Advanced Robotics Vol.15, No.1, 2001, pp.71–95

BY HOW MUCH SHOULD A GENERAL PURPOSE PROGRAMMING LANGUAGE BE EXTENDED TO BECOME A MULTI-ROBOT SYSTEM PROGRAMMING LANGUAGE?

Cezary Zieliński[1]

(full paper)

[1]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

Abstract

The paper gives the answer to the question formulated in the title for the case of manipulator level languages [1]. Theoretical considerations show that regardless of the type and number of robots and cooperating devices used in the system as well as irrespective of the number and kind of sensors included in the system, a general purpose language has to be extended by a single instruction with a rather complex semantics. Due to this complexity it is more convenient to introduce two, but much simpler instructions. The paper also presents the method of implementing those instructions in a hierarchical distributed control system. The presented approach has been used in the implementation of `MRROC/MRROC++` robot programming libraries/languages. The programmer uses predefined library modules to construct the controller structure solving a specific multi-robot task. The structure is fine-tuned to the task at hand by supplying adequate motion generators in a plug-in fashion. The practical validity of the formal approach followed in the implementation of `MRROC` and `MRROC++` has been positively verified on diverse robotic tasks.

Keywords

robot programming, robot programming languages/libraries, multi-robot systems

1 INTRODUCTION

Late seventies and the eighties witnessed an outbreak of the number of robot languages defined and implemented. Some effort was put into standardization of robot programming, but mainly for industrial purposes (e.g. `IRDATA` [2], `MMS` [3], `SLIM/STROLIC` [4]). Unfortunately most of the new languages usually had a single-site implementation. As subsequently new definitions did not introduce significant advantages to the already existing languages, the interest of the scientific community in new methods of robot programming gradually faded away. No formal theory of robot programming emerged. This paper tries to look once again at the field of robot programming taking into account the new developments that took place in computer science during the nineties. Those especially include the widespread use of object-oriented programming paradigm.

What differs **robot programming languages** (RPLs) from general purpose languages are the motion instructions, i.e. the ones causing the motion of the effectors. Geometric data types and world model instructions are introduced into RPLs to create a tool for a simplified specification of the goal of end-effector motion. Both implicit (e.g. `RAPT` [5], `ROBEX` [2], `TORBOL` [6]) and explicit (e.g. `WAVE` [7], `AL` [8], `AML` [9], `PasRo` [10], `VAL II` [11], `SRL` [2], `RCCL` [12], `KALI` [13, 14, 15]) specifications are possible. Implicit motion instructions rely on the ability of the run-time system to evaluate the necessary arguments taking into account the current state of the world model contained within this system. In the case of explicit specification the arguments are delivered in the form of expressions containing variables of geometric types. Apart from goal specification, motion instructions need the definition of diverse motion parameters (e.g. speed, duration, intermediate locations, precision of termination). Those parameters are either directly incorporated into motion instruction optional parameters or separate instructions set them. Quite often diverse motion instruction keywords are used to differentiate between types of interpolation utilised.

Simple sensor monitoring was incorporated into motion instruction execution quite early (e.g. `AL` [8], `AML` [9], `SRL` [2]). Fusion of data obtained from different sensors has been the subject of separate intensive research. Usually the RPL limited itself to the sensors that had been included in the robotic system for which the language was defined.

The RPL and the structure of the executing controller are strongly influenced by the trajectory generation mechanism employed. One of the most successful approaches to that problem took the position description from `PAL` [16, 12], world model manipulation from `AL` [8], and started evolving from `RCCL` [12] onto `Multi-RCCL` [17, 18] and was generalised in `KALI` [14, 15]. Initially a transform equation [16] for a single robot was considered. It contained a drive transform [12] which was interpolated in such a way that the goal location was reached. Later this approach was generalised to several robots. A motion system [14], which could consist of a set of transforms describing kinematic loops sharing a common drive transform, was introduced. Sensor monitoring and active utilisation of sensor readings in motion control was taken into account by introducing into the kinematic loops functionally defined transforms. Different interpolation techniques have been tested within the general trajectory generation framework offered by `RCCL` and `KALI`. `RCCL` and `KALI` installations are fairly widespread in robotics research sites [18].

This paper shows an alternative approach to the problem of the generic trajectory generation framework and the motion instructions handled by the control system, so the resulting structure of the controller is somewhat different from that of `RCCL` and `KALI`. `RCCL` and `KALI` rely on an asynchronous (background) user task and synchronous manipulator control tasks communicating through motion queues [14]. In the presented solution all the processes can be synchronous and queuing mechanism is not utilised. Due to a fast sampling rate the behaviour of the system components, from external perspective, is asynchronous, although the implementation is synchronous.

The formal approach presented in the paper resulted in the implementation of a library of `C++` objects and functions named `MRROC++` (**Multi-Robot Research-Oriented Controller**). The library contains building blocks necessary for the construction of robot controllers dedicated to the execution of specific tasks. The architecture of those controllers is also defined, so `MRROC++` can also be treated as a system. A library of functions extends the capabilities of the initial programming language (`C++` in this case), so sometimes libraries are treated as language extensions. If the extensions create capabilities to express programs in new domains, e.g. robotics, then the base language and the library is treated as a new, domain specific, programming language. This line of reasoning has been followed by those who treat `RCCL`, `Kali`, `PasRo` or `MRROC` as programming languages.

2 ROBOT SYSTEM DECOMPOSITION

A robot system S is composed of three subsystems:

$$S =< C; E; R >, \qquad (1)$$

C – **control subsystem**, (i.e. memory: variables, program and program execution control flow),

E – **effectors** (manipulator arm or arms, tool and the devices cooperating with the robot),

R – **receptors** or **real sensors**.

Real sensors include all the measuring devices gathering information from the environment of the system. Devices for measuring the internal state of the system (e.g. position encoders, resolvers) are not treated here as sensors. They supply data about the state of E.

The state s of the system S is denoted as:

$$s =< c; e; r >, \tag{2}$$

c – state of the control subsystem C,

e – state of the effectors E,

r – state of the real (hardware) sensors R.

The main instructions of RPLs are the ones causing the motion of the effectors, i.e. **motion instructions**. In the case of a robot 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.

Data obtained from real sensors usually cannot be used directly in robot motion control. For instance, to control the arm motion, only the location of the centre of gravity of an object to be grasped would be necessary. In the case of such a complex sensor as a camera a bit-map has to be processed to obtain the above mentioned location. In some other cases a simple sensor in its own right would not suffice to control the motion (e.g. a single strain gauge), but several such sensors deliver meaningful data. The process of extracting meaningful information for the purpose of motion control is named **data aggregation** and is performed by **virtual sensors** V. As a result virtual sensor readings v are obtained:

$$v = f_v(c, e, r) \tag{3}$$

Vector function f_v is termed an **aggregating function**.

3 SENSOR UTILIZATION

The job of a robot system is to execute a task supplied to it in the form of a program. Motion instructions in a 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**. Currently robots are digitally controlled, so 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. Usually the duration of each step is either equivalent to the servo sampling rate (e.g. $1ms$) or a low integer 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** (Fig. 1). 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, so the initial and the terminal states are no different from the intermediate states.

Three distinct purposes of monitoring are (Fig. 2):

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

In the case of **initial condition monitoring** the system executes consecutive steps waiting for the initial condition to be satisfied, so that the motion can proceed. The most general semantics of initial condition monitoring is:

$$e^{i+1} = \begin{cases} e^{i} = e^{i_0} & \text{when} \quad f_I(c^i, e^i, v^i) = false \ \wedge \ f_E(c^i, e^i, r^i) = false \\ e^{i_k} = e^{i_0} & \text{when} \quad f_I(c^i, e^i, v^i) = true \ \wedge \ f_E(c^i, e^i, r^i) = false \\ e^{i_{k*}} = e^{i_0} & \text{when} \quad f_E(c^i, e^i, r^i) = true \end{cases} \qquad (4)$$
$$\text{for } i = i_0, \ldots, i_k, \quad i_{k*} \leq i_k$$

i	–	current step number,
i_0	–	initial step number,
$f_I(c^i, e^i, v^i)$	–	initial condition (Boolean value function),
$f_E(c^i, e^i, r^i)$	–	error condition (Boolean value function),
i_k	–	step in which $f_I(c^i, e^i, v^i)$ becomes *true*, and so the initial condition monitoring is interrupted,
i_{k*}	–	step number in which $f_E(c^i, e^i, r^i)$ is satisfied (i.e. an error occurs, so the instruction execution has to be terminated prematurely).

In the above definition, as well as in the following ones, the next effector state e^{i+1} is computed by taking into account the part of the definition for which the associated condition is fulfilled in the current step i. Only one condition is true in each step i, so the next effector state e^{i+1} is evaluated uniquely. The next effector state e^{i+1} is evaluated iteratively for each step $i = i_0, \ldots$, until the currently monitored condition is fulfilled. Each definition contains the specification of the terminal effector state e^i both for normal termination (i.e. when the currently monitored condition is fulfilled) and abnormal termination (i.e. when error condition is detected). The reason for this is to explicitly label the terminal effector state and to distinguish between normal and abnormal instruction execution termination. Moreover, it should be noted that from the engineering point of view e^{i+1} is attained in two phases. First, the next state is computed according to the definitions presented here and the result e_c^{i+1} of those computations becomes the set value for the servos. Second, the servos force the effectors to attain the state $e^{i+1} = e_c^{i+1}$. Both phases are performed in one step (usually servo sampling time or a low multiple of that). For the sake of brevity the intermediate stage has been excluded from the presented definitions.

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.

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

$f_e(c^i, e^i)$	–	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 (Boolean value function),
i_m	–	step number in which $f_T(c^i, e^i, v^i)$ becomes *true*, and so the terminal condition monitoring is interrupted,
i_{m*}	–	step number in which $f_E(c^i, e^i, r^i)$ is satisfied, i.e. an error occurs, so the 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:

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

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 (4), (5), and (6) are combined, the semantics of the most general motion instruction results:

$$\begin{cases} e^{i+1} = e^i = e^{i_0} & \text{when} \quad f_I(c^i, e^i, v^i) = false \quad \wedge \quad f_E(c^i, e^i, r^i) = false \\ & \text{for } i = i_0, \ldots, i_k \\ e^i = e^{i_k} & \text{when} \quad f_I(c^i, e^i, v^i) = true, \\ & \text{for } i = i_k \\ e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{when} \quad f_T(c^i, e^i, v^i) = false \quad \wedge \quad f_E(c^i, e^i, r^i) = false \\ & \text{for } i = i_k, \ldots, i_m \\ e^i = e^{i_m} & \text{when} \quad f_T(c^i, e^i, v^i) = true, \\ & \text{for } i = i_m \\ e^i = e^{i_{m*}} & \text{when} \quad f_E(c^i, e^i, r^i) = true, \\ & \text{for } i = i_{m*} \end{cases}$$
$$\text{for } i = i_0, \ldots, i_k, \ldots, i_m; \quad i_{m*} \leq i_m \tag{7}$$

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 (with semantics (4) and the respective flow chart in Fig. 3),

`Move` – monitoring the terminal condition and simultaneously controlling the future states (with semantics (6) and the respective flow chart in Fig. 4).

It should be noted that error condition monitoring is not included in the flow diagrams (this is handled as an exception). The error condition has to be monitored throughout the entire execution of all instructions of the program. The form of f_E does not have to be formulated analytically, if error condition monitoring is done in the background of the user program execution. Whenever the user program detects a specific error it throws an exception and the error handling code takes care of the necessary reaction. Technically speaking f_E is a conjunction of all specific error situations. As each situation is handled separately the global analytic form of f_E does not have to be supplied neither by the user nor the one who implements the system.

4 MRROC++

The system S, as described by (1), is further decomposed by taking into account that there are n_e effectors $E_j, j = 1, \ldots, n_e$, and that rather aggregated sensor readings v than real sensor readings r are used by the control subsystem to compute a motion trajectory, so n_v virtual sensors $V_l, l = 1, \ldots, n_v$ are considered. The control subsystem calculates the next effector state. Obviously those computations can be done by a single centralised control subsystem, but a much better and clearer structure is obtained, if a hierarchical distributed control subsystem is considered. In this case the control subsystem C is partitioned into $n_e + 1$ parts, where there is a single coordinator C_0 and n_e parts C_j responsible for control of each effector E_j, $j = 1, \ldots, n_e$. As a result the following is obtained:

$$S = \ < C_0, C_1, \ldots, C_{n_e}; E_1, \ldots, E_{n_e}; V_1, \ldots, V_{n_v} > \tag{8}$$

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, too many interconnections and, on the other hand, enable the execution of any control algorithm. In the considered case the state of each effector and of each part of the control subsystem, was chosen to evolve in the following way:

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

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

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

The general structure the Multi-Robot Research-Oriented Controller `MRROC++` (Fig. 5) resulted. Each subsystem C_j, $j = 1, \ldots, n_e$, 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 $E_j, j = 1, \ldots, n_e$ an **Effector Control Process** (ECP) is associated. Its state is expressed by c_j, $j = 1, \ldots, n_e$. The coordinating process is called the **Master Process** (MP) and its state is expressed by c_0.

Each virtual sensor $v_l, l = 1, \ldots, n_v$ is implemented as a **Virtual Sensor Processes** (VSP) running concurrently to the other VSPs and ECPs. In consequence of (3)

$$v_l^i = f_{v_l}(c_0^i, c_j^i, e_j^i, r^i) \tag{14}$$

is obtained, where e_j is the state of the *j-th* effector (the one associated with v_l). Here it is assumed that only a single effector influences directly a virtual sensor, because only this effector can directly change the configuration of the real sensors that are mounted on it.

The execution of `MRROC++` based controllers is supervised by the QNX ([19]) distributed real-time operating system. The processes can be assigned either to a single computer or to several computers connected into a network. The basic method of scheduling processes within a single computer that QNX uses is the prioritized FIFO algorithm. That can be switched to round-robin scheduling of processes with priorities. For a single robot controller executing a task that is not very time demanding one computer suffices. In the case of multiple robot configuration each robot will require its own computer. In this case each robot's ECP, as well as the VSPs used by the ECP, will run in a time sharing fashion on the computer allotted to that robot, and the user interface and the MP may run on a separate node, but they can also share the computer with one of the robots. In the case of a single robot configuration either all the processes share one computer or the user interface and MP have their own.

The processes communicate through messages (`Send-Receive-Reply` [19]). The communication of each ECP with the Virtual Sensor Processes 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 Virtual Sensor Process. The VSP reads the real sensors, aggregates the obtained data (computes f_{v_l} using (14)) and sends the result to the ECP. 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 ECP. An ECP can access the latest sensor data immediately by reading the buffer where the aggregated data is stored.

As the semantics of the `Wait` and `Move` instructions formulated by (4) and (6) are implementation independent, they do not take into account the partitioning of the control subsystem into parts. The following discussion will resolve this problem. From (9), (10) and (12) it is evident that only the coordinator (C_0, MP) perceives all of the effectors. From (11) and (13) it is evident that all components C_j (ECP$_j$) perceive only one effector each, i.e. E_j. The above mentioned problem is resolved, if in each process that perceives an effector E_j, an abstract **image** of that effector is created. In this way the coordinator (MP) must contain the images of all effectors and each ECP_j must posses the image of E_j. Each process operates on its own images of effectors and as a result of this activity sends motion commands to images of lower level processes or finally directly to the control hardware. The MP executes its `Move` and `Wait` instructions on the images of its effectors and so it computes: $f^*_{e_0}$, f_{I_0}, f_{T_0}. Each ECP also executes its `Move` and `Wait` instructions, so adequate functions: $f^*_{e_j}$, f_{I_j}, f_{T_j}, $j = 1, \ldots, n_e$, are computed. These functions within each level operate on adequate images and produce set values, or commands (decisions) for lower level images housed in subordinate processes or for the control hardware itself. By using the same principle each virtual sensor used by MP or ECP_j has to be reflected in that process, so an image of each virtual sensor used has to be created in those processes.

The images of effectors are created within the memory resources of a process, hence MP will contain the images of all the effectors within C_0. This enables it to compute the next desired state for each of the effectors. An ECP_j contains within C_j the image of only one effector, that is E_j, and so upon the guidance of the MP it can compute the modified value of the next desired effector state or simply transmit (for execution) the one computed by MP. The influence of MP over the ECP_j can be nil. In that case the computation of the next desired state of effector E_j is the sole responsibility of ECP_j. ECP_j updates its image of the state of E_j on the basis of information obtained from the hardware, and also transmits that information to the MP.

5 MOTION INSTRUCTIONS

The ECPs and VSPs can also be further partitioned into subprocesses in complex cases. In `MRROC++` from each ECP an **Effector Driver Process** (EDP) has been extracted. ECP_j is responsible for executing the user's task associated with effector E_j, and EDP_j is strictly a hardware dependent driver responsible for executing such jobs as: direct and inverse kinematics, as well as for servo-control. New hardware dependent subprocesses of ECPs (i.e. EDPs) and VSPs have to be supplied only when new hardware (e.g. robot or sensor) is added to the system. The ECPs and MP change whenever the system has to execute a new task. MP and ECPs operate only on images of effectors, so they use only abstract notions such as end-effector position expressed in terms of, e.g. homogeneous transformations or Cartesian positions and Euler angles. It is the responsibility of EDPs to transform those abstract notions into joint angles and subsequently into motor shaft positions. In this way MP and ECPs are readily portable to other platforms using the same operating system, but different types of robots. Obviously new EDPs have to be supplied for new types of robots. 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.

As the current version of `MRROC++` mainly controls robots, instead of using the generic concept of effectors the term robot is utilised. From the discussion of previous sections it can be concluded that two motion instructions (`Move` and `Wait`) have to be supplied. Their structure should not vary with the number of robots or sensors used. Only functions f_e^*, f_I and f_T change from motion to motion. Because there is a contradiction between changing numbers of hardware devices used in each motion, and preferably constant number of instruction arguments, it was decided that robot and sensor lists will be the formal parameters of instructions and not robots or sensors themselves. Robot and sensor images on the MP level are object classes (i.e. data and code operating on it) with the capability of influencing and reading the states of adequate ECPs and VSPs respectively. By looking at the flow diagrams of the `Move` (Fig. 4) and `Wait` (Fig. 3) instructions one can notice that there are compact portions, delimited by dashed lines, responsible either for computation and testing of f_{I_0}, or computation of $f_{e_0}^*$, f_{T_0} and testing of f_{T_0}. This suggests that two program entities (e.g. objects) can be produced to handle the tasks of:

- initial condition monitoring
- terminal condition monitoring and simultaneous generation of the next effector state,

so object classes named `condition` and `generator` have been introduced. 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 those classes. The programmer creates descendants according to the task at hand. All this is possible due to polymorphism.

The `Move` and `Wait` procedures at the MP level are presented in Fig. 6. The programmer creates the MP by supplying adequate robot and sensor lists (images of robots and sensors), as well as initial conditions and motion 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 robots and sensors will be used. This is done by supplying the above mentioned lists. Each specific `robot` or `sensor` “knows” how to interact with its ECP or VSP. On the ECP level the `Move` and `Wait` instructions, instead of a `robot` list, require a single `robot` as one of their arguments.

6 MOTION GENERATORS

An interesting point is that motion generators can be classified on the basis of the arguments of the function f_e^*. A separate classification is needed for the coordinator (MP) motion generators ($f_{e_{MP}}^*$) and a separate for all ECP motion generators ($f_{e_{j_{ECP}}}^*$). From (9) it can be deduced that there are eight possibilities at the most on the MP level, because there are three arguments: c, e, v. Each one of them can either be present or absent. Only the following five cases are meaningful: $f_{e_{MP}}^{*0}(c_0)$, $f_{e_{MP}}^{*1}(c_0, v)$, $f_{e_{MP}}^{*2}(c_0, e)$, $f_{e_{MP}}^{*3}(c_0, e, v)$, $f_{e_{MP}}^{**}() = const$. In the first four cases c_0 is used to compute the next (demanded) effector state e^{i+1}. In the fifth case it is missing, so no computations can be performed. This means that only a certain constant is sent to the lower level. The ECPs operate independently of each other and the MP, but they need initial activation to know when to start their jobs. The activation is caused by sending this constant. Obviously, when c_0 is not present in the argument list of $f_{e_{MP}}^*$, no computations can be carried out, so the information contained in e and v cannot be utilised. When c_0 is in the argument list, four possibilities arise, with all combinations of arguments e and v. With v present, the motion generator modifies a predefined trajectory (e.g. a taught-in trajectory) or generates a completely new one on the basis of sensory information (e.g. unknown contour following). When e is present in the argument list the motion generation takes into account high level effector state feedback.

Similar considerations are valid for the ECP level. Formula (11) show that function $f_{e_{ECP}}^*(c_0^i, c_j^i, e_j^i, v^i)$ has four arguments, but c_j must always be present, because unlike on the MP level the computations cannot be delegated to a lower level. This produces eight possible cases $f_{e_{j_{ECP}}}^{*k}$, $k = 0, \ldots, 7$ – all of them valid. When c_0 is not present in the argument list the effectors act independently – virtually no contact with MP is necessary. The action initiating constant sent by MP to ECP is neglected, as it initiates the overall operation of ECP rather than a specific action of the low level motion generation performed by $f_{e_{j_{ECP}}}^{*k}$. If c_0 is present, then the effectors are coordinated by MP, i.e. they cooperate. Two forms of cooperation are possible:

- loose – where the effectors are synchronised in time and space sporadically,
- tight – where the effectors execute a synchronous movement (e.g. jointly transfer a rigid object).

In the case of loose cooperation the coordinator MP transmits only decision information (an item from a finite set), and in the case of tight cooperation numeric information (describing the location to be attained) is being sent to the ECPs.

When e_j is present in the argument list of $f^{*k}_{e_{j_{ECP}}}$, the ECP level effector state feedback is taken into account during motion generation. An obvious example of that is any form of interpolation between the current arm position and the desired one. To compute the absolute locations of the interpolation nodes the current arm position must be obtained by the ECP from lower control level. As the feedback is required only once per each trajectory section, this is the case of sporadic feedback. More frequent feedback is also possible. Effector state e_j can be absent from the argument list of $f^{*k}_{e_{j_{ECP}}}$. For instance, motion generation relative to the current arm location does not require ECP level feedback (motion by an offset).

A similar situation arises in the case of virtual sensor readings v. If they are present in the argument list of $f^{*k}_{e_{j_{ECP}}}$, the motion is generated on the basis of information contained in the sensor readings or this data is used to modify a predefined trajectory stored in c_j. If v is missing from the argument list sensorless motion generation takes place.

The presented classification of motion generators could be attained intuitively, but then there would be no way of showing that it is complete. By using the arguments of functions f^*_e as a classification criterion, completeness is guaranteed (all possible combinations of arguments are taken into account). Completeness is important from the point of view of implementation of multi-robot control software. This software must be designed in such a way that none of the above mentioned categories of motion generators is neglected. It also shows what are the capabilities of the system as a whole.

A classification based on the argument lists of functions f_I and f_T, both for the MP level and ECP level, can be conducted in a similar way. For lack of space, obviousness of the result and in the face of the above considerations, this shall be omitted.

7 C++ IMPLEMENTATION OF MP

The implementation of MRROC++, which is based on the above theoretical considerations, is fairly straightforward. Images of effectors (robots in this case), images of virtual sensors, motion generators were implemented using object oriented paradigm and C++ language as an implementation platform. The presented code for the MP level has been stripped to bare-bone essentials for the sake of brevity.

```
class robot {
// Effector base class (robot image)
// Data members and error handling are omitted

 virtual void execute_motion (void) = 0;
   // Robot motion command (realised by a descendant class)
   // On the MP level this is a command for ECP
}; // end: class robot



class sensor {
// Virtual sensor base class (sensor image)
// Data members and error handling are omitted

 virtual void initiate_reading (void) = 0;
   // Orders data from VSP (realised by a descendant class)

 virtual void get_reading (void) = 0;
   // Gets data from VSP (realised by a descendant class)
}; // end: class sensor



class generator {
// Motion generator base class
// Data members and error handling are omitted

 virtual BOOLEAN first_step ( list<sensor>* sensor_list,
                              list<robot>* robot_list ) = 0;
    // Generates the first motion step.
    // It usually differs from each next step.

 virtual BOOLEAN next_step ( list<sensor>* sensor_list,
                             list<robot>* robot_list) = 0;
    // Generates next motion step and evaluates
    // and tests the terminal condition
}; // end: class generator
```

Lists of `sensor`s and `robot`s are necessary, so list templates were created. `E_ptr` points to an object (`robot` or `sensor`) being the element of the list. The robot and sensor images can be treated as data structures containing the current state of a respective device and in the case of robots also its next (demanded) state.

The simplified code of `Wait` and `Move` instructions is presented below. The semantics of those instructions is specified both formally by equations (4) and (6) respectively and less formally by figures 3 and 4. Each loop executes one step, i.e. takes either $1ms$ or $2ms$ depending on the implementation.

Obviously descendant objects, specific to the ECP and EDP communication format, are used at run-time. Detected errors are dealt with by exception handling. The exceptions are caught and handled at the topmost level of the process (`main` function).

```
void Wait ( list<robot>* robot_list, list<sensor>* sensor_list,
            condition&  the_condition ) {
```

```
 list<sensor>* sensor_lptr; // sensor pointer
 list<robot>*  robot_lptr;  // robot  pointer

 do { // waiting
   // Order data from relevant virtual sensors
   for ( sensor_lptr = sensor_list;  sensor_lptr;
         sensor_lptr = sensor_lptr->next )
     sensor_lptr->E_ptr->initiate_reading();

   // Get data from relevant virtual sensors
   for ( sensor_lptr = sensor_list;  sensor_lptr;
         sensor_lptr = sensor_lptr->next )
     sensor_lptr->E_ptr->get_reading();

 // Test the initial condition
 } while ( !the_condition.condition_value ( robot_list, sensor_list ) );
}; // end: Wait()




void Move ( list<robot>*  robot_list, list<sensor>* sensor_list,
            generator& the_generator ) {
 list<sensor>*  sensor_lptr; // sensor pointer
 list<robot>*   robot_lptr;  // robot  pointer

 // Generate first motion step
 if (!the_generator.first_step (sensor_list, robot_list)) return;

 do { // Realize the motion
    // Order data from relevant virtual sensors
    for ( sensor_lptr = sensor_list;  sensor_lptr;
          sensor_lptr = sensor_lptr->next )
     sensor_lptr->E_ptr->initiate_reading();

    // All robots execute their motion step
    for ( robot_lptr = robot_list;  robot_lptr;
          robot_lptr = robot_lptr->next )
      robot_lptr->E_ptr->execute_motion();

    // Get data from relevant virtual sensors
    for ( sensor_lptr = sensor_list;  sensor_lptr;
          sensor_lptr = sensor_lptr->next )
      sensor_lptr->E_ptr->get_reading();

 // Test the terminal condition and compute the next step for all robots
 } while ( the_generator.next_step ( sensor_list, robot_list ) );
}; // end: Move
```

The controller program executing a task is constructed by using the `Move`, `Wait` and `C++` instructions. Usually only control flow instructions (`if then else`, `while`, `for`, `switch`) and mathematical expression processing capability of C++ are utilised. The control flow instructions can use the values stored in sensor and robot images as their arguments to supplement the decision making capability of MRROC++. The paths are generated by the motion generators that the user has to supply by coding them in C++ in conformance to a predefined template. There are plenty of standard ones in the system, so usually either the existing ones are used directly or modified. The software is modular so more or less a fill in the blanks procedure is followed in creating the system. First virtual sensors are produced. The `Move` and `Wait` instructions "know" how to get the data from a virtual sensor to its image (this is possible because of encapsulation and inheritance). Then the generators are produced (C++ code operating on images). Finally, the code invoking the necessary `Move` and `Wait` instructions is produced. The images contain data fields containing both the current state of the particular robot and the next state it has to attain (in one step). All the data transfers between processes are hidden away from the programmer – they are handled by the `Move` and `Wait` instructions. For instance, within the `Move` procedure `initiate_reading` and `get_reading` methods are invoked. Each one of them executes message passing operation (`Send-Receive-Reply`) between the MP and and the VSP. The reply message contains the current reading of the virtual sensor and that is used to update the sensor image. Similarly the `execute_motion` method also uses message passing to convey the results of processing performed by the MP level generator to the ECP. In this way the processes can reside on different nodes of the computer network.

8 SIMPLE EXAMPLE

To present the principles of programming using MRROC++ let us make the following assumptions keeping the example within the bounds of a journal article.

- A trivial task of transferring a rigid body by two manipulators: RNT and Polycrank, will be realised.
- The two robot system has been ideally calibrated, so no sensors are necessary.
- The grasping locations on both sides of the rigid body are known accurately.
- The robots are equipped with on/off grippers and so are controlled in conjunction with the arm they are mounted on.
- The EDPs for both robots already exist.

In this case the `main` function of the MP process is composed of constant portions of code devoted to the initiation of communication with other processes and error handling, which are not presented here, as the programmer neither has to write that nor is supposed to change anything there. The programmer inserts into an appropriate slot the following instructions.

```
// Declare the robots that will be used (create their images)
rnt_robot rnt("ecp_rnt");    // "ecp_rnt" is the name of the ECP
                             // executable file for the RNT robot
poly_robot poly("ecp_poly"); // "ecp_poly" is the name of the ECP
                             // executable file for the Polycrank robot

// Create the robot list
list<robot> rl(&rnt);
rl.insert_list_element(&poly);

// Create an empty sensor list, as no sensors are used
list<sensor> sl = NULL;

// Declare the generators
separate_motion_generator generator1;
common_motion_generator   generator2;

// Move the two robots to the grasping position and close their grippers
Move(rl, sl, generator1);

// Move the grasped object
Move(rl, sl, generator2);
```

Besides writing the above trivial sequence of statements the code of the generators has to be provided. To do that two member functions: `first_step` and `next_step` have to be coded for each of the generators. In the case of `separate_motion_generator`, which causes independent but simultaneous motions of the robots to their respective grasping positions the `first_step` function initially and the `next_step` function cyclically have only to insert into robot images a constant command that nudges the ECP level generators to calculate the next motion step and execute it. The cyclic intervention is necessary, because only the MP has direct access to the user interface, through which the operator might want to abort or suspend the current motion. Moreover in this way the current state of the robots can be updated in the image and information about any errors can be conveyed to the user. The ECP generators are responsible for contacting the MP and the calculation of the next trajectory point along the trajectory to the destination, i.e. the grasping location. With the execution of the last trajectory segment the ECP generators insert into the ECP level images of robots a command ordering them to close the grippers. The motion commands are extracted from the robot images by the `execute_motion` member function. It delivers this command to the EDP, and that executes the adequate motion step.

In the case of the `common_motion_generator` the situation is slightly more complex, as its `first_step` and `next_step` member functions have not only to nudge the ECP level generators, but also to calculate the locations of points or frames along the trajectories of motion for both robots. Knowing the grasping positions, which are the current locations of both end-effectors (this information is available from the MP robot images) and both the initial and the destination location of a certain coordinate frame affixed to the rigid body that is to be transferred, a relative transformation between that frame and the grasping locations can be calculated using homogeneous matrices. That would be done by the `first_step` function and subsequently used by the `next_step` function. To facilitate the calculations homogeneous matrix manipulation functions are provided within `MRROC++`. Now the motion of the frame from the initial location to the destination one can be transformed into a series of incremental displacements of both robot arms. The MP level generator is capable of passing the information about the displacement that each of the robots has to execute during the next time step to the ECPs by inserting it into the robot images (i.e. `rnt` and `poly`). This information is later transferred to the ECPs by invoking the `execute_motion` function within the `Move` instruction. The ECP generators have only to transmit those displacements to the EDPs for execution and obtain an updated robot state in return. The number and the distribution of the interpolation nodes along the trajectory, and so the number of steps and the velocity profile of trajectory following, results from the interpolation algorithm used by the programmer. Usually the trajectory is a straight line segment with trapezoidal or triangular velocity profile. As it is assumed that the two robot system is adequately calibrated and the generated motion increments are small the stress in the rigid body and the arms is negligible. In reality it would be advisable to use a force sensor to monitor that stress. In that case the generator would also involve sensor images in the calculation of the next effector state.

On the ECP level again two motion instructions are used, so the code of the ECP for the RNT robot would be:

```
// Create the robot image
rnt_robot rnt("edp_rnt");     // "edp_rnt" is the name of the
                              // EDP executable file for the RNT robot

// Create an empty sensor list, as no sensors are used
list<sensor> sl = NULL;

// Declare the generators
rnt_separate_generator generator1;
rnt_common_generator   generator2;
```

```
// Move the robot to the grasping position and close their gripper
Move(rnt, sl, generator1);

// Move the grasped object
Move(rnt, sl, generator2);
```

The code of the ECP for the other robot is similar, only `rnt` must be changed to `poly`. The `rnt_separate_generator` member functions `first_step` and `next_step` accept the nudges from the MP and calculate the next location along the trajectory to the grasping position. Once the calculated position is available it is inserted into the robot image `rnt`. The `execute_motion` function then transmits it to the EDP, and that process forces the arm to move accordingly. In the case of `rnt_common_generator` the next position on the trajectory is obtained from the MP, and that is inserted into the robot image. From there it is transferred to the EDP in the same way as above.

The images contain data members for all possible methods of expressing the current and the desired location of the end-effector (e.g. joint space, Cartesian space plus Euler angles, homogeneous transforms). The programmer in the code of the generator uses only one of those representations and is responsible for updating others, if a switch of coordinates is necessary. Only the joint coordinates are robot type dependent. If those are not used the same generator can be used for different robots. The MP and the ECP images of robots with the same number of degrees of freedom are the same, only the interpretation of the joint coordinates is different. The EDP is responsible for the recalculation of any end-effector coordinate representation into motor shaft positions and vice versa. Due to this descendant classes of `robot` are more or less hardware independent.

The real C language code of the generators is not presented as it operates on data members of the robot images using special member functions for that purpose, so both would need explanation. Moreover, although the coordinate transformations are straight forward the code of four generators would take some space, so the discussion has been limited only to the general descriptions of actions that the generators perform and the code has been left out. For the ECP generators that only transmit the results of calculations of the MP generator the C code is 15 lines. For the MP generator performing the trajectory calculations it is below 100 lines. Currently the system has about a dozen parametric generators among others expressing trajectories along straight lines in different end-effector coordinate spaces and having diverse velocity profiles. Usually those are used as templates for modification, if something new is necessary.

There is a considerable benefit of separate coding of the program and each motion. First the programmer creates the general structure of the program by specifying all the necessary motions without going into trajectory details. A crude shell of the program results. If testing of the overall program structure is necessary any readily available generators can be used. Once that is complete each motion can be fine tuned by supplying an adequate generator. In extreme cases the generators can define besides the kinematic properties of the trajectory also the dynamic behaviour of the robot by changing the parameters of the joint regulators [24]. This two phase construction of programs facilitates the well established method of programming by stepwise refinement.

9 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 programming languages/libraries submerged in general purpose programming languages are used by the research community than specialised RPLs. `MRROC++` is submerged in `C++` running under real-time operating system QNX [19] capable of supervising a computer network. Initially `MRROC` [20] was implemented using procedural approach, but currently this has been changed to object-oriented approach [21], 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. Using `C++` instead of `C` functions further simplifies programming, as the former have access to all the data members of objects, whilst functions either have to rely on global variables or long parameter lists. 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. 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 ASEA type IRb-6 robots (one of them mounted on a track), prototype serial-parallel structure RNT robot [22], and a prototype fast robot without joint limits – Polycrank [23]. All of those robots require specialised hardware controllers [24]. 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 – both industrial and research.

MRROC++ has been successfully used to build a typical industrial controller for a task consisting in engraving inscriptions in soft materials (e.g. wood) by a robot equipped with a milling machine [25]. The controller inputs data files produced by a CAD system – describing the Cartesian paths along which the engraving has to take place. The path generator uses moving segment B-spline interpolation between points in the same way that the CAD station produces on-screen drawings of tool trajectories. Later it reproduces these paths with high precision due to high rigidity of the serial-parallel structure prototype RNT robot [22]. This is a continuous path industrial application, which most of the industrial robots would have difficulty performing, as in this case the executed trajectories, unless taught-in, would have to be interpolated either by straight lines or circular arcs. In the case of MRROC++ based control system the trajectories can be programmed to have any shape and velocity profile along them. In this case the shape was defined to be a series of B-spline curves spanning eight point segments. From each such segment only the curve between the first two points is utilised, and the remaining portion is discarded. Then the segment is shifted to the next eight point part of the trajectory starting with the second point of the first segment and ending with the next point after the last point of the first segment. In such a way a very smooth curve is built, and that is executed during milling.

Cooperative transfer of a rigid body by two robots having 5 d.o.f. each has been demonstrated by using MRROC [26]. It shows how the motion of over-constrained systems can be programmed using the presented software. To automate the tedious process of calibrating the two-robot system another controller was built. For calibration two high precision electronic theodolites were used [27]. The same procedure and software was later used in the case of the RNT robot [28].

MRROC based software was also used to build a system containing a robot and an ultrasonic matrix overhanging a conveyor. The 3D image obtained through that matrix enabled the detection, localisation and recognition of objects moving on a conveyor. For that purpose neural networks were incorporated into the controller [29, 30]. Thus obtained information was utilised in acquiring objects from a moving conveyor and sorting them by a robot. In a separate experiment a CCD camera was used for the same purpose.

The presented library/language can also be utilised for creating reactive controllers [31, 32, 33, 34], which have gained much attention lately, especially in the area of autonomous mobile robots. If robot arms are substituted (as effectors) by robot legs or wheels the same software can be used to build controllers for autonomous mobile systems. Originally the library/language was used to build a controller for a robot transferring a touch probe and later a force sensor inside a maze. The robot gradually gained information on its surroundings by reacting to collisions with the walls of the maze while trying to attain a global goal of finding a way out of the maze. Another controller was built which used global information about the maze layout obtained through a CCD camera, although in this case a reactive controller was unnecessary and a distance-optimal path could be traced. Reactive control was also used to acquire moving objects from a conveyor. In this case infra-red sensors were the source of information both about velocity and position of the object. An interesting aspect of this research was that the same formalism that has been used in this paper can be extended to describe reactive robot systems and that hierarchic distributed controllers can be used as a platform to implement reactive control.

Although any robot system can be programmed using `C` or `C++` and a real time operating system directly, using a language extension in the form of a library and a controller architecture definition, as provided by `MRROC++`, significantly simplifies this task. This is mainly due to the architecture providing the fill–in–the–blancs structure into which ready made blocks, i.e. library modules, are inserted. Even if a specialised block is necessary, that is not present in the library, the existing one can be modified accordingly, what is a simpler task than writing the code from scratch. The flexibility and the generality of the presented approach enables coding of more diverse tasks than by using many of the other existing RPLs.

Acknowledgments

This work has been conducted in Warsaw University of Technology. The author gratefully acknowledges the participation of Dr W. Szynkiewicz in the implementation of `MRROC` and `MRROC++`.

References

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

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

[3] *Industrial automation systems – Manufacturing Message Specification – Part 3: Companion Standard for Robotics*. ISO/IEC 9505–3, 1991.

[4] Matsumoto A., Arai T., Mohri S., Ishii H., Sato K.: *Activities for the Standardization of Robot Languages in Japan*. Proc. 20th Int. Symp. on Industrial Robots (ISIR), Tokyo, Japan, 4–6 October 1989. pp.843–850.

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

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

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

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

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

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

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

[12] 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.

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

[14] 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. pp.547–558.

[15] Backes P., Hayati S., Hayward V., Tso K.: *The KALI Multi-Arm Robot Programming and Control Environment*. Proc. NASA Conf. on Space Telerobotics, 1989. pp.179-188.

[16] Paul R.: *Robot Manipulators: Mathematics, Programming and Control*. The MIT Press, MA, 1981.

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

[18] Lloyd J., Hayward V.: *Real-Time Trajectory Generation in Multi-RCCL.* Journal of Robotics Systems, 10 (3), 1993. pp.369–390.

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

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

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

[22] Nazarczuk K., Mianowski K., Olędzki A., Rzymkowski C.: *Experimental Investigation of the Robot Arm with Serial-Parallel Structure*. Proc. 9-th World Congress on the Theory of Machines and Mechanisms, Milan, Italy, 1995, pp. 2112-2116.

[23] Nazarczuk K., Mianowski K.: *Polycrank – Fast Robot Without Joint Limits*. Proc. of the 12-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'12, 6-9 June 1998. Springer-Verlag, Wien, pp.317-324.

[24] Zieliński C., Rydzewski A., Szynkiewicz W.: *Multi-Robot System Controllers*. Proc. of the 5th International Symposium on Methods and Models in Automation and Robotics MMAR'98, 25–29 August 1998, Międzyzdroje, Poland, Vol.3, pp.795–800.

[25] Mianowski K., Nazarczuk K., Wojtyra M., Szynkiewicz W., Zieliński C., Woźniak A.: *Application of the RNT Robot to Milling and Polishing*. Proc. of the 13-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'13, 3–6 July 2000, Zakopane, Poland.

[26] 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.

[27] Frączek J., Buśko Z.: *Calibration of Multi Robot System Without and Under Load Using Electronic Theodolites*. Proc. of the 1st Workshop on Robot Motion and Control RoMoCo'99, Kiekrz, Poland, 28–29 June 1999. pp. 71-75.

[28] Frączek J., Buśko Z.: *Calibration of Serial and Serial-Parallel Robot Systems Using Electronic Theodolites and Error Correction Procedure*. Proc. of the 10th World Congress on the Theory of Machines and Mechanisms, Oulu, Finland, 20–24 July 1999. Vol.3. pp.972–977.

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

[30] Brudka M., Pacut A.: *Intelligent Robot Control Using Ultrasonic Measurements*. Proc. of the 16-th IEEE Instrumentation and Measurement Technology Conference IMTC/99, Venice, Italy, vol. 2, May 1999. pp. 727–732.

[31] Zieliński C.: *Reaction Based Robot Control*. Mechatronics, Vol.4, no.8, 1994. pp.843–860

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

[33] Zieliński C.: *Sensorimotor robot control*. 7-th IFAC/IFORS/IMACS Symposium on Large Scale Systems: Theory and Applications, 10–13 July 1995, London, United Kingdom. Vol.2, pp.797–802.

[34] Zieliński C.: *Reactive Robot Control Applied to Acquiring Moving Objects*. Proc. of the 3rd International Symposium on Methods and Models in Automation and Robotics MMAR'96, 10–13 September 1996, Międzyzdroje, Poland. Vol.3, pp.893–898.

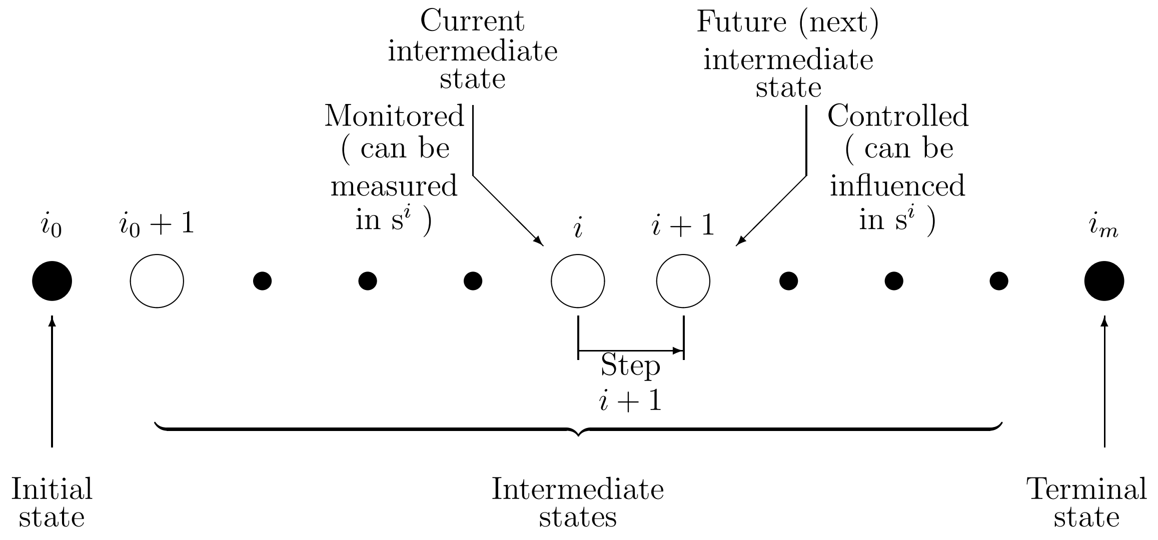

Current intermediate state
Future (next) intermediate state
Monitored (can be measured in s^i)
Controlled (can be influenced in s^i)
i_0
$i_0 + 1$
i
$i + 1$
i_m
Step $i + 1$
Initial state
Intermediate states
Terminal state

Figure 1: Evolution of the system state during execution of a motion instruction

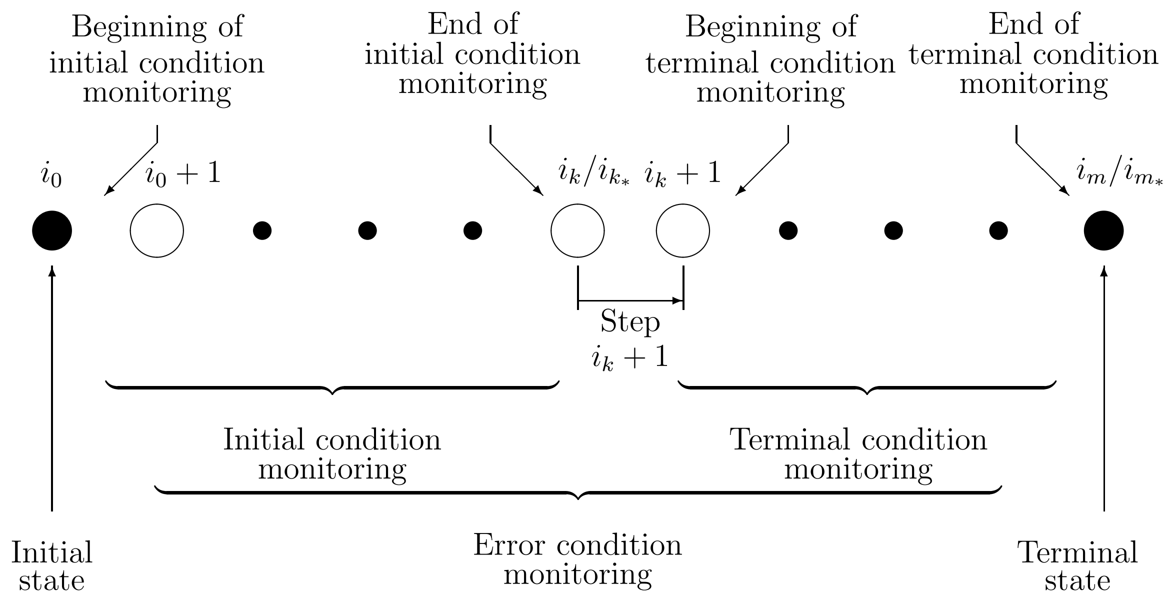

Beginning of initial condition monitoring
End of initial condition monitoring
Beginning of terminal condition monitoring
End of terminal condition monitoring
i_0
$i_0 + 1$
i_k/i_{k_*}
$i_k + 1$
i_m/i_{m_*}
Step $i_k + 1$
Initial condition monitoring
Terminal condition monitoring
Initial state
Error condition monitoring
Terminal state

Figure 2: Monitoring the execution of a motion instruction

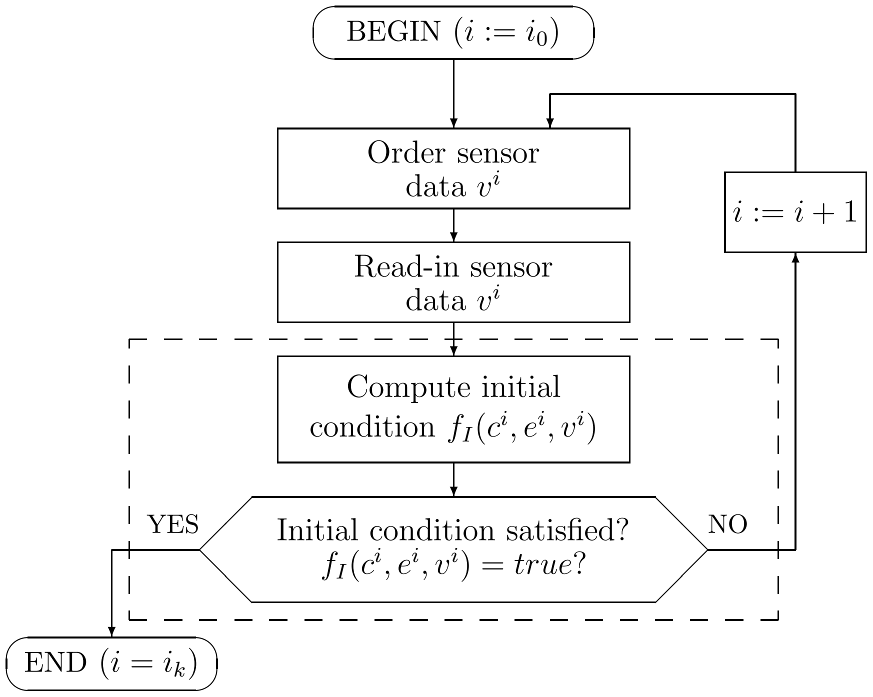

BEGIN $(i := i_0)$
Order sensor data v^i
Read-in sensor data v^i
Compute initial condition $f_I(c^i, e^i, v^i)$
Initial condition satisfied? $f_I(c^i, e^i, v^i) = true$?
YES
NO
$i := i + 1$
END $(i = i_k)$

Figure 3: Flow chart of the `Wait` instruction

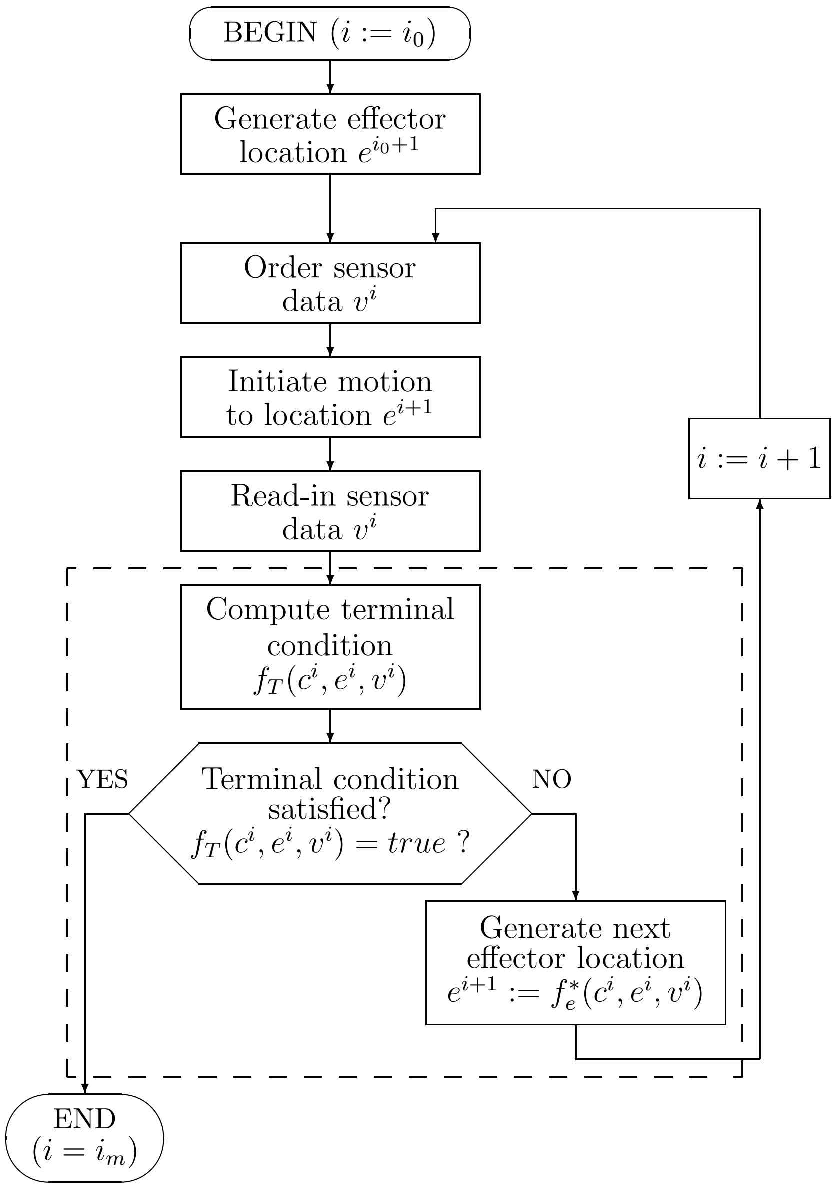

BEGIN $(i := i_0)$
Generate effector location e^{i_0+1}
Order sensor data v^i
Initiate motion to location e^{i+1}
Read-in sensor data v^i
Compute terminal condition $f_T(c^i, e^i, v^i)$
Terminal condition satisfied? $f_T(c^i, e^i, v^i) = true$?
YES
NO
Generate next effector location $e^{i+1} := f_e^*(c^i, e^i, v^i)$
$i := i + 1$
END $(i = i_m)$

Figure 4: Flow chart of the `Move` instruction

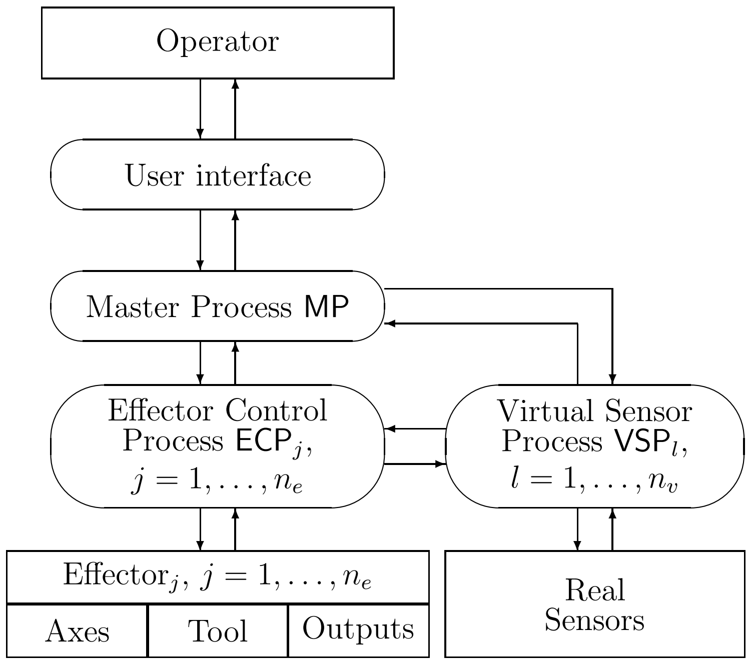
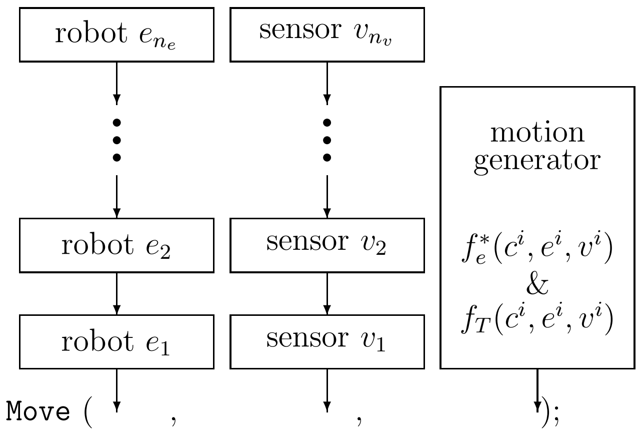

Operator
User interface
Master Process MP
Effector Control Process ECP_j, $j = 1, \ldots, n_e$
Virtual Sensor Process VSP_l, $l = 1, \ldots, n_v$
Effector$_j$, $j = 1, \ldots, n_e$
Axes
Tool
Outputs
Real Sensors

Figure 5: Hierarchical structure of a multi-robot controller



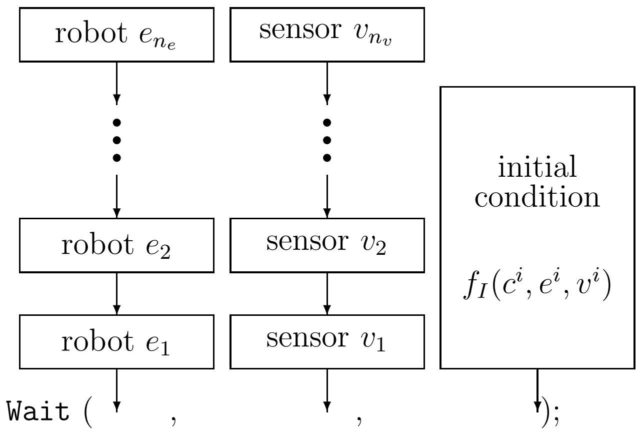


Figure 6: `MRROC++` motion instructions

Cezary Zieliński received M.Eng. degree in control in 1982, Ph.D. degree in control and robotics in 1988 and habilitation degree also in control and robotics in 1996, all from Warsaw University of Technology (WUT), Department of Electronics and Information Technology, Warsaw, Poland. He is a professor of WUT employed by the Institute of Control and Computation Engineering (ICCE). He spent 9 months in all as a visiting researcher at Mechanical Engineering Department of Loughborough University of Technology, Loughborough, UK, in 1990 and 1992, working on robot programming methods. Currently he is on sabbatical from WUT and a senior fellow at Nanyang Technological University, Singapore, involved in robotics research there. In WUT he headed the Robotics Group in ICCE and the interdepartmental Robotics Group of WUT working on the design of special purpose robot manipulators, and their controllers and programming methods. His research interests include: robot programming methods, multi-robot system controllers, robot kinematics, utilisation of sensors in robot control, behavioural control of robots, general purpose programming languages, mechatronics, design of digital circuits.