6th Int. Conf. on Control, Automation, Robotics and Vision, ICARCV'2000, 5–8 December 2000, Singapore. (on CD-ROM)

Programming and Control of Multi-Robot Systems

Cezary Zieliński[*]

School of MPE, Nanyang Technological University, Nanyang Avenue, Singapore 639798. MCZielinski@ntu.edu.sg

Abstract

The paper presents a formalized approach to designing the structure of a controller and its programming method for a multi-robot system equipped with diverse external sensors. It shows that it is sufficient to extend a universal programming language by a single robot specific instruction to program such a system. Although, a single complex instruction would suffice, it is much more convenient to introduce two, but simpler ones. The paper shows how to implement such instructions using object-oriented approach. This approach has been used to implement Multi-Robot Research-Oriented Controller `MRROC++`.

1. Introduction

Considerable effort has been concentrated on developing new robot programming languages (RPLs), both specially defined for robots [1], [11], [12], and libraries of robot specific procedures coded in a general purpose computer programming language [1], [2], [4], [15]. 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. If new sensors are to be incorporated the hardware specific software has to be supplied and 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. This makes specialised RPLs of little use when an open structure, reconfigurable system, possibly incorporating new hardware, is considered. In such cases libraries of software modules submerged in a general purpose language are more appropriate. As robot tasks are described in terms of invocations of those modules and control flow instructions of the general purpose language supplemented by its data processing capabilities, the resulting program can be viewed as being expressed in another language – a robot programming language derived from the general purpose language. This is why the term library and language are often used interchangeably.

[*]Currently on long term leave from Warsaw University of Technology, Institute of Control and Computation Engineering, ul. Nowowiejska 15/19, 00-665 Warsaw, Poland, C.Zielinski@ia.pw.edu.pl

Usually `Pascal` or `C` are used as language platforms e.g.: `C`: `RCCL` [4], `ARCL` [2], RCI [7], `KALI` [6], `RORC` [15]; `MRROC` [15], [16]; `Pascal`: `PASRO` [1], `ROPAS` [14]; object-oriented version of `Pascal`: `ROOPL` [13]; and `C++`: `MRROC++` [19], [20].
The following discussion shows what should the extension of a general purpose language instruction set look like to transform it into a language for programming multi-robot, multi-sensor systems and how to structure the controller of such a system.

2. System Structure

A robot system is composed of three subsystems: **effectors** (e.g. manipulators), **receptors** (sensors), and the **control subsystem** (e.g. user program and variables). Effectors are any devices that can change the state of the environment, e.g. by shifting objects or machining materials. Receptors are the devices acquiring information about the environment. The control subsystem uses the information gathered by the receptors in conjunction with the information stored in its internal memory in the form of a program describing the task at hand, to influence the effectors in such a way as to execute this task. The system state s is denoted as:

$$s = < e, r, c > \qquad (1)$$

e — state of the effectors,
r — state of the receptors (hardware sensors),
c — state of the control subsystem.
For brevity, here and in the subsequent discussion, the subsystems and their state use the same symbols.
The main instructions of RPLs are the ones causing the motion of the effectors, i.e. **motion instructions**. They influence the state of the effectors. That state can be expressed in diverse terms (using different abstract notions), depending on the type of the effector and the way the system designer wants the programmer to perceive the effector. In the case of the manipulator level RPLs [3] the abstract notions that these instructions refer to are: actuator positions, manipulator joint angles or displacements, locations of coordinate frames affixed to the end-effector or the grasped objects.
Raw data obtained from hardware sensors usually cannot be used directly to control the system. It has

to be transformed. There are cases when only several hardware sensors simultaneously can deliver meaningful data for motion control (e.g. several strain gauges produce a force or a torque vector). On the other hand there are complex sensors that deliver data that has to be processed in order to obtain information that can be used in motion control (e.g. CCD camera delivering a bit-map out of which a centre of gravity of the object must be extracted to locate the best grasping position). Sensor reading transformation is called **data aggregation**. As a result a **virtual sensor reading** v is obtained:

$$v = f_v(c, e, r) \tag{2}$$

Vector function f_v is called an **aggregating function**. This function always depends on r, sometimes on c too, and on e very rarely, but for the sake of completeness all possible arguments have been enumerated.
The job of a robot system is to execute a task supplied to it in the form of a user program. If the robot system is programmed using a specialised language, the user program is read by an input device of the controller and is subsequently interpreted. Prior to reading it the program may be compiled. If the robot system is programmed using a library submerged in a general purpose language, the user program and the controller software are bound together. The controller is created for the task at hand. The user program and the controller software are compiled together to create an executable code of the controller software carrying out the task.

3. Sensor Utilization

Motion instructions in a user program cause changes of the state of effectors e. The execution of a motion instruction begins in an **initial state**, ends in a **terminal state**, and traverses a sequence of **intermediate states**. The execution of each instruction is subdivided into **steps**. Each step results in the change of system state from one intermediate state to the next. The step duration is equal to the servo sampling rate or is a multiple of that.
In each intermediate state (or while attaining it) the state of the system can be measured – **monitored** by sensors. The current state of the system can only be monitored, but the future intermediate states can be influenced – **controlled**. The initial state can be treated as a current intermediate state at the beginning of motion instruction execution. The terminal state is the current intermediate state in which the execution of the instruction terminates. This enables us to treat all kinds of states in the same manner.
Three distinct purposes of monitoring can be named:

- initial condition monitoring,
- terminal condition monitoring,
- error condition monitoring.

Let the initial state of an execution of a motion instruction be labeled i_0 and the consecutive intermediate states $i = i_0+1, \ldots, i_m$, where i_m is the label of the terminal state. If the system has executed i steps, and is currently in intermediate state s^i, the next intermediate state of effectors e^{i+1} is computed by means of the **effector transfer function** $f_e(c^i, e^i)$ or $f_e^*(c^i, e^i, v^i)$.
In the case of **initial condition monitoring** the system tests the initial condition in consecutive steps. When the condition is satisfied the monitoring ends. The most general semantics of initial condition monitoring is:

$$e^{i+1} = \begin{cases} e^i = e^{i_0} & \text{when} \\ & f_I(c^i, e^i, v^i) = false \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^{i_m} = e^{i_0} & \text{when} \\ & f_I(c^i, e^i, v^i) = true \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^{i_{m*}} = e^{i_0} & \text{when} \\ & f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{for } i = i_0, \ldots, i_m, \quad i_{m*} \le i_m \tag{3}$$

$f_I(c^i, e^i, v^i)$ – initial condition function,
$f_E(c^i, e^i, r^i)$ – error condition function,
i_0 – initial step number,
i_m – step in which f_I becomes *true* (at that moment initial condition monitoring is interrupted),
i_{m*} is the step number in which f_E is satisfied (i.e. an error occurs – instruction execution is terminated prematurely).
The error condition f_E is caused by computational errors (hence c^i as its argument), robot or sensor hardware malfunction (hence e^i and r^i). In error detection rather r^i is used directly than v^i.
Terminal condition monitoring consists in changing the system state until the terminal condition is satisfied.

$$\begin{cases} e^{i+1} &= f_e(c^i, e^i) \quad \text{when} \\ & \quad f_T(c^i, e^i, v^i) = false \bigwedge \\ & \quad f_E(c^i, e^i, r^i) = false \\ e^i &= e^{i_m} \quad \text{when} \\ & \quad f_T(c^i, e^i, v^i) = true \bigwedge \\ & \quad f_E(c^i, e^i, r^i) = false \\ e^i &= e^{i_{m*}} \quad \text{when} \\ & \quad f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{for } i = i_0, \ldots, i_m, \quad i_{m*} \le i_m \tag{4}$$

$f_e(c^i, e^i)$ – effector transfer function (it does not depend on v^i, because here the state is only monitored and not controlled using sensor readings),
$f_T(c^i, e^i, v^i)$ – terminal condition function,
i_m – step number in which f_T becomes *true* (at that moment terminal condition monitoring is interrupted),
i_{m*} – step number in which f_E is satisfied, i.e. an error occurs – instruction execution has to be terminated prematurely.

The **control of future intermediate states** is usually combined with monitoring of the terminal condition, so it can be expressed as:

$$\begin{cases} e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{when} \\ & f_T(c^i, e^i, v^i) = false \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_m} & \text{when} \\ & f_T(c^i, e^i, v^i) = true \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_{m*}} & \text{when} \\ & f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{for } i = i_0, \ldots, i_m, \quad i_{m*} \leq i_m \tag{5}$$

where $f_e^*(c^i, e^i, v^i)$ is the effector transfer function (it depends on v^i, because the state is not only monitored, but also controlled using sensor readings).
If (3), (4), and (5) are combined, the semantics of the most general motion instruction becomes:

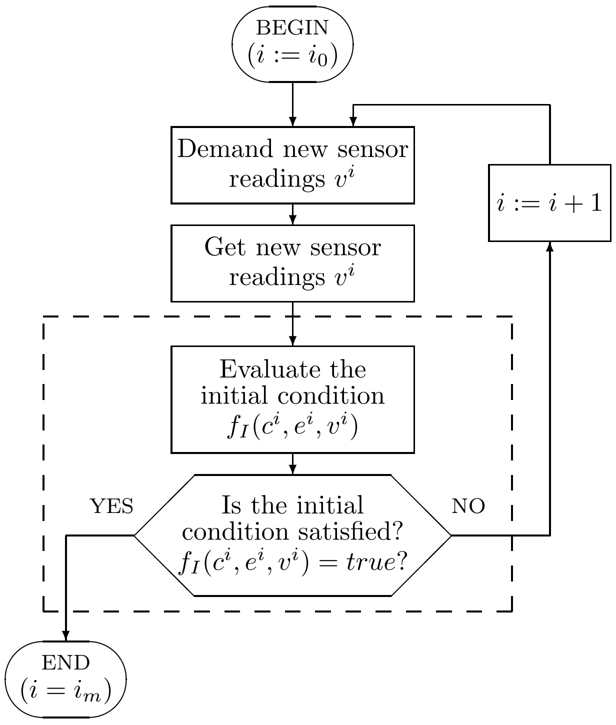
$$\begin{cases} e^{i+1} = e^i = e^{i_0} & \text{when} \\ & f_I(c^i, e^i, v^i) = false \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ & \text{then } i = i_0, \ldots, i_k \\ e^i = e^{i_k} & \text{when} \\ & f_I(c^i, e^i, v^i) = true, \\ & \text{then } i = i_k \\ e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{when} \\ & f_T(c^i, e^i, v^i) = false \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ & \text{then } i = i_k, \ldots, i_m \\ e^i = e^{i_m} & \text{when} \\ & f_T(c^i, e^i, v^i) = true, \\ & \text{then } i = i_m \\ e^i = e^{i_{m*}} & \text{when} \\ & f_E(c^i, e^i, r^i) = true, \\ & \text{then } i = i_{m*} \end{cases} \tag{6}$$

Usually the most general form of the motion instruction is not implemented, because rarely both the initial and the terminal conditions are monitored in one motion instruction. It is quite reasonable to assume that the initial condition monitoring will be conducted separately. Moreover, as it has been mentioned, terminal condition monitoring is combined with control of future intermediate states. In this way two separate instructions are obtained:
`Wait` – monitoring the initial condition (3), Fig. 1,
`Move` – monitoring the terminal condition and simultaneously controlling the future states (5), Fig. 2.
It should be noted that error condition monitoring is not included in the flow diagrams (Fig. 1, Fig. 2) – this is handled as an exception.

4. Control System

The system state s (1) is further decomposed by taking into account that it consists of distinct effectors and virtual sensors.

$$s = \; < e_1, \ldots, e_{n_e}, v_1, \ldots, v_{n_v}, c > \tag{7}$$



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

Fig. 1
`Wait` INSTRUCTION FLOW CHART

n_e – number of effectors,
n_v – number of virtual sensors.

The control subsystem calculates the next effector state. This can be done by a single centralised control subsystem, but a much better and clearer structure is obtained, if the state of the control subsystem c is partitioned into $n_e + 1$ parts.

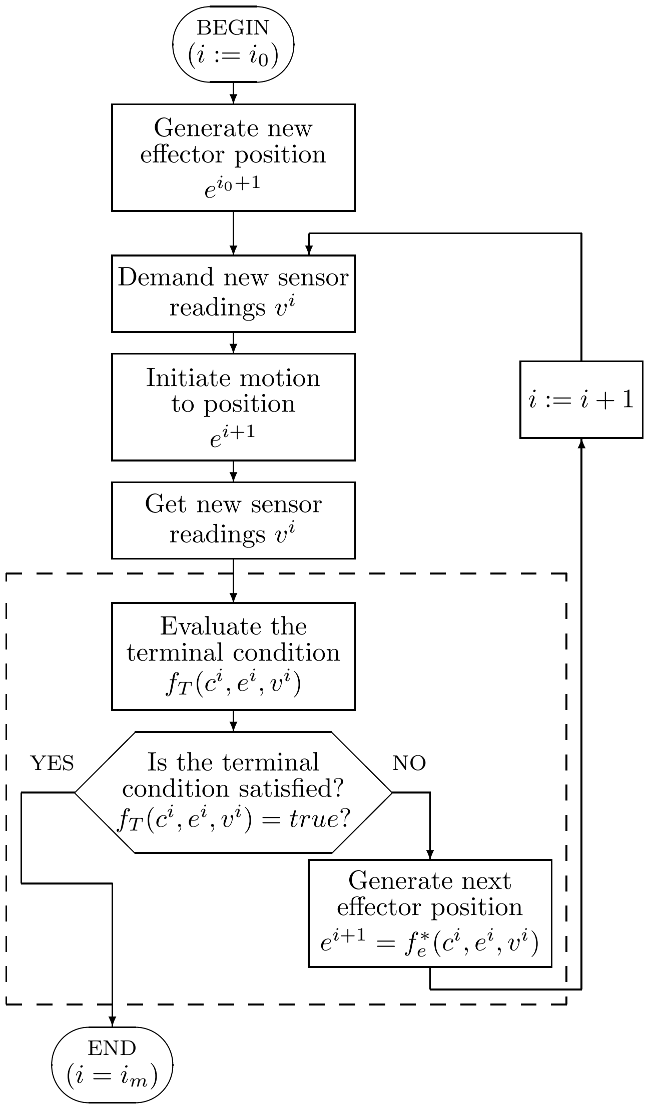
$$c = \; < c_0, c_1, \ldots, c_{n_e} > \tag{8}$$

Let each subsystem c_l, $l = 1, \ldots, n_e$, be responsible for controlling an effector associated with it, and the subsystem c_0 coordinate all effectors. Hence, with each of the effectors $e_l, l = 1, \ldots, n_e$ an **Effector Control Process** (ECP) 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.
The interconnections between the system components can be deduced from the general forms of effector and control subsystem transfer functions.

$$e^{i+1} = f_e^*(c^i, e^i, v^i) \tag{9}$$

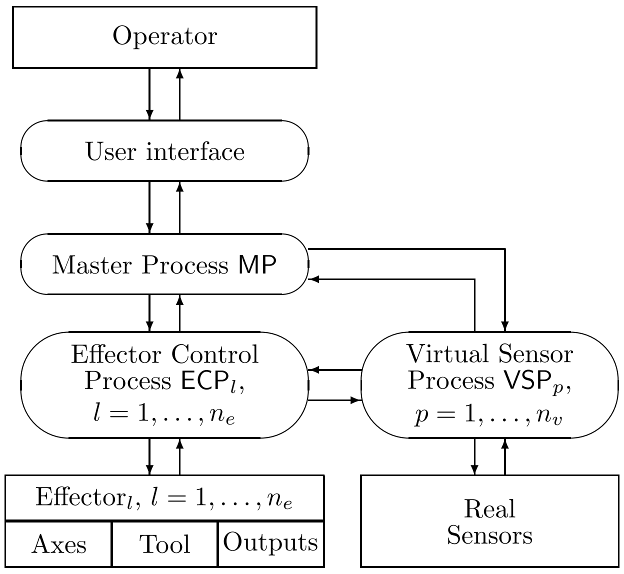
$$c^{i+1} = f_c(c^i, e^i, v^i) \tag{10}$$

An interconnection between two subsystems has to be produced, if the next state of one system component (i.e. its transfer function) depends on the current state of the other component (i.e. takes the state of this element as a transfer function argument). Reasonable forms do not cause, on the one hand,



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 $f_T(c^i, e^i, v^i)$
YES
Is the terminal condition satisfied? $f_T(c^i, e^i, v^i) = true$?
NO
Generate next effector position $e^{i+1} = f_e^*(c^i, e^i, v^i)$
$i := i + 1$
END
$(i = i_m)$

Fig. 2
`Move` INSTRUCTION FLOW CHART



Operator
User interface
Master Process MP
Effector Control Process ECP$_l$, $l = 1, \dots, n_e$
Virtual Sensor Process VSP$_p$, $p = 1, \dots, n_v$
Effector$_l$, $l = 1, \dots, n_e$
Axes
Tool
Outputs
Real Sensors

Fig. 3
HIERARCHICAL STRUCTURE OF A MULTI-ROBOT CONTROLLER

too many interconnections and, on the other hand, enable the execution of any control algorithm. Let:

$$e_j^{i+1} = f^*_{e_j}(c_0^i, c_j^i, e_j^i, v_1^i, \dots, v_{n_v}^i) \tag{11}$$

$$c_0^{i+1} = f_{c_0}(c_0^i, c_1^i, \dots c_{n_e}^i, e_1^i, \dots, e_{n_e}^i, v_1^i, \dots, v_{n_v}^i) \tag{12}$$

$$c_j^{i+1} = f_{c_j}(c_0^i, c_j^i, e_j^i, v_1^i, \dots, v_{n_v}^i) \tag{13}$$

This produces the structure presented in Fig. 3. Each ECP executes its `Move` and `Wait` instructions, so adequate functions: f_{e_l}, $f^*_{e_l}$, f_{I_l}, f_{T_l}, f_{E_l}, $l = 1, \dots, n_e$, are computed by it. The MP also executes its `Move` and `Wait` instructions and so it computes: f_{e_0}, $f^*_{e_0}$, f_{I_0}, f_{T_0}, f_{E_0}. The ECP and MP instructions communicate through c_0. In the case of tight cooperation between robots (e.g. transfer of a rigid object by several robots) most of the control and monitoring tasks will be done by the `Wait` and `Move` instructions of the MP. If the robots cooperate loosely or do not interact at all, control and monitoring will be executed mainly by appropriate ECPs. The `Move` instructions of the MP produce control data for the `Move` instructions of the ECP, which in turn generate the control signals for the servos. This structure was utilised in MRROC++.

Each virtual sensor $v_p, p = 1, \dots, n_v$ is implemented as a **Virtual Sensor Processes** (VSP) running concurrently to the other processes. In consequence of (2):

$$v_p^i = f_{v_p}(c_0^i, c_l^i, e_l^i, r^i) \tag{14}$$

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 (if any) influences directly a virtual sensor, because an effector can change the locations of the hardware sensors that are affixed to it.

The processes communicate through messages. The communication of each ECP with the VSPs it uses can be of two kinds: **interactive** and **non-interactive**. In the case of **interactive communication** the ECP sends a data request message to an adequate VSP. The VSP reads the hardware sensors, aggregates the obtained data (computes f_{v_p} using (14)) and sends the result to the ECP. In the case of **non-interactive communication** the VSP reads the hardware sensors, aggregates data and leaves the resulting reading in a buffer without any request from any ECP. The ECP can access the latest sensor data at any moment by reading the buffer where the aggregated data is stored. In both cases the ECP can obtain sensor data in each step i or, if necessary, less frequently.

5. Implementation Considerations

The hardware dependent subprocesses (ECPs and VSPs) change only when new hardware (e.g. robot or sensor) is added to the system. The upper layers of ECPs and MP change whenever the system has to execute a new task. Although `MRROC++` enables easy incorporation of new hardware, the programmer usually deals with changing tasks for a stable hardware configuration. That is why the coding of motion instructions at the level of ECP and MP has to be as simple as possible.

Because there is a contradiction between changing numbers of hardware devices used in each motion, and preferably constant number of `Move` and `Wait` instruction arguments, it was decided that rather `robot` (effector) and `sensor` object lists will be the formal parameters of instructions than `robot` or `sensor` objects themselves. Moreover, object classes named `condition` and `generator` are needed (Fig. 4). The `condition` supplies methods for computing the initial condition f_I (executes the part of the algorithm in Fig. 1 circumscribed by the dashed line). The `generator` is responsible for computing the transfer function f_e^* as well as the terminal condition f_T (executes the part of the algorithm in Fig. 2 circumscribed by the dashed line). The `Move` and `Wait` instructions (procedures) use within their bodies: `robot`, `sensor`, `condition` and `generator` base classes, but at run-time they invoke descendant objects of these classes. The programmer creates descendants of `robot` and `sensor` classes in conformance with the hardware used and descendants of `condition` and `generator` according to the task at hand. This is due to polymorphism.

The programmer creates the MP by supplying adequate robot and sensor lists, as well as conditions and generators. It should be noted that the variability of system structure has been contained in several independent objects. For each motion the programmer points out which `robot` and which `sensor` objects will be used. Each specific `robot` or `sensor` "knows" how to interact with its MP/ECP or VSP. Path generation for the cooperating robots is done by the specific `generator` that is supplied by the programmer. If tight cooperation is needed the MP level generator supplies a pose for each step of the trajectory, and if the robots are loosely coupled or do not interact it supplies only a few poses and the ECPs do the path interpolation.

For a number of reasons errors can occur during task execution. Those are:

- non-fatal errors,
- hardware fatal errors,
- system fatal errors.

A reaction to each type of error has to be different. Non-fatal errors are due to programmer's mistakes and result in computation errors. Hardware fatal errors are due to improper functioning of hardware.

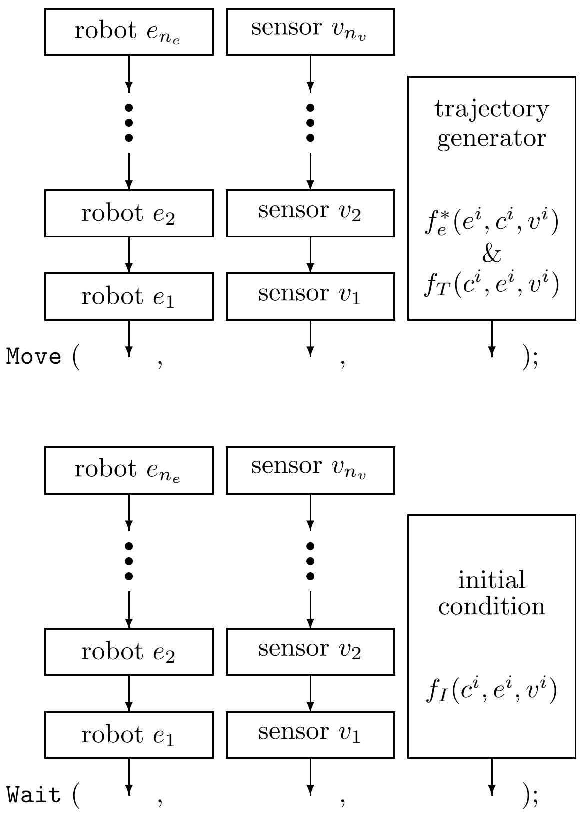


Fig. 4
MRROC++ MOTION INSTRUCTIONS

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. 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 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 the code that the error has been detected. The same error occurring in different system states might need different actions.

6. Conclusions

The reduction of RPL implementation effort suggests the utilisation of general purpose programming languages extended by robot specific libraries rather than the definition of specialised languages. Initially `MRROC` 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. Objects, i.e. data and code operating on it, caused a significant reduction of function parameter lists, so a much more intelligible code resulted. Last but not least, exception handling enabled the separation of the code processing normal system functioning from the code dealing with error situations. Finally, the formal approach 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.

`MRROC++` can currently control modified ASEA type IRb-6 and IRb-60 robots, a prototype serial-parallel structure robot [8] and the Polycrank robot [9]. Force/torque, ultrasonic [10] 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 IRb-6 robots [17] and engraving inscriptions in wood by the prototype robot equipped with a milling tool were among them.

Acknowledgments

The author gratefully acknowledges the participation of Dr W. Szynkiewicz in developing `MRROC++` at Warsaw University of Technology.

References

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

[2] 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.

[3] Gini G., Gini M.: *ADA: A Language for Robot Programming?* Computers In Industry, Vol.3, No.4, 1982. pp.253–259.

[4] 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.

[5] Hayward V., Hayati S.: *KALI: An Environment for the Programming and Control of Cooperative Manipulators*. Proc. American Control Conf., 1988. pp.473-478.

[6] 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.

[7] Lloyd J., Parker M., McClain R.: *Extending the RCCL Programming Environment to Multiple Robots & Processors*. Proc. IEEE Int. Conf. Robotics & Automation, 1988. pp.465-469.

[8] Nazarczuk K., Mianowski K., Oledzki 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.

[9] Nazarczuk K., Mianowski K.: *Polycrank – Fast Robot Without Joint Limits*, Proc. 12th CISM-IFToMM Symposium Ro.Man.Sy'98, 6–9 July 1998, pp. 317-324.

[10] Pacut A., Brudka M., and Jaworski M.: *Neural Processing of Ultrasound Images in Robotic Applications*, IEEE Int. Workshop on Emerging Technologies, Intelligent Measurement and Virtual Systems for Instrumentation and Measurements ETIMVIS'98, St. Paul, USA, May 1998. pp. 59–66.

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

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

[13] Zieliński C.: *Robot Object-Oriented Pascal Library: ROOPL*. J. of Theoretical and Applied Mechanics, Vol.31, No.3, 1993. pp.525-535.

[14] Zieliński C.: *Sensory Robot Motions*. Archives of Control Sciences, Vol.3, no.1, 1994. pp.5-20.

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

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

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

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

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

[20] Zieliński C.: *The MRROC++ System*, 1st Workshop on Robot Motion and Control, RoMoCo'99, 28–29 June, 1999, Kiekrz, Poland. pp.147–152.