Robotica (1997) volume 15, pp 41–48. © 1997 Cambridge University Press

Object-oriented robot programming

Cezary Zieliński

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

SUMMARY

The paper presents an object-oriented approach to the implementation of a software library (MRROC+) which contains building blocks for the construction of multi-robot system controllers tailored to meet specific demands of a task at hand. Moreover, the paper supplies a brief overview of robot programming methods.

KEYWORDS: Robot programming; Object-oriented approach; Software library; Multi-robot system.

1. INTRODUCTION

Methods of robot programming can be assigned to two broad classes: on-line and off-line programming methods. **On-line programming** utilizes the robot while the program is being created. **Off-line programming** does not use the robot to write the program.

1.1. On-line programming

On-line programming is based on teaching a robot the trajectories it has to follow. **Teaching** is done by leading the robot arm through a sequence of motions and recording these motions, so that later they can be replayed automatically. The arm during teaching can be propelled either manually (i.e. the operator uses his muscles to shift the arm from one position to the other) or by its drives (i.e. the operator uses the joystick, the keyboard of the teach pendant, or a scaled down replica of the manipulator to command the drives to appropriate positions). Regardless of the way of propelling the arm during teaching, there are two ways of recording the trajectory of the arm motion. In the PTP (Point To Point) method the arm is transferred to each characteristic point of the trajectory, stopped there, and by pressing a special button on the control panel, this position is memorized by the control system. During playback the robot arm goes through these points using some form of interpolation between them. In the CP (Continuous Path) method, as the arm is transferred, the positions are being recorded automatically at constant intervals of time. As the points are very near each other no special interpolation routines are necessary. Moreover, the motions can be played back with different speeds by changing the time base (the interval of time allowed for reaching the next point).

The main advantage of teaching is its simplicity, so that even an operator with virtually no qualifications can do it. The main drawbacks of this method, in its pure form, are that: it is very difficult to incorporate the data gathered by sensors, no documentation of the program is created, it is easier to create a new program than to modify an old one, and last but not least, during teaching the robot is occupied by the programming and not by the production task.

1.2. Off-line programming

Off-line programming is based on textual means of expressing the task that the robot system has to accomplish. The task is expressed in a *robot programming language* (*RPL*). This can be either a specially defined language for robots or a universal *computer programming language* (*CPL*). The advantage of using RPLs is associated with making the robot more productive (e.g. it is not used for programming), the ease of utilization of sensor data, and creation of program documentation.

To make a robot more productive, the phase in which it is required for programming has to be as short as possible. In other words, robot programming has to be made independent of the robot. The program is developed off-line and later only loaded to the control system for execution. The problem with this approach is that although currently manufactured robots feature high repeatability, they exhibit low accuracy. This necessitates the calibration of the program created off-line. As the solution to this is an open research problem[1], the industrial robots used currently cannot be programmed strictly off-line. Nevertheless, RPLs draw quite a lot of attention. The solution to the calibration problem is one cause of this and the other is the simplification of coding programs in these languages. In addition, only RPLs fully solve the problem of sensor integration.

Every programming language operates on specific abstract concepts. An instruction of a language is composed of one or more keywords and zero or more arguments. These arguments express abstract concepts. Computer languages operate on variables of different types. The values of these variables describe the state of certain abstract notions. The instructions, and therefore the languages, are classified according to the abstract notions they refer to.

The main instructions of RPLs are the ones causing the motion of the effectors, i.e. **motion instructions**. The abstract notions that these instructions refer to are: the manipulator joints, the end-effector or the objects of the work space. These notions are used to express the state of the effectors.

Each of the enumerated notions creates a certain virtual environment, in which the instructions of the

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

language operate. The **virtual environment** is a model of a robot system as perceived by the programmer through the programming language he uses. In other words, those elements which had been considered important were selected from the real environment to constitute the virtual environment. Only some elements of the real environment (including the robot) are the basis for creating abstract notions that compose the virtual environment. The virtual environment is a simplified model of the real environment. On the degree of this simplification and on the abstract notions that were chosen to make up the virtual environment depends the complexity of the control system of a robot. The programmer through each level of the control structure perceives the real environment as a simplified model—he perceives the virtual environment. In the case of RPLs the above-mentioned abstract concepts are hierarchically related and so these languages can be classified into levels.[2] It should be noted that the virtual environment is a certain abstraction of the robot system.

The languages of the lowest level are called *joint level languages.* The instructions of those languages cause the generation of sequences of signals controlling the drives of the manipulator. Hence, in this case the manipulator joints form the virtual environment. The design of a control system accepting these instructions is quite routine, but to forecast how the tool will behave when all the drives are in motion is not as simple. For simplification of the design we have to pay the price of the programming complexity. Because of the difficulties associated with using such languages they are no longer implemented.

The languages of the next level free the users from this disadvantage (e.g. *AML*[3,4], *RCCL*[5,6], *KALI*[7,8], *RORC*[9–11]). The main concept of the virtual environment of this level is the manipulator's end-effector, so these languages are called the *manipulator level languages.* Although it is easy to predict the trajectory of the robot tool when using languages of this level, the programmer still has to be concerned with the description of all the motions of the manipulator instead of simply stating what actions have to be performed to accomplish the task.

The instructions of *object level languages* (e.g. *AL*,[12,13] *RAPT*,[13,14] *TORBOL*,[9,13,15], *SRL*,[4] *PASRO*[4,16]) operate in virtual environments composed of models of objects existing in the work space. The programmer states only which objects should be transferred, so that the task will be accomplished. The robot control system, using its knowledge of the objects and the relations between them, will relocate the manipulator in such a way so as to complete the job. From this level onward the programmer does not have to busy himself with the motions of the robot arm, but can concentrate on the operations that have to be executed.

On the fourth level, the **task level**, instead of specifying all operations, only a general description of the goal should suffice (e.g. *AUTOPASS*[17]). In this case the control system has to generate the plan of actions, and later carry it out. The prime difference between the third and fourth level languages is that to express tasks in the former we supply the plan of actions and in the latter the plan is generated automatically.

1.3. Hybrid programming

The drawbacks of pure off-line and on-line programming methods caused an intensive search for solutions of the afore-mentioned problems. One of the solutions consists in enhancing teaching by a textual method of programming—**hybrid programming**. In this case the control system is equipped with an interpreter of a language and only the arguments of motion instructions are supplied by teaching. In many cases the program can be created off-line and only these arguments are supplied on the factory floor. This solution partially eliminates the problems of handling sensors and program calibration, but still the robot is not productive during tecahing. In this case program documentation is obtained too.

VAL II[18] can be treated as a hybrid robot programming language, because besides the possibility of supplying numeric arguments to the motion instructions (e.g. **MOVE, MOVES**), the location to which the arm is commanded can be taught-in. The arm is transferred to the goal location by the teach-pendant. This location will be stored as an argument of the motion instruction when it is typed in. Usually, not many of such locations have to be taught-in. The remaining locations that the robot arm must attain either do not have to be stated very precisely or can be specified in relation to the exact ones that have been taught-in.

Until the calibration problem has been solved satisfactorily, some form of ''calibration by teaching'' will have to exist, so the hybrid programming will be used rather than the pure off-line method. Nevertheless, the off-line component of programming will be dominating, especially due to the tendency of incorporating sensors into modern robotic systems.

1.4. Open structure programming systems

Till now over a hundred robot programming languages have been implemented. Partial surveys of *RPLs* can be found in references 4 and 9. Unfortunately, most of RPLs have only single-site implementations, and their manuals are unavailable.

Quite a considerable effort has been concentrated on developing new robot programming languages, both specially defined for robots,[4,13,15] and computer programming languages enhanced by libraries of robot specific procedures.[4,5,11,19] Specialised languages exhibit a closed structure. If new hardware is to be added to the system, usually some changes to the language itself have to be done. Especially, if new sensors are to be incorporated this problem arises, both because the hardware specific software has to be supplied and because the method of sensor reading utilisation in motion control has to be coded. Those changes have to be reflected in the language and this brings about the necessity of modifying the language compiler or interpreter. Obviously we can try to make the language very universal by taking into account as many types of robots and sensors as possible,

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

but then most of the capabilities of the language will remain unused when programming a stable configuration system and will render the language difficult to master and implement. Because of this, robot programming languages/libraries submerged in universal computer programming languages are currently favored by robotics research community. Such programming systems have an open structure. Whenever the robot system has to be enhanced new hardware specific procedures are appended to the library/language and the universal language compiler remains unaltered.

This paper shows how such an open structure system can be designed and programmed on an example of a controller made for a new robot prototype. Once it is made certain that an open structure control/programming system has to be designed, one has to decide what programming methodology will be used and hence what implementation language platform to apply. This choice has to be made very carefully in the case of open structure systems, because not only is the language platform the means of implementing the library, but also it will be used by the future programmers as a means of expressing the tasks that the system will have to execute. Wrong choice made here can result either in difficulties with the implementation of the library or difficulties in programming future tasks or both. The most frequently utilised language platform has been either *Pascal, C* or *C++* very recently. Large libraries of robot specific procedures for the creation of both single- and multi-robot controllers have been designed, e.g. in *C*: *RCCL,*[5] *ARCL,*[19] *RCI,*[6] *KALI,*[7,8,20] *RORC*;[9–11] *MRROC*;[9,21] in *Pascal*: *PASRO,*[4] *ROPAS*[22]: in an object-oriented version of *Pascal*: *ROOPL*;[23] and in *C++*: *ZERO++*.[24]

While designing a controller for a new experimental manipulator with an arm of serial-parallel structure[25] exhibiting a very rigid structure, the following assumptions were made:

- the controller should have an open structure facilitating any robot control investigations, e.g. external sensor incorporation and utilisation, trajectory planning and generation, research of servo-control algorithms,
- the controller should treat the new manipulator as one of many it will control,
- it should have a user-friendly operator interface,
- it should be easy to create a controller tailored exactly to a research task at hand by building it of ready library blocks (objects) or new procedures and processes that can be easily coded.

To fulfil the above requirements it was decided that object oriented programming methodology (not to be mistaken with object level RPLs) will be most suitable for the task, so a concurrent version of *C++* running on top of a multi-computer real-time operating system **QNX-4** was chosen as an implementation platform. A formal approach to designing such controllers presented in reference 9, 11 and 21 has been followed. The paper shows the formal design approach followed, the obtained generic structure of the controller and how this structure is implemented by *C++* classes, objects, methods and processes, i.e. object-oriented programming paradigm.

2. STRUCTURE OF MULTI-ROBOT SYSTEMS

An open multi-robot system containing cooperating devices and equipped with diverse sensors is considered. No assumption is made as to what tasks will be performed by the system.

A multi-robot system is composed of three subsystems: **effectors** (manipulator arm or arms, tool and the devices cooperating with the robot), **receptors—real sensors,** and the **control subsystem** (i.e. memory: variables, program and program execution control). The state $\mathbf{s} \in \mathbf{S}$ of such a system is denoted in the following way:

$$\mathbf{s} = \langle \mathbf{e}, \mathbf{r}, \mathbf{c} \rangle, \ \mathbf{s} \in \mathbf{S}, \ \mathbf{e} \in \mathbf{E}, \ \mathbf{r} \in \mathbf{R}, \ \mathbf{c} \in \mathbf{C}, \tag{1}$$

where:

s —the state of the system,
e —the state of the effectors,
r —the state of the real sensors,
c —the control subsystem state,

S —the system state space,
E —the effector state space,
R —the real sensor reading space,
C —the control subsystem state space.

Since the data supplied by hardware sensors cannot directly be utilised in motion control it has to be processed to obtain an aggregate that can be used for trajectory modification or generation. This aggregate is named the virtual sensor reading.

$$\mathbf{f} = \mathbf{f}_v(\mathbf{r}, \mathbf{e}, \mathbf{c}) \tag{2}$$

where: **e** is the state of effectors, **r** is the state of receptors (hardware sensors) and **c** is the state of the control subsystem.

The control subsystem is responsible for computing motion trajectories for the effectors using its internally stored data, current state of the effectors and the virtual sensor readings.

The system state **s** is decomposed by taking into account several distinct effectors and that rather aggregated sensor readings **v** than real sensor readings **r** are used by the control subsystem to compute a motion trajectory.

$$\mathbf{s} = \langle e_1, \ldots, e_{n_e}, v_1, \ldots, v_{n_v}, \mathbf{c} \rangle \tag{3}$$

where: n_e is the number of distinct effectors in the system ($\mathbf{e} = \langle e_1, \ldots, e_{n_e} \rangle$) and n_v is the number of virtual sensors ($\mathbf{v} = \langle v_1, \ldots, v_{n_v} \rangle$). For the purpose of this paper the subsystems and their state are denoted by the same symbols.

To calculate the next effector state some computations have to be done. Those computations are done by the control subsystem. Obviously they can be done by a single centralised control subsystem, but a much better and clearer structure is obtained, if the state of the

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

control subsystem **c** is partitioned into $n_e + 1$ parts. As a result the following is obtained:

$$\mathbf{c} = \langle c_0, c_1, \ldots, c_{n_e} \rangle \tag{4}$$

Each subsystem c_l, $l = 1, \ldots, n_e$, is responsible for controlling an effector associated with it, and the subsystem c_0 is responsible for the coordination of all effectors. Hence, with each of the effectors e_l, $l = 1, \ldots, n_e$ an Effector Control Process is associated. Its state is expressed by c_l, $l = 1, \ldots, n_e$. The coordinating process is called the Master Process (MP) and its state is expressed by c_0.

Each control subsystem part c_l in conjunction with the part c_0 is responsible for calculating the next state e_{c_l} (calculated state) of the effector e_l (current state) and causes e_l to become equal to e_{c_l}, i.e. executes a motion step. Treating the system as a discrete time system, the next state of each of the effectors can be computed by a transfer function f_{e_l}:

$$e_{c_l}^{i+1} = f_{e_l}(e_l^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_l^i), \quad l = 1, \ldots, n_e \tag{5}$$

The Master Process and each ECP are responsible for computing adequate transfer functions f_{e_l} and executing the related motions. The state of each part of the control subsystem, is the following:

$$\begin{aligned} c_0^{i+1} &= f_{c_0}(e_1^i, \ldots, e_{n_e}^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_1^i, \ldots, c_{n_e}^i) \\ c_l^{i+1} &= f_{c_l}(e_l^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_l^i), \end{aligned} \tag{6}$$

Relationships (6) and (5) (the lists of function arguments exactly) show the interactions between the parts of the system and hence point out what communication links have to be established between the subsystems.

Each virtual sensor v_p, $p = 1, \ldots, n_v$ is implemented as a process running concurrently to other Virtual Sensor Processes and the Effector Control Processes. In consequence of (2)

$$v_p^i = f_{v_p}(r^i, e_l^i, c_0^i, c_l^i) \tag{7}$$

is obtained, where e_l is the state of the *l-th* effector (the one associated with v_p). Here it is assumed that only a single effector influences directly a virtual sensor, because only this effector can directly change the state (or rather the configuration) of the real sensors that are mounted on it. It is envisaged here that real sensors fixed to different effectors will not form a single virtual sensor. Usually f_{v_p} depends only on r^i.

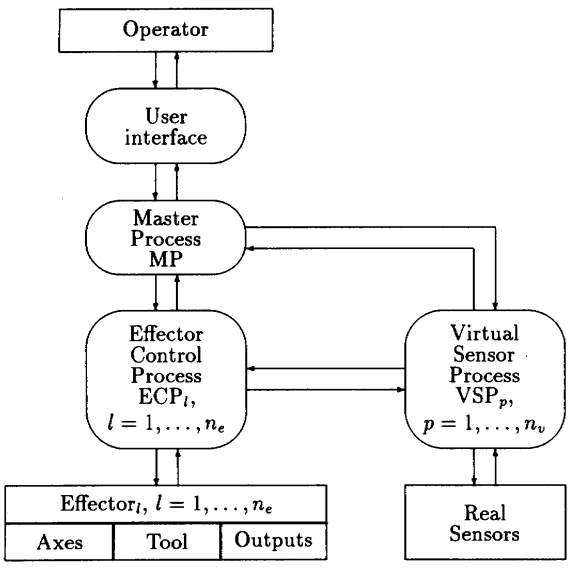

Fig. 1. Hierarchical structure of a multi-robot controller.

The above mentioned theoretical considerations suggest not only the components of the system but also the connections between them (Figure 1). This structure has to be implemented effectively by using the resources supplied by the real-time operating system and the implementation language. In the case of the multi-robot system in Warsaw University of Technology the structure has been implemented in the following way. Each Effector Control Process creates Virtual Sensor Processes according to the needs of control of motion. The Effector Control Processes in each step i obtain data from the Virtual Sensor Processes. Both kinds of processes can be treated as device dependent drivers. In this way, if only one component of the system is changed the remaining components remain unaltered.

The processes communicate through messages. The communication of each Effector Control Process with the Virtual Sensor Processes it uses can be of two kinds: interactive (Figure 2) and non-interactive (Figure 3). In the case of interactive communication the Effector Control Process sends a data request message to an adequate Virtual Sensor Process. The Virtual Sensor Process reads the real sensors, aggregates the obtained data and sends the result to the Effector Control Process. In the case of non-interactive communication the Virtual Sensor Process reads the real sensors, aggregates data and leaves the resulting reading in a buffer without any request from any Effector Control Process. An Effector Control Process can access sensor data immediately by reading the buffer where the aggregated data is stored.

A more elegant structure of the software component of the system can be obtained, if each Effector Control Process is partitioned into ECP proper and the Effector Driver Process EDP. The Effector Driver is responsible for:

- transformation of end-effector coordinates into joint coordinates and vice versa,
- transformation of joint coordinates into motor control increments and vice versa,
- transmission of the set-values to the servo-drives,
- transmission of servo status to upper levels of control structure,
- computation of the servo-control algorithm.

The ECP proper, in this case, is responsible for trajectory generation, when the robots loosely interact or are not related to each other. In the case of cooperative action of the robots, the ECP proper simply transmits the commands of the Master Process, which acts as a coordinator. This structure was utilised in the Multi-Robot Research-Oriented Controller MRROC+ (Figure 4). Large portions of MRROC+ have been

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

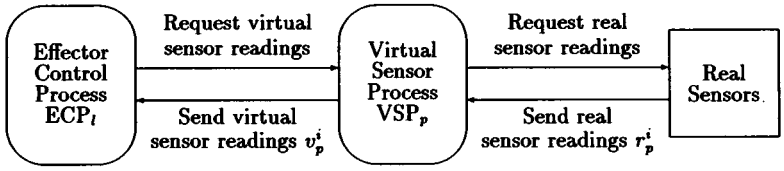

Fig. 2. Interactive method of reading sensor data.

implemented using object oriented programming methodology.

3. OBJECT ORIENTED PROGRAMMING—OOP

Object-oriented programming (OOP) methodology evolved from structured programming. **Structured programming** is a method of describing a programming task in a hierarchy of modules, each describing the task in increasing detail, until the final stage of coding is reached (programming by step-wise refinement). Strict adherence to modules renders GOTO instructions unnecessary, in effect exhibiting a clear program structure. Nevertheless, initially structured programming treated data and algorithms operating on this data as two separate entities—**procedural programming.** The **object-oriented programming** paradigm integrates data and procedures. An *object* is a collection of *data* (variables of appropriate type, which should be treated as *fields* of a *record*) and *procedures* and *functions,* which are called *methods,* operating on these variables. Several properties characterise an *object-oriented programming language*:[26]

- *encapsulation*—treating *data* and *code* operating on it as one entity—an *object.*
- *inheritance*—defining a hierarchy of *objects* in which each *descendant object* acquires all the properties of the *ancestor objects* (access to *data* and *code* of the *ancestors*) and receives some new properties specific to the newly created *object.*
- *polymorphism*—using the same name for an action that is carried out on different objects related by *inheritance.* The action is semantically similar, but it is implemented in a manner appropriate to each of the individual *objects of the hierarchy.*
- *exceptions*—separate code for ordinary situations and for error handling.
- *overloading*—change of operator semantics for new data types.

OOP methodology assumes that certain abstract *objects* will be defined by the programmer. These *objects* have their properties (*data*) and exhibit behaviours (*methods*). The program is written in terms of *objects* behaving in such a way so as to change their properties, i.e. applying *methods* to change *data.*

A *class* is a template of *objects* having the same structure (i.e. *data fields* and *methods*). *Objects* are instances of *classes.* Many objects can be instances of the same *class* and so will have the same properties, but usually a different state (e.g. values of data fields). The relationship between *objects* and *classes* is similar to the relationship between *variables* and *data types.*

At this point the misunderstanding which can arise from the traditional use of the term '*object*' in '*object level robot programming languages*' and in '*object-oriented programming languages*' (this time *CPLs*) has to be clarified. In the case of *RPLs* the notion of an '*object*' pertains to the real objects that are located in the robot's environment or to abstract models of these objects represented in a *RPL.* In the case of the *CPLs* the term '*object*' represents an abstract notion, which encapsulates *data* and *code,* and possesses the properties of *inheritance* and *polymorphism.*

This paper describes the application of OOP methodology to the creation of a manipulator level *RPL* (*object* library to be strict). For this purpose *C++* language was used.

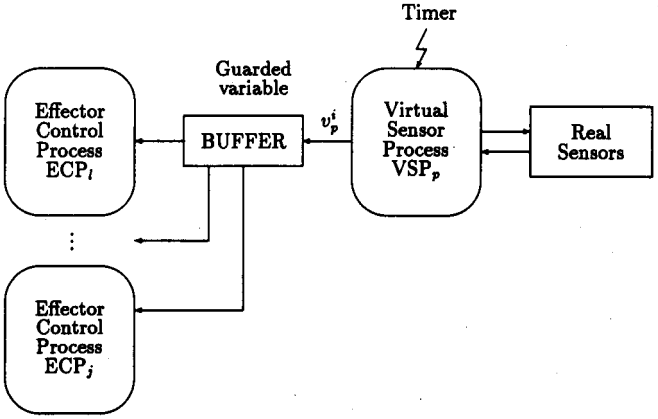

Fig. 3. Non-interactive method of reading sensor data.

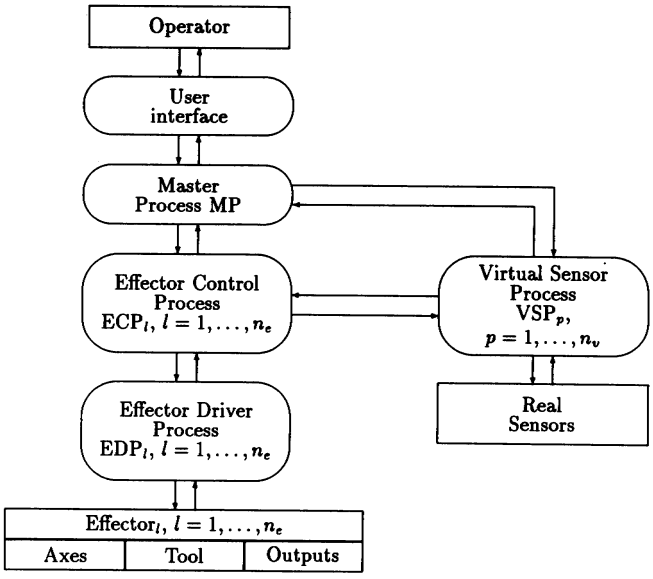

Fig. 4. Structure of *MRROC+.*

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

4. OOP IMPLEMENTATION OF *MRROC+*

The overall structure of the system is dictated by theoretical considerations of section 2 which resulted in the division of the system into independent processes running concurrently either on separate computers connected into a network or on a single computer in a time-sharing fashion or both. The choice is made by the programmer implementing a specific task. The OOP paradigm can be used only within the scope of each process separately. Nevertheless the communication buffers treated as *classes* can be shared by two processes communicating with each other. Each process has to do a job of its own and has to communicate with its neighbors obtaining data from them and in return delivering the results of its processing. This will be examplified by explaining in more detail the functioning of the Effector Driver Process EDP which itself is composed of several sub-processes (Figure 5).

The EDP receives commands from its corresponding ECP. The commands are (Figure 6):

- synchronise the robot,
- SET: tool definition, arm pose, outputs,
- GET: tool definition, arm pose, inputs,
- SET and GET simultaneously.

SETting an end-effector pose results in an arm motion and GETting the arm pose results in reading its current position and orientation. The tool definition can be specified in several ways: homogeneous transformation, X, Y, Z coordinates in relation to the wrist flange and several sets of orientation angles (e.g. Euler) or by using an angle and axis notation. The arm pose can be specified in the above manner too and moreover using joint variables and motor increments. The coordinates can be relative or absolute.

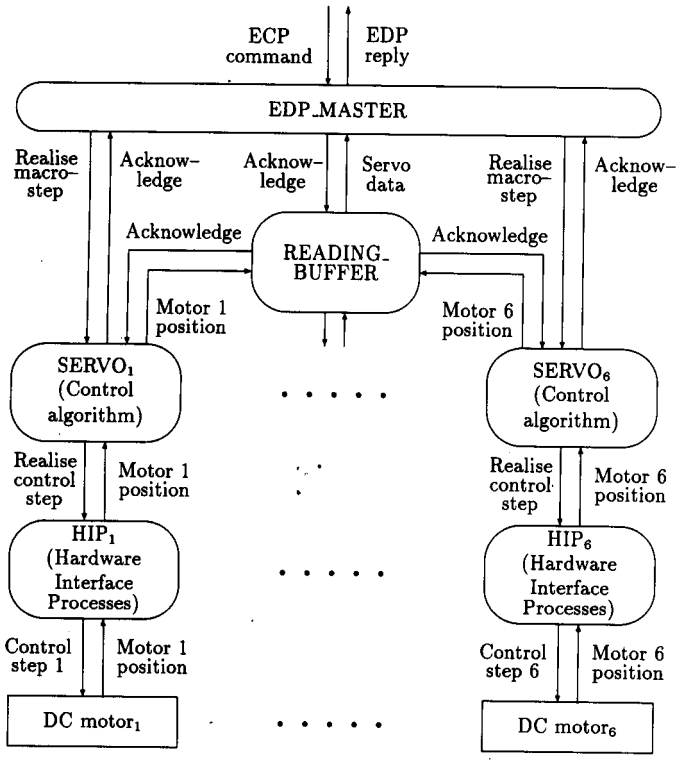

Fig. 5. Internal structure of the Effector Driver Process.

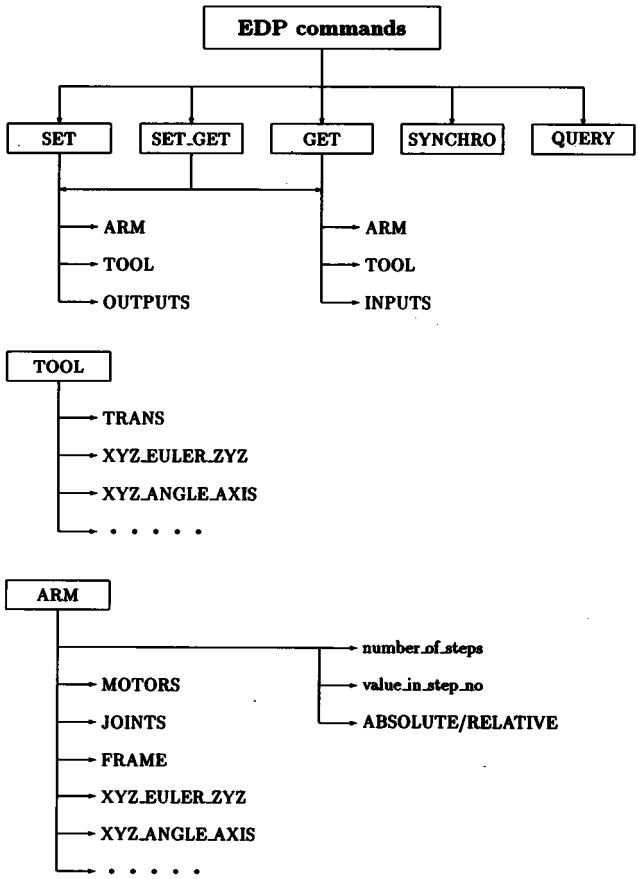

Fig. 6. List of EDP commands.

This variety of command argument combinations causes the inter-process message to have variable structure. Since at both ends of the communication channel the message buffer has to have the same structure its definition has to be shared by both communicating processes, so a single *class* is defined for both. On both ends of the communication channel different actions are performed on the buffer (one process loads it and the other unloads it and interprets its contents), so the buffer is enclosed in another *class* that is specific to each of the processes. Moreover each process performs its specific actions on the obtained data. This data and the actions (*methods*) are contained in a process specific *class*. Actually this *class* is the only one that the future user might want to alter. This would happen in a rather unusual case for a stable configuration system when, for example, a servo-controller would have to be changed, but when a new robot is added to the system a new EDP has to be written and so the joint servo-regulators and coordinate transformer for a different kinematic structure of the arm would have to be coded. The OOP methodology enabled the accumulation of the code and data that has to be altered into two *classes* (e.g. **transformer** and **regulator**) rendering the inclusion of a new robot in the system relatively simple.

Finally a single *class* is derived from the three classes, i.e.: process specific class, input and output buffer classes.

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

Hence, within a process single *object* exists which inherits its properties from all of the three component *classes.* This method of coding has been replicated for: EDP_MASTER, READING_BUFFER and all of the SERVO processes (Figure 7). In the case of EDP_MASTER the transformer *class* is responsible for all coordinate transformations and in the case of SERVO processes the **regulator** *class* stores control data and computes the servomotor control algorithm. The derived *classes* are responsible for the inter-process communication.

For a number of reasons errors can occur during computations. Three types of errors can occur:

- non-fatal errors,
- robot fatal errors and
- system fatal errors.

A reaction to each type of errors has to be different due to different causes of errors and possible remedies. Non-fatal errors are due to invalid parameters of commands external to the driver and result in computation errors. Robot fatal errors are due to improper functioning of robot hardware. From the above mentioned errors the controller must be able to recover and continue functioning. System fatal errors are caused by improper functioning of the computer network or are due to errors in the EDP software itself. In this case the operator has to be informed about the cause of the problem and the system has to be halted. In *MRROC+* the first two kinds of errors are treated as *exceptions* and adequate *exception handlers* deal with them separately. Separation of the code dealing with errors from the code handling normal operation again greatly simplifies programming, rendering the code more reliable and easier to debug and modify. Error handling depends not only on the cause of error but also on the place in code that the error has been detected. The same error occurring in different system states might need different actions. The OOP error handling capability is especially well suited to this purpose.

5. CONCLUSIONS

The proposed method of coding has several advantages. The change of the robot or of a servomotor brings about only the change in internal functioning of coordinate **transformer** or **regulator** *class* without changing the inter-process communication. In this way, assessment of the results of changes in the robot arm kinematic structure, servo-control algorithms or a type of an actuator, forces the investigator to change a relatively small portion of the overall controller code, well localised at that, and without unexpected interactions. This feature is due to proper structuring of the code and utilisation of OOP methodology. The structure of the system resulted from formal considerations which took into account the fact that each distinct component of the system should be controlled by its own process, thus limiting inter-process communication and reducing implementation code dependencies. As *classes* contain *data* and *methods* operating on it in one entity the formal parameter lists for *methods* are usually empty, what greatly simplifies coding and reduces possibilities of errors.

Out of the properties of OOP methodology enumerated in section 3 encapsulation, inheritance and exception handling proved instrumental in the implementation of the control system. It is envisaged that polymorphism will be useful in recording ECP and MP (currently they are written in C), e.g. using the same name for an action performed by different robots or sensors. Overloading of operators was successfully utilised[24] in *ZERO++* to implement robot specific infix mathematical expressions based on homogeneous transformations.

Summarising, the experiment with implementing a multi-process concurrent controller for a multi-robot system proved that utilisation of OOP paradigm significantly simplifies both the initial programming effort and reduces the time needed for modifying the code when adding robots of a new type to the system. Currently the upper layer of the controller structure, i.e. ECP and MP, are being recoded using OOP methodology. Although OOP is limited, to a great extent, only to each process, the clarity of code and its simplification due to proper structuring prove beneficial. Currently the system contains two IRb-6 robots (one mounted on a track), conveyor belt, vision system, force/torque sensor, ultrasonic and infra-red proximity sensors. The latest addition to this system is the prototype of the rigid robot,[25] which is currently thoroughly investigated by using the described controller.

It is quite probable that the two methods of implementing *RPLs,* i.e. specialised languages versus submerged languages (libraries), will be used in conjunction. The system will be designed as an open structure one with a library of processes, procedures or objects defined for programming purposes. If the system applications are known beforehand the class of tasks it will execute is known, so a specialised language for coding them can be defined. By using a compiler compiler a translator of that language can be obtained

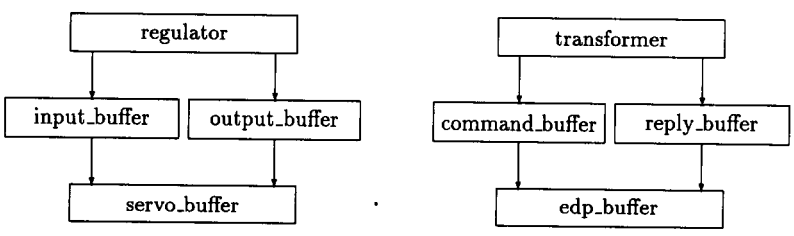

Fig. 7. Class inheritance within the SERVO and EDP_MASTER processes.

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press

relatively quickly. The produced translator would generate code invoking library routines and processes. Currently research of such specialised languages for the description of a limited class of tasks is under way. The future will show if it is a step in the right direction towards really user-friently robot programming methods.

Acknowledgements

The design of *MRROC+* has been supported by the Program in Control, Information Technology and Automatization PATIA and the preparation of this manuscript by statutory funds of Warsaw University of Technology. The author acknowledges the participation of Dr W. Szynkiewicz in the implementation of *MRROC+*.

References

[1] J.M. Hollerbach, "A Survey of Kinematic Calibration" **In**: *The Robotics Review* 1. (Eds: O. Khatib, J.J. Craig, T. Lozano-Perez) (The MIT Press, Cambridge 1989). pp. 207–241.

[2] G. Gini & M. Gini, "*ADA*: A Language for Robot Programming?" *Computers In Industry* **3,** No. 4, 253–259 (1982).

[3] R.H. Taylor, P.D. Summers & J.M. Meyer, "*AML*: A Manufacturing Language" *Int. J. Robotics Research* **1,** No. 3, 19–41 (1982).

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

[5] V. Hayward & R.P. Paul, "Robot Manipulator Control Under Unix RCCL: A Robot Control C Library". *Int. J. Robotics Research* **5,** No. 4, 94–111 (1986).

[6] J. Lloyd, M. Parker & R. McClain, "Extending the RCCL Programming Environment to Multiple Robots and Processors". *Proc. IEEE Int. Conf. Robotics & Automation* (1988) pp. 465–469.

[7] V. Hayward & S. Hayati, "KALI: An Environment for the Programming and Control of Cooperative Manipulators" *Proc. American Control Conf.* (1988) pp. 473–478.

[8] V. Hayward, L. Daneshmend & S. Hayati, "An Overview of KALI: A System to Program and Control Cooperative Manipulators" **In:** *Advanced Robotics* (Ed. K. Waldron) (Springer-Verlag, 1989) pp. 547–558.

[9] C. Zieliński, *Robot Programming Methods* (Publishing House of Warsaw University of Technology, 1995).

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

[11] C. Zieliński, "Controller Structure for Robots with Sensors" *Mechatronics* **3,** No. 5, 671–686 (1993).

[12] S. Mujtaba & R. Goldman, *AL Users' Manual* (Stanford Artificial Intelligence Laboratory, 1979).

[13] C. Zieliński, "Object Level Robot Programming Languages" **In:** *Robotics Research and Applications* (Ed.: A. Morecki et al.) (Warsaw, 1992) pp. 221–235.

[14] A. P. Ambler & D. F. Corner, *RAPT*1 *User's Manual* (Department of Artificial Intelligence, University of Edinburgh, 1984).

[15] C. Zieliński, "TORBOL: An Object Level Robot Programming Language" *Mechatronics* **1,** No. 4, 469–485 (1991).

[16] C. Blume & W. Jakob, *PASRO: Pascal for Robots* (Springer-Verlag, Berlin 1985).

[17] L.I. Lieberman & M.A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly" *IBM Journal of Research and Development* **21,** No. 4, 321–333 (July, 1977).

[18] User's Guide to VAL II. *Programming Manual,* Ver. 2.0 (Unimation Incorporated, A Westinghouse Company, August 1986).

[19] P. Corke, & R. Kirkham, "The ARCL Robot Programming System" *Proc. Int. Conf. Robots for Competitive Industries,* Brisbane, Australia, (14–16 July 1993) pp. 484–493.

[20] P. Backes, S. Hayati, V. Hayward & K. Tso, "The KALI Multi-Arm Robot Programming and Control Environment" *Proc. NASA Conf. on Space Telerobotics* (1989).

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

[22] C. Zieliński, "Sensory Robot Motions". *Archives of Control Sciences* **3,** No. 1, 5–20 (1994).

[23] C. Zieliński, "Robot Object-Oriented Pascal Library: ROOPL". *J. of Theoretical and Applied Mechanics* **31,** No. 3, 525–535 (1993).

[24] C. Pelich & F. M. Wahl, "A Programming Environment for a Multiprocessor-Net Based Robot Control Unit". *Proc. 10th Int. Conf. on High Pefformance Computing,* Ottawa, Canada (June 5–7, 1996) (on CD-ROM).

[25] J. Bidziński, K. Mianowski, K. Nazarczuk & T. Słomkowski, "A manipulator with an arm of serial-parallel structure". *Archives of Mechanical Engineering* **34,** 65–78 (1992).

[26] *Turbo C++: Getting Started* (Borland International Incorporated, 1990).

https://doi.org/10.1017/S0263574797000064 Published online by Cambridge University Press