First Workshop on Robot Motion and Control, RoMoCo'99, 28–29 June, 1999, Kiekrz, Poland. pp.147–152.

The MRROC++ System

Cezary Zieliński

Institute of Control and Computation Engineering, Warsaw University of Technology,
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland, e-mail: C.Zielinski@ia.pw.edu.pl

Abstract

The paper proposes a structure for open, hierarchical, multi-device controllers. The proposed structure takes into account that the system may contain several robots of different type, a certain number of cooperating devices, diverse sensors and also the fact that the task, the system has to execute, and the number and type of its components may vary considerably over time. The concept has been verified by designing a controller for a prototype RNT robot and an ASEA IRb-6 type robot. The flexibility of the system is due to the software, so the programming aspect is treated comprehensively in the paper.

1 Introduction

Robot controllers and the programming languages they interpret are inseparably bound together. Robots have to execute ever more complex and diverse tasks. The components of the system, i.e. number and type of robots, number and type of cooperating devices, number and kind of external sensors, that are necessary to carry out the job are not known before the task is specified and the solution to the problem is found. Controllers and programming methods of such systems have to take into account this fact. There are two solutions to this problem. Either the controller has to be universal, and so its specialised programming language must have wide capabilities, or the controller can be very specialised (i.e. suited to a very limited class of tasks and a single hardware configuration), but then it must be very easy to design, so that a specific controller for any task at hand can be designed quickly. This proposal follows the latter approach. It consists of: a general structure of the controller, a moderately sized set of construction modules that facilitate the construction of specialised controllers, and a method for both designing these controllers and for constructing and adding new modules to the original set.

Initially the idea of universal robot controllers and their specialised programming languages prevailed in the robotics community (e.g. `WAVE` [10], `AL` [8], `AML` [11], `RAPT` [1], `SRL` [4], `TORBOL` [12]). It soon turned out that such a controller has to be able to interpret a very complex language, having the same abilities that general purpose computer programming languages have and, moreover, extra capabilities for dealing with robots and sensors. Even when a general purpose programming language was adopted as a basis, a robot programming language had to have extensions (i.e. instructions and data types) due to specific devices composing the system. It was extremely difficult to decide what kind of additional components should this general purpose language contain so that any foreseen system could be controlled and programmed. So this approach has been given up for cases where it was expected that the system hardware may vary considerably with changing tasks. Paradoxically it turned out that the universal controllers are much better suited to dealing only with initially well defined classes of tasks. In such a case the language can be tailored to the system configuration and the class of tasks at hand. Obviously, the broader the class the more general the language.

Soon another idea emerged. Instead of defining a language that would have all the components of a general purpose language and, moreover, a few additions proprietary to the particular needs of robot system control, it became evident that it is much more convenient to use a general purpose language and to code the robot specifics as a library of software modules. This idea was followed in: `PASRO` [3, 4], `RCCL` [6], `KALI` [7], `ARCL` [5], `RORC` [14, 13], `MRROC` [15]. Usually procedural programming paradigm has been followed, but currently this changes to object–oriented approach [17]. This proposal uses the above mentioned approach and `C++` as an implementation platform.

2 Structure of a MRROC++ controller

Theoretical reason for selecting the presented structure of the Multi-Robot Research-Oriented Controller: `MRROC++` is presented in [14, 18, 19]. It is implied by the following facts.

- Each robotic system consists of three subsystems:
  - **effectors** e, i.e. devices exerting influence over the environment, e.g. robot arms, conveyors,
  - **receptors** r, i.e. devices gathering the information about the system and the environment state – real (hardware) sensors,

This work has been supported by Warsaw University of Technology statutory grant 504/036/8 and Program in Control, Information Technology and Automation PATIA. The author gratefully acknowledges the participation of Dr. W. Szynkiewicz in the implementation of `MRROC++`

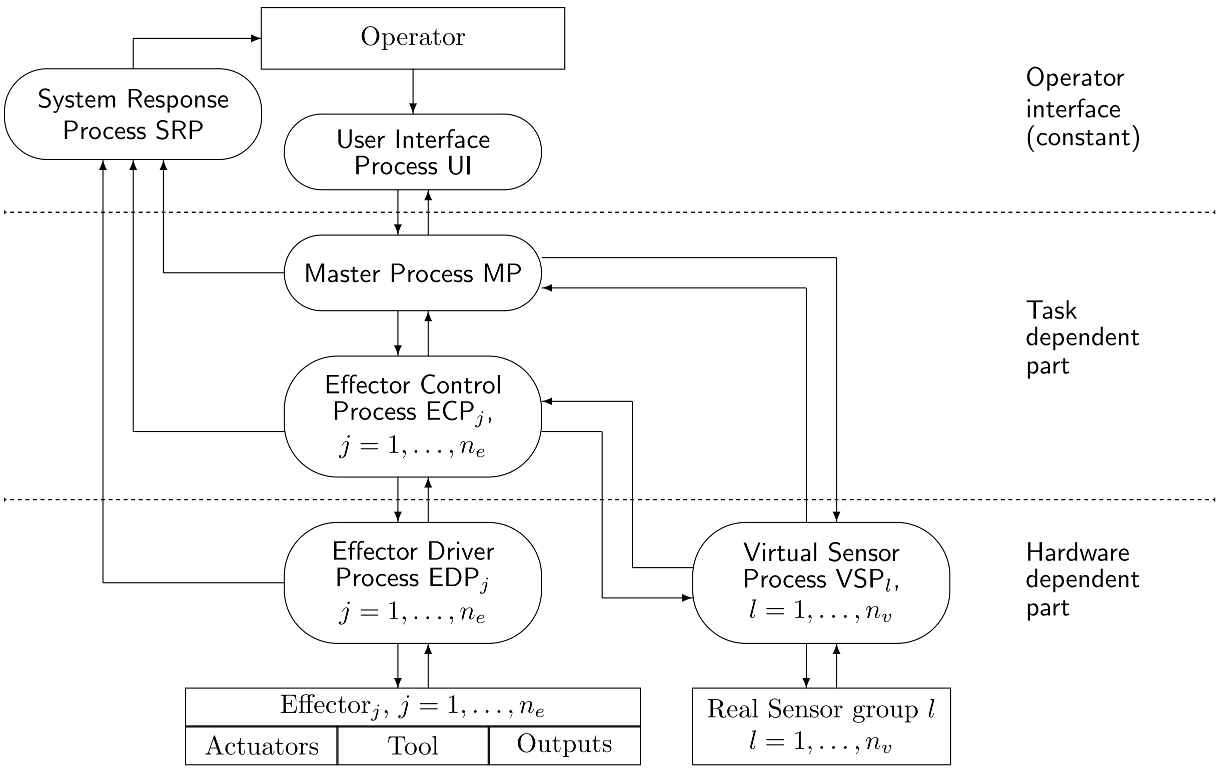

Operator
System Response Process SRP
User Interface Process UI
Operator interface (constant)
Master Process MP
Task dependent part
Effector Control Process ECP_j, $j = 1, \ldots, n_e$
Effector Driver Process EDP_j $j = 1, \ldots, n_e$
Virtual Sensor Process VSP_l, $l = 1, \ldots, n_v$
Hardware dependent part
Effector_j, $j = 1, \ldots, n_e$
Actuators
Tool
Outputs
Real Sensor group l $l = 1, \ldots, n_v$

Figure 1: Structure of a MRROC++ controller (n_e – number of effectors, n_v – number of virtual sensors)

   – **control subsystem** c – executing the task by utilising sensor data and moving the effectors accordingly.

- Rarely the information from hardware sensors can be used in motion control directly. Either the data from several simple sensors (e.g. several strain gauges) needs to be used collectively or some specific information has to be extracted from the data obtained by a complex sensor (e.g. a camera). Because of that the raw data obtained from real sensors must be aggregated into a **virtual sensor** reading v.
- The system must have the capability of reconfiguring real sensor groups into diverse virtual sensors.
- Each of the effectors must have a dedicated driver due to the possibility of connecting or disconnecting those devices to/from the system.
- The effectors should be capable of independent operation and any form of cooperation.

The structure of the system is divided into three parts (fig. 1). The first part is hardware dependent, the second is task dependent and the third composes the operator interface and is constant. In this way the modifications due to hardware or task changes are minimised.

MRROC++ is a library of software modules (i.e. classes, objects, processes and procedures) that can be used to construct any multi-robot system controller. This set of ready made modules can be extended by the user by coding extra modules in C++. The freedom of coding is, however, restricted by the general structure of the system. New modules have to conform to this general structure. Even if a single-robot controller is designed it is assumed that it can work in a multi-robot environment, so its controller really has the capability of controlling several robots. The same applies to sensors. Regardless of the fact, whether they are necessary for the execution of the user's task, the potential for their utilisation always exists in the system.

The MRROC++ system has a hierarchical structure (fig. **??**). It runs on PC computers (Pentium or 486 processor based are preferred) connected by an Ethernet network. This network is supervised by a real-time operating system QNX-4 [20]. A single process coordinating the operation of the whole system is called Master Process MP. Each effector (either a robot or a cooperating device) has two processes controlling it: Effector Control Process ECP and Effector Driver Process EDP. The former is responsible for the execution of the user's task dedicated to this effector, and the latter for direct control of this effector. EDP is supervised by ECP. In this way the user's task and the effector specific control have been separated and are independent of each other. The process of extracting meaningful information for the purpose of motion control is named data aggregation and is performed by a virtual sensor. Data aggregation is done by Virtual Sensor Processes VSPs. Moreover the system contains two processes dedicated to the interaction with the operator. User Interface Process UI handles operator commands. System Response Process SRP displays all the system status and error messages on the screen of the monitor. Both processes perform in a windows environment, so operator commands such as: initiation of execution of the user's program, its termination or pausing and resuming are done by clicking on certain icons.

The user's program (task) is coded by writing some distinct portions of MP and ECP. There are three kinds of tasks that multi-robot systems deal with, namely:

- robots performing independently,
- loosely cooperating robots (e.g. one robot handing an object to the other one),
- tightly cooperating robots (e.g. common transfer of a rigid object over a specified trajectory).

The first kind requires of the MP only the initiation and termination of the task. The second requires additionally the synchronisation of the ECPs, from time to time. In the last case the MP must generate the trajectory for all the robots. In this case the ECPs only transfer the MP commands to adequate EDPs.

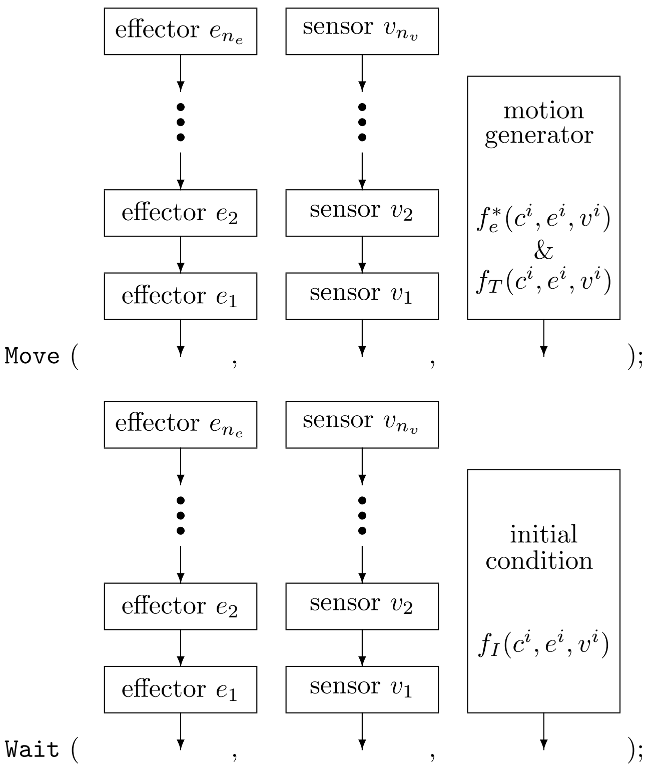
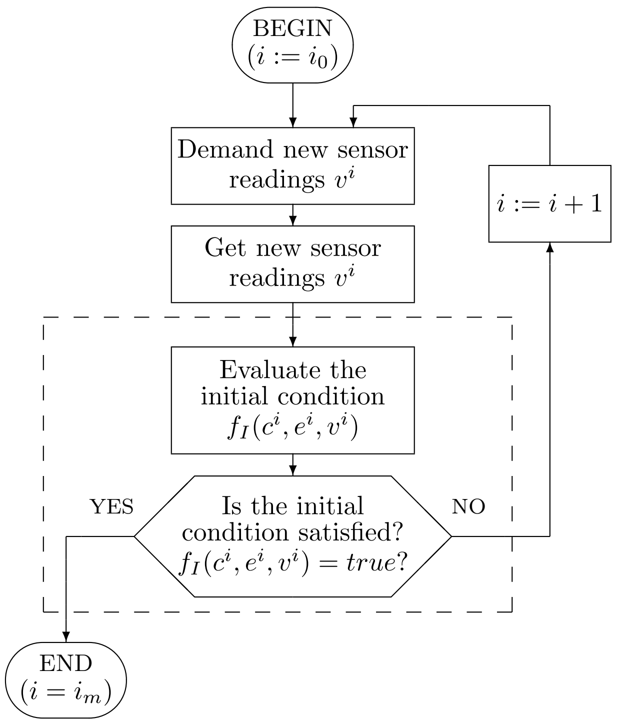


Figure 2: MRROC++ motion instructions

3 Motion instructions

Each of the **motion instructions** (i.e. instructions exerting direct or indirect influence over the effectors) is executed in steps i, starting with i_0 and ending in i_m, i_m may be unknown *a priori.*

Virtual sensors are used either to monitor the current state of the system and its environment or to control those two entities, i.e. to influence their future states [14]. Monitoring is used to detect one of the following conditions. **Initial condition** $f_I(c^i, e^i, v^i)$ monitoring results in the designation of an instant from which the motion commences. Monitoring of **terminal condition** $f_T(c^i, e^i, v^i)$ is done to stop the motion when the virtual sensors detect adequate circumstances. Throughout the life of the system, **error condition** $f_E(c^i, e^i, v^i)$ must be monitored. MRROC++



BEGIN ($i := i_0$)
Demand new sensor readings v^i
Get new sensor readings v^i
$i := i + 1$
Evaluate the initial condition $f_I(c^i, e^i, v^i)$
YES
Is the initial condition satisfied? $f_I(c^i, e^i, v^i) = true$?
NO
END ($i = i_m$)

Figure 3: Wait instruction flow chart

utilises sensors for both: monitoring of all three conditions and influencing the future states through the **effector transfer function** $f_e^*(c^i, e^i, v^i)$ designating the next effector state. Effector motion utilising sensors is programmed by using motion instructions. One complex motion instruction, which encompasses all possible situations, can be defined [18, 19]. Nevertheless, it is much more convenient to the users to introduce two separate, but simpler instructions (fig. 2). The `Wait` instruction, with its semantics reflected in the flow chart presented in fig. 3, monitors the initial condition. The `Move` instruction, with its semantics reflected in the flow chart presented in fig. 4, controls the motion and monitors the terminal condition. Error condition monitoring is done by exception handling, so it is performed in parallel to the execution of those instructions, as well as any other instructions of the general purpose language, i.e. C++ in this case. Each of those instructions needs two lists of objects: robots (or speaking more generally effectors) and virtual sensors. Besides that the `Move` instruction takes as its third argument an object named motion generator (i.e. `generator`). The methods (i.e. portions of C++ code) of this object execute the operations circumscribed by the dashed line in the flow chart shown in fig. 4. They are responsible both for evaluating the terminal condition $f_T(c^i, e^i, v^i)$ and computation of the future (or demanded) state of the effectors $f_e^*(c^i, e^i, v^i)$. The `Wait` instruction needs an object responsible for the evaluation of the initial condition $f_I(c^i, e^i, v^i)$ (i.e. `condition`). The methods of this object execute the operations circumscribed by the dashed line in the flow chart shown in fig. 3. The `generator` and `condition` descendant objects contain methods delivered by the user and dependent on the task the system has to execute.

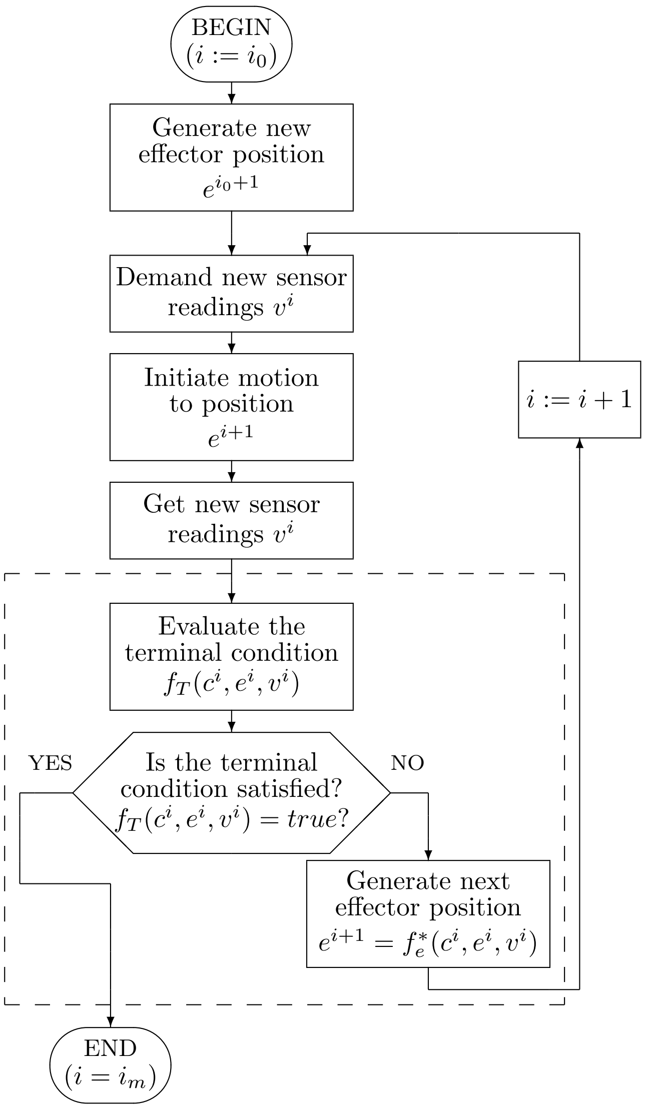

BEGIN
$(i := i_0)$
Generate new effector position e^{i_0+1}
Demand new sensor readings v^i
Initiate motion to position e^{i+1}
Get new sensor readings v^i
Evaluate the terminal condition $f_T(c^i, e^i, v^i)$
Is the terminal condition satisfied? $f_T(c^i, e^i, v^i) = true?$
YES
NO
Generate next effector position $e^{i+1} = f_e^*(c^i, e^i, v^i)$
$i := i + 1$
END
$(i = i_m)$

Figure 4: `Move` instruction flow chart

4 Internal structure of MP and ECP processes

Both the MP and the ECPs consist of two parts (fig. 5). The shell is responsible for initialisation and inter-process communication. The kernel is created by the user out of the `Move` and `Wait` instructions and any other `C++` statements deemed necessary. On the MP level these instructions (fig. 2) take as their arguments lists of robots and virtual sensors. On the ECP level, for each ECP only a single robot exists, so this robot and a list of sensors are the arguments of these instructions. Each level operates on its own image of a robot (ECP) or several robots (MP) and images of virtual sensors. Precisely speaking, those images are the arguments of the `Move` and `Wait` instructions. The shell contains all the necessary software means for updating the state of the images. The image of an effector on the ECP level maintains direct contact with its respective EDP, so its state reflects the state of the effector itself, and issuing of the commands to the image causes the effector to execute them. On the MP level the images are updated by using the information transmitted through the ECP level. Whenever an ECP of the MP need sensor data they use the images of sensors reflecting the state of virtual sensors. The respective sensor images maintain direct contact with the VSPs. The above mentioned contact is carried out by using adequate communication buffers and *rendezvous* operations of the `QNX` operating system. The internal structure of the MP and the ECPs is presented in figures 6 and 7 respectively. The structures of the buffers and the images are device dependent and so sometimes have to be defined when new hardware is added to the system. The `MRROC++` system contains predefined buffers and images that usually suffice. The motion generator is responsible, on the MP level, for the generation of trajectories of the end-effectors of all the robots in the list forming the argument of the `Move` instruction. On the ECP level the generator creates a trajectory for a single robot. Each higher level issues commands for the image of the lower level. On the EDP level hardware command are generated. A generator uses the information obtained from sensors (i.e. sensor image list), the current state of the robots (i.e. robot image list) and the commands of the upper layer, if it exists, to produce the next pose of the robots being in the list of robots forming the argument of the `Move` instruction. Moreover, the motion generator can use its internal data to produce the next pose, e.g. a list of previously taught-in poses or can compute those poses from certain parameters (functional description).

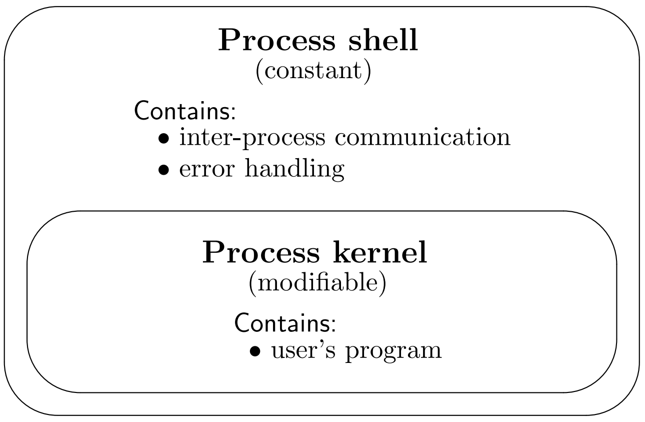


Figure 5: Structure of ECP and MP proceses.

The condition, being the argument of the `Wait` instruction, if true, terminates the waiting, and if not causes the system to pause. For each `Move` and `Wait` instruction the user writes in `C++` his or her own generator and condition objects. In this way, usually only very small portions of MP and ECPs have to be rewritten when the task changes. The modifications are cumulated in the separate code of specific generators and conditions. Errors are dealt with within the whole system by exception handling, so the user needs not deliver the program code responsible for that.

5 Effector Driver Process EDP

The code of EDP is not modified by the normal user. It varies only when the effector hardware changes. The

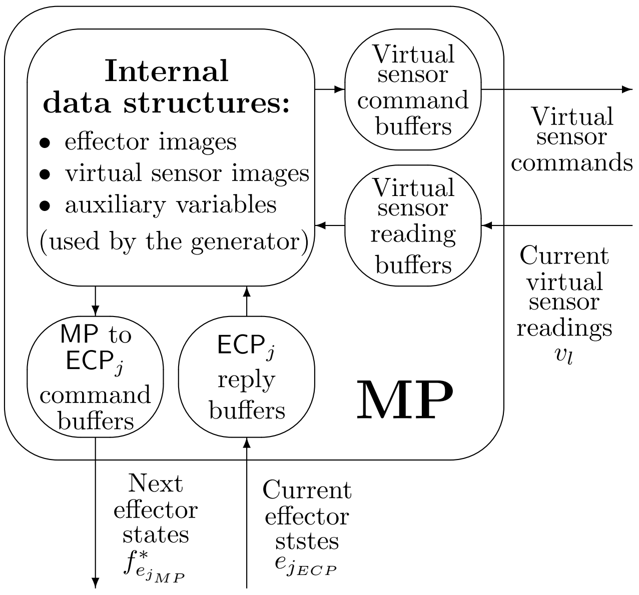

Internal data structures:
• effector images
• virtual sensor images
• auxiliary variables
(used by the generator)
Virtual sensor command buffers
Virtual sensor commands
Virtual sensor reading buffers
Current virtual sensor readings v_l
MP to ECP$_j$ command buffers
ECP$_j$ reply buffers
MP
Next effector states $f^*_{e_{j_{MP}}}$
Current effector ststes $e_{j_{ECP}}$

Figure 6: Internal structure of MP process.

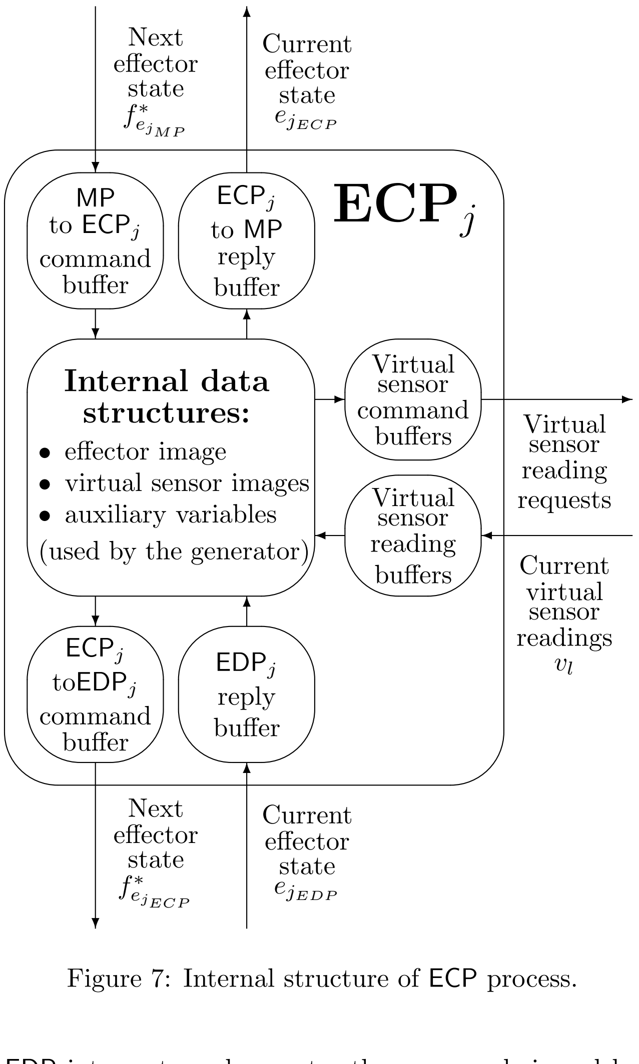

Next effector state $f^*_{e_{j_{MP}}}$
Current effector state $e_{j_{ECP}}$
MP to ECP$_j$ command buffer
ECP$_j$ to MP reply buffer
ECP$_j$
Internal data structures:
• effector image
• virtual sensor images
• auxiliary variables
(used by the generator)
Virtual sensor command buffers
Virtual sensor reading requests
Virtual sensor reading buffers
Current virtual sensor readings v_l
ECP$_j$ toEDP$_j$ command buffer
EDP$_j$ reply buffer
Next effector state $f^*_{e_{j_{ECP}}}$
Current effector state $e_{j_{EDP}}$

Figure 7: Internal structure of ECP process.

EDP interprets and executes the commands issued by its respective ECP. The list of all commands is presented in fig. 8. There are two main commands: `SET` and `GET`. The former influences the state of the EDP and so the robot, and the latter causes the EDP to read its current status. Sometimes, the user needs to exert simultaneous influence on the robot and to read its current state, so a `SET_GET` command has been defined, which causes simultaneous execution of a `SET` and `GET` command. To retain client-server mode of EDP operation the ECP issues a `QUERY` command to obtain any feedback from the EDP, so `QUERY` has to follow any other command. As most robots have incremental position measuring devices, it is required that prior to task execution the robot defines its current position in the workspace. This is usually done only once and by moving the arm to a known base location. This is caused by the `SYNCHRO` command.

The `SET` command can: set the arm position, i.e. cause the robot to move to the desired position, redefine the tool affixed to the arm, change the set of parameters or the local corrector of the kinematic model, switch the servo-control algorithm of any or all of the arm motors, alter the parameters of the servo algorithm, or set the binary outputs of the robot controller. The `GET` command can read: the current position of the arm, the currently used tool, and the kinematic model and corrector and servo algorithm parameters, or the binary inputs to the robot controller. Switching of kinematic model parameters and correctors should improve local precision of arm motions. Modification of servo algorithms or their parameters can improve tracking ability. This switch can be performed when significant load modification is anticipated. Both the tool and the arm positions can be defined in terms of homogeneous transforms, Cartesian coordinates with orientation specified either as Euler angles or in angle and axis convention. In the case of the arm position, moreover, it can be specified in terms of joint angles or motor shaft angular increments. The arm position argument in the command can be regarded as an absolute or relative value. Each motion command `SET ARM` is treated as a macro-step. An extra argument specifies into how many interpolation steps it should be divided. Because the incremental position measurement is delivered simultaneously with commanding the new PWM value for the motors, to obtain a continuous motion without stopping, the reading has to be delivered to the upper control layers a few steps before the interpolated motion terminates. The user has control over that by specifying in which step number the reading is required. If this value is one more than the number of interpolation steps, the reading is delivered after the motion stops. For uninterrupted trajectory segment transition it suffices if it is one less than the number of interpolation steps.

6 Conclusions

A considerable effort has been concentrated on developing new RPLs, both specially defined for robots and computer programming languages enhanced by libraries of robot specific procedures. Specialised languages result in a closed structure of the controller. If new hardware is to be added to the system, usually some changes to the language itself have to be done. Those changes have to be reflected in the language compiler or interpreter. Because of this, rather robot

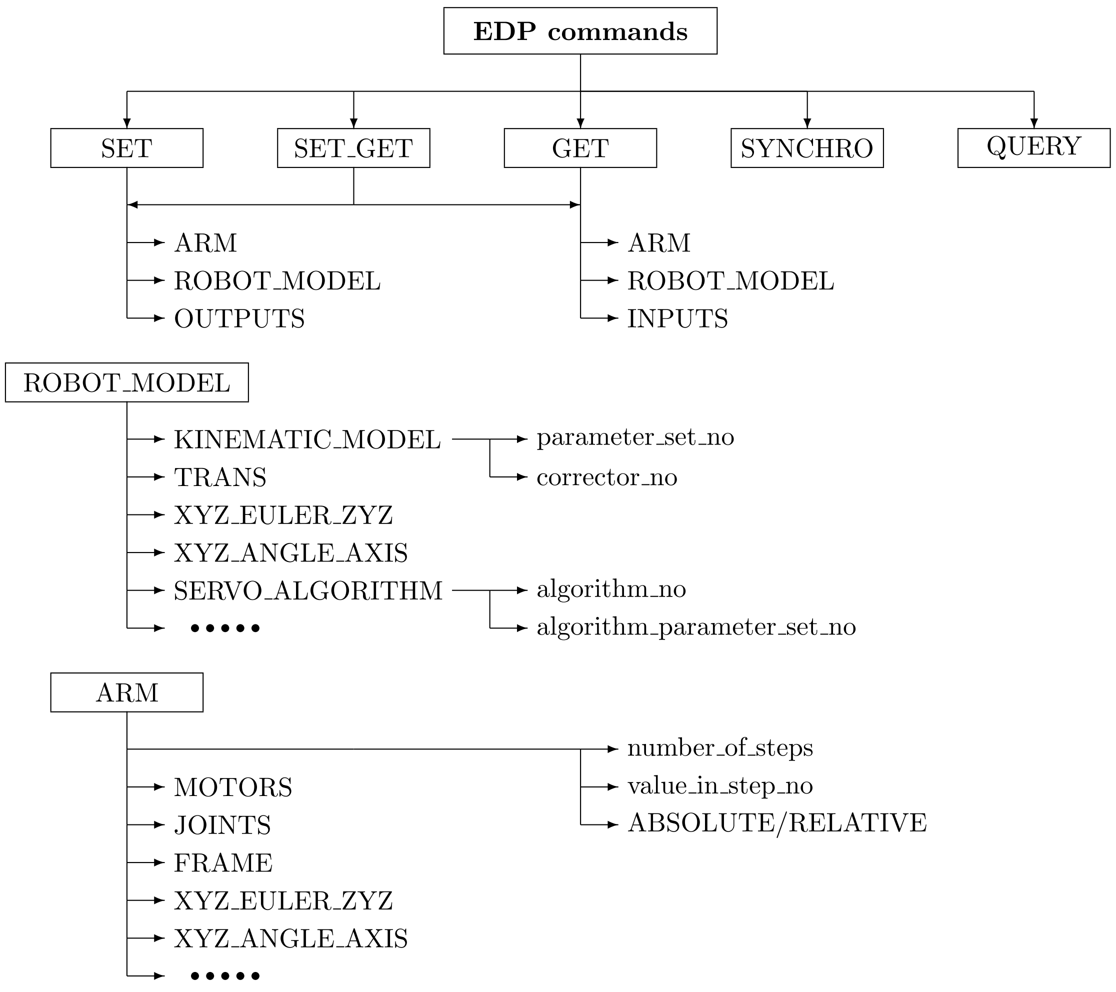


Figure 8: Effector Driver Process commands

programming languages/libraries submerged in general purpose programming languages are used by the research community than specialised RPLs. Multi-Robot Research-Oriented Controller is submerged in `C++` running under real-time operating system QNX [20] capable of supervising a computer network. Initially `MRROC` [15] was implemented using procedural approach, but currently this has been changed to object-oriented approach, and hence `MRROC++` resulted. The switch of programming approach not only simplified robot task coding, but also proved to be much more effective in the implementation. Polymorphism enables late binding, so `Move` and `Wait` procedures could be coded without the specific knowledge of what types of robots and sensors will be used. Exception handling enabled the separation of the code processing normal system functioning from the code dealing with error situations. Finally, the formal approach [18, 19] pointed out what should be the structure of the software and limited the user interference with the system to a few object classes that the programmer has to derive from: `robot`, `sensor`, `generator` and `condition` classes. Whenever a new task is to be undertaken by the system, a new controller is assembled out of the above objects and adequate calls to `Move` and `Wait` procedures and other C++ instructions. The programming of such a system consists in assembling out of library objects and procedures a controller dedicated to the execution of the task at hand.

`MRROC++` can currently control a modified ASEA type IRb-6 robots and the prototype serial-parallel structure robot [2, 9]. Force/torque, ultrasonic, and infrared sensors, CCD cameras and a conveyor belt have been included in the system. The described approach to programming has been validated on different tasks. Cooperative transfer of a rigid body by two robots [16] and engraving inscriptions in wood by the prototype robot equipped with a milling tool were among them.

References

[1] Ambler A. P., Corner D. F.: *RAPT1 User's Manual*. Dept of Artificial Intelligence, University of Edinburgh, 1984.

[2] Bidziński J., Mianowski K., Nazarczuk K., Słomkowski T.: *A manipulator with an arm of serial parallel structure*. Archives of Mechanical Engineering, Vol.39, No.1-2, 1992, pp.65-78.

[3] Blume C., Jakob W.: *PASRO: Pascal for Robots*. Springer-Verlag, Berlin 1985.

[4] Blume C., Jakob W.: *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[5] Corke P., Kirkham R.: *The ARCL Robot Programming System*. Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, 14-16 July 1993. pp.484-493.

[6] Hayward V., Paul R. P.: *Robot Manipulator Control Under Unix RCCL: A Robot Control C Library*. Int. J. Robotics Research, Vol.5, No.4, Winter 1986. pp.94-111.

[7] Hayward V., Daneshmend L., Hayati S.: *An Overview of KALI: A System to Program and Control Cooperative Manipulators*. In: *Advanced Robotics*. Ed. Waldron K., Springer-Verlag, 1989.

[8] Mujtaba S., Goldman R.: *AL Users' Manual*. Stanford Artificial Intelligence Lab., 1979.

[9] Nazarczuk K., Mianowski K., Olędzki A., Rzymkowski C: *Experimental investigation of the robot's arm with serial-parallel structure*, Proc. IX World Cong. Theory of Machines and Mechanisms, Milan 1995, pp. 2112-2116,

[10] Paul R.: *WAVE: A Model Based Language for Manipulator Control*. The Industrial Robot, March 1977, pp.10–17.

[11] Taylor R. H., Summers P. D., Meyer J. M.: *AML: A Manufacturing Language*. Int. Journal of Robotics Research, Vol.1, No.3, 1982. pp.842–856.

[12] Zieliński C.: *TORBOL: An Object Level Robot Programming Language*. Mechatronics, Vol.1, No.4, 1991. pp.469-485.

[13] Zieliński C.: *Flexible Controller for Robots Equipped with Sensors*. 9th Symp. Theory and Practice of Robots & Manipulators, Ro.Man.Sy'92, 1-4 Sept. 1992, Udine, Italy, Lect. Notes: Control & Information Sciences 187, Springer-Verlag, 1993. pp.205-214.

[14] Zieliński C.: *Robot Programming Methods*. Publishing House of Warsaw University of Technology, 1995.

[15] Zieliński C.: *Control of a Multi-Robot System*, 2nd Int. Symp. Methods & Models in Automation & Robotics MMAR'95, 30 Aug.–2 Sept. 1995, Międzyzdroje, Poland. pp.603-608.

[16] Zieliński C., Szynkiewicz W.: *Control of Two 5 d.o.f. Robots Manipulating a Rigid Object*, IEEE Int. Symp. on Industrial Electronics ISIE'96, 17–20 June 1996, Warsaw, Poland. Vol.2, pp.979–984.

[17] Zieliński C.: *Object-Oriented Robot Programming*, Robotica, Vol.15, 1997. pp.41–48.

[18] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems Utilising Sensory Information*. 3rd ECPD Int. Conf. Advanced Robotics, Intelligent Automation and Active Systems, 15–17 September 1997, Bremen, Germany, pp.176–181.

[19] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems*. 4th Int. Symp. Methods and Models in Automation and Robotics MMAR'97, 26–29 August 1997, Międzyzdroje, Poland, pp.1121–1126.

[20] *QNX System Architecture*. Quantum Software, 1992.