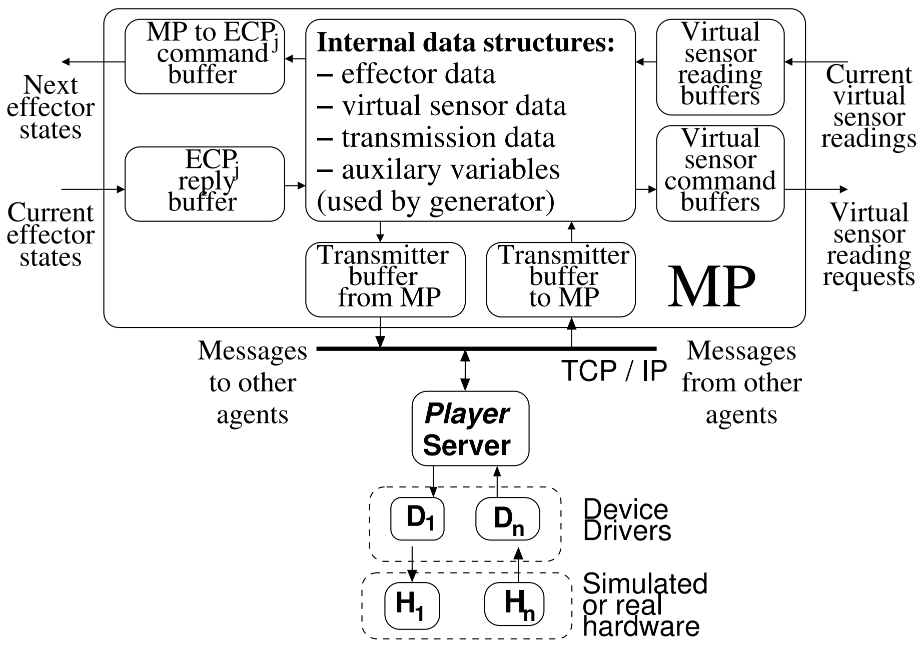
MP to ECP$_j$ command buffer
Internal data structures:
– effector data
– virtual sensor data
– transmission data
– auxilary variables (used by generator)
Virtual sensor reading buffers
Current virtual sensor readings
Next effector states
ECP$_j$ reply buffer
Virtual sensor command buffers
Current effector states
Virtual sensor reading requests
Transmitter buffer from MP
Transmitter buffer to MP
MP
Messages to other agents
TCP / IP
Messages from other agents
Player Server
D$_1$
D$_n$
Device Drivers
H$_1$
H$_n$
Simulated or real hardware

Figure 3. Integration of `Player` server as an external agent of a `MRROC++` controller (communication between the coordinator and remote devices)

current state of the interchange of information between those processes and the agents. This class is a functional equivalent of stubs, which represent remote object proxy, e.g., in `CORBA`. The transmitter class has two methods, which copy data between the internal controller data structures and the communication buffers.

The communication with the `Player` server is handled by a separate thread inside the transmitter class. This allows nonblocking calls from the `MRROC++` code and thus enables adaptation to diverse device update rates of industrial manipulators (typically few milliseconds) and mobile robots (typically tens or hundreds of milliseconds). The dedicated thread handles communication buffers data transmission utilizing native `Player` protocol.

At the initialization stage the transmitter object connects to a remote server and then sends and receives data at the frequency governed by the `Player` protocol. Messages from and to the remote devices are processed by *MP* (in the case of cooperating robots) and *ECP* (in the case of a single remote controlled manipulator) control loops in a similar way that they communicate with the native *VSP* and *EDP* processes, i.e., by utilizing the buffers (fig. 3).

The inter-framework communication on the `MRROC++` side has been implemented using *messip* library (Message Passing over TCP/IP). In this way inter-framework message passing is made similar to the native QNX message passing over QNET used by the `MRROC++` processes running on QNX only. In both cases the initiator of communication sends a message and waits for a reply from the receiver. As communication is handled by separate threads the other activities of the proces can be handled without disruption. The interface to the inter-framework communication tools consists of seven functions handling: creation and destruction of the communication channel, opening and closing of the channel, waiting for the message, dispatching the message and dispatching the reply to the message. If a programming environment permits the use of external system libraries, this interface can be used by programs written in different languages – not only `C++`, which is the base language for `MRROC++`. Thus, e.g. `Java` or `Lisp` can be used to create components of the created system. Low latency of communication has been achieved by minimization of the number of system calls necessary to control the TCP/IP communication and the elimination of dynamic data structure allocation, which induces indeterministic times of execution.

5 Applications

The proposed approach has been implemented and verified by two specific applications. The first one shows how to overcome the problem of unsupported hardware in realtime operating systems such as QNX (e.g., USB devices and firewire cameras), which is used by `MRROC++`. As an example the control of a manipulator with a common USB gamepad device has been implemented. The manipulator controller is `MRROC++` based, thus relies on QNX, which does not support the USB devices, however `Player`, which is Linux based does. The problem of unsupported devices often leads to handcrafted remote device servers and custom TCP/IP based communication, which can be successfully avoided using the above presented method.

The second application controls cooperating mobile robot and immobile manipulator. Each of this two kinds of robots is controlled by a native driver processes (*ECP* and `Player` server respectively). At the level of the coordinator (i.e., *MP*) those processes are treated as a dependant effector and remote agent respectively. This application presents successful integration of robots with significantly different control rates (1 ms for the manipulator and 100 ms for the mobile robot).

6 Conclusions

The presented method of integrating software belonging to two different robot programming frameworks using TCP/IP based connection is effective for controlling distinct agents and utilizing foreign exteroceptors or effectors. The transmission delays introduced both by the transmission hardware and software (both by `MRROC++` and `Player`) is about 100 μs each. The total delay due to both of those factors is about 200 μs on AMD 1 GHz processors and 100 Mb/s Ethernet LAN – and that is acceptable.

The experience gained while using the `MRROC++` distributed programming framework showed that the communication model based on passing message packets to input and output buffers is the simplest, yet most efficient. It is intuitive and well suited to formal description of robot controllers, since it is natural to enclose message passing as an element of data processing loop. In terms of reduction of development time the major benefit stems from using the software that has been already developed and tested within each of the programming frameworks that are being connected, while the postulated connection is easy to implement if TCP/IP can be utilized on both sides – and this is usually the case.

Bibliography

R. Alami, R. Chatila, S. Fleury, M. Ghallab M., and Ingrand F. An architecture for autonomy. *Int. J. of Robotics Research*, 17(4):315–337, 1998.

C. Blume and W. Jakob. *Programming Languages for Industrial Robots.* Springer-Verlag, Berlin, 1986.

A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck. Orca: A component model and repository. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, pages 231–251. Springer, 2007.

H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time control core of the orocos project. In *Proceedings of the IEEE International Conference on Robotics and Automation, Taipei, Taiwan*, pages 2766–2771. September, 14–19 2003.

S. Fleury and M. Herrb. Genom user's guide. Report, LAAS, Toulouse, December 2001.

B.P. Gerkey, R.T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics, ICAR'03, Coimbra, Portugal*, pages 317–323. June 30 – July 3 2003.

V. Hayward and R. P. Paul. Robot manipulator control under unix RCCL: A robot control C library. *Int. J. Robotics Research*, 5(4):94–111, 1986.

V. Hayward, L. Daneshmend, and S. Hayati. An overview of KALI: A system to program and control cooperative manipulators. In K. Waldron, editor, *Advanced Robotics*, pages 547–558. Springer-Verlag, Berlin, 1989.

A. Makarenko, A. Brooks, and T. Kaupp. Orca: Components for robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06)*, December 2006.

MRROC++, 2007. URL `http://robotics.ia.pw.edu.pl`.

OROCOS, 2007. URL `http://www.orocos.org/rtt/documentation`.

Player/Stage, 2007. URL `http://playerstage.sf.net/`.

R.T. Vaughan, B.P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS'03, Las Vegas, Nevada*, pages 2121–2427. October 2003.

C. Zieliński. By How Much Should a General Purpose Programming Language be Extended to Become a Multi-Robot System Programming Language? *Advanced Robotics*, 15(1):71–96, 2001.

C. Zieliński. Transition-function based approach to structuring robot control software. In K. Kozłowski, editor, *Robot Motion and Control: Recent Developments, Lecture Notes in Control and Information Sciences, Vol.335*, pages 265–286. Springer Verlag, 2006.

C. Zieliński. The MRROC++ System. In *First Workshop on Robot Motion and Control, RoMoCo'99*, pages 147–152, June 28–29 1999.