Second International Workshop on Robot Motion and Control, RoMoCo'01, 18–20 October 2001, Bukowy Dworek, Poland. pp.121–128

A Quasi-Formal Approach to Structuring Multi-Robot System Controllers

Cezary Zieliński

Abstract— **The paper presents a formal approach to structuring multi-robot system controllers. It was used to structure the `MRROC++` based controllers. The deduced structure is such that on the one hand it is not too complex for implementation and on the other hand it does not limit the tasks that can be executed. Both coordinated and uncoordinated motions are taken into account. Use of diverse sensors is also considered. This approach enables the classification of motion generators and execution conditions giving further insight into future performance of the designed system. The same approach can be used to elaborate different structures to the one used by `MRROC++` based systems.**

Keywords— **robot programming, multi-robot systems, multi-robot controllers, sensor incorporation**

I. INTRODUCTION

A considerable effort has been concentrated on developing new Robot Programming Languages (RPLs), both specially defined for robots (e.g. `WAVE` [23], `AL` [20], [28], `VAL II` [41], `AML` [26], `RAPT` [1], [28], `SRL` [7], [9] `TORBOL` [27], [28]) and general purpose programming languages enhanced by libraries of robot specific procedures (e.g. `RCCL` [13], `Multi-RCCL` [17], [18] `KALI` [14], [15], [5], `RORC` [31], [29], `MRROC` [31], [32], `PASRO` [8], [9], `MRROC++` [37], [39]). Specialised languages render the controller with a closed structure. If new hardware is to be added to the system, usually the language itself has to be modified too. Those changes have to be reflected in the 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. This approach is also gradually gaining recognition in the industrial community (e.g. `RobotScript` is an RPL submerged in `Visual Basic` [16]).

`MRROC++` is submerged in `C++`. It utilises the real-time operating system QNX [40] capable of supervising a computer network. Initially `MRROC` [32] was implemented using procedural approach, but currently this has been changed to object-oriented approach [36], 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 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. 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. The user's program is incorporated into the controller code. The user delivers the code for just a few object classes that are used by `Move` and `Wait` instructions. The formal approach presented in this paper not only facilitated the structuring of the whole system software, but also helped in distinguishing the few objects that the user has to modify while creating a user's program.

The approach followed in implementing `MRROC++` based controllers has been tested on diverse robots and tasks. `MRROC++` can currently control ASEA type IRb-6 robots (one of them mounted on a track), prototype serial-parallel structure RNT robot [6], [21], and a prototype fast robot without joint limits – Polycrank [22]. All of those robots require specialised hardware controllers [38]. Force/torque, ultrasonic, and infrared sensors, CCD cameras and a conveyor belt have been included in the implemented systems. 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 [19]. 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 [21]. 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 middle (4th and 5th) points is utilised, and the remaining portion is discarded. Then the segment is shifted by one point to the next eight point part of the trajectory starting with the second point of the first

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

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` [34]. 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 [11]. The same procedure and software was later used in the case of the RNT robot [12].

`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 [24], [25]. 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.

This software can also be applied to create reactive controllers [30], [31], [33], [35], which have gained much attention lately, especially in the area of autonomous mobile robots [2], [3], [4]. If robot arms are substituted by robot legs or wheels the same software can be used to build controllers for autonomous mobile systems. Originally a controller was built for a robot transferring inside a maze a touch probe and later a force sensor. 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 [35]. An interesting aspect of this research was that the same formalism that will be presented in this paper can be extended to describe reactive robot systems and that the hierarchic distributed controllers can be used as a platform to implement reactive control.

This paper gives formal reasons for the assumed structure of the `MRROC++` system. Formal reasoning distinguished the variable part of the software and the part that is fixed (not subjected to modifications). The variable part is composed of motion instructions (e.g. `Move`, `Wait`) and object classes (e.g. `robots`, `sensors`, `generators`, `conditions`). The user expresses the task in terms of motion instructions, and then delivers, in a plug-in fashion, the specific objects that deal with the details of its execution (e.g. trajectories). The formally postulated motion generators can cope with any type of independent or cooperative motions with or without sensors. The division of software into the fixed and variable part lets the user concentrate only on the task that has to be executed. The remaining portion is not modified and so does not distract the user's attention from coding the task.

II. ROBOT SYSTEM DECOMPOSITION

A robot system S is composed of three subsystems:

$$S =< C; E; R >, \tag{1}$$

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 of the system S is denoted as:

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

where:

c	–	state of the control subsystem C,
e	–	state of the effectors E,
r	–	state of the real (hardware) sensors R.

The motion instructions of the user's program, in the most general case, take into account the the current state c of the control subsystem C and the current readings r of the receptors R, and exert influence over effectors E by changing (enforcing) the desired state e. The state of each component of the system can be expressed in terms of diverse abstract notions. In the case of effectors these can be: manipulator joints, end-effector or objects of the work-space. Data obtained from real sensors usually cannot be used directly in robot motion control. For instance, to control the arm motion, only the grasping location of the object that is to be picked would be necessary. In the case of such a complex sensor as a camera a bit-map has to be processed to obtain the grasping location. In some other cases a simple sensor in its own right would not suffice to control the motion (e.g. a single touch sensor), 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:

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

III. SENSOR UTILIZATION

The execution of a motion instruction begins in an **initial state**, ends in a **terminal state**, and causes the system to traverse 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. Usually the duration of each step is either equivalent to the servo sampling rate (e.g. $1ms$) or a low integer multiple of that. The initial state and the terminal state can be treated as boundary cases of intermediate states.

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**. Three reasonable purposes of intermediate state monitoring can be distinguished:

- initial condition monitoring (waiting for the satisfaction of a certain condition, so that the instruction execution can proceed),
- terminal condition monitoring (execution of an instruction – usually an instruction causing a motion – until a certain condition is fulfilled),
- error condition monitoring (detection of abnormal states).

Besides monitoring, i.e. passively observing the state evolution, we want instructions to have an active influence on the future states. That is called control of future intermediate states.

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. As the effectors are immobile the following semantics results:

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

$f_I(c^i, e^i, v^i)$	–	initial condition (Boolean value function),
$f_E(c^i, e^i, r^i)$	–	error condition (Boolean value function),
i_m	–	step in which $f_I(c^i, e^i, v^i)$ becomes *true*, and so the initial 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).

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 – e^{i_m}) and abnormal termination (i.e. when error condition is detected – $e^{i_{m*}}$). The reason for this is to explicitly label the terminal effector state and to distinguish between normal and abnormal instruction execution termination.

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.

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

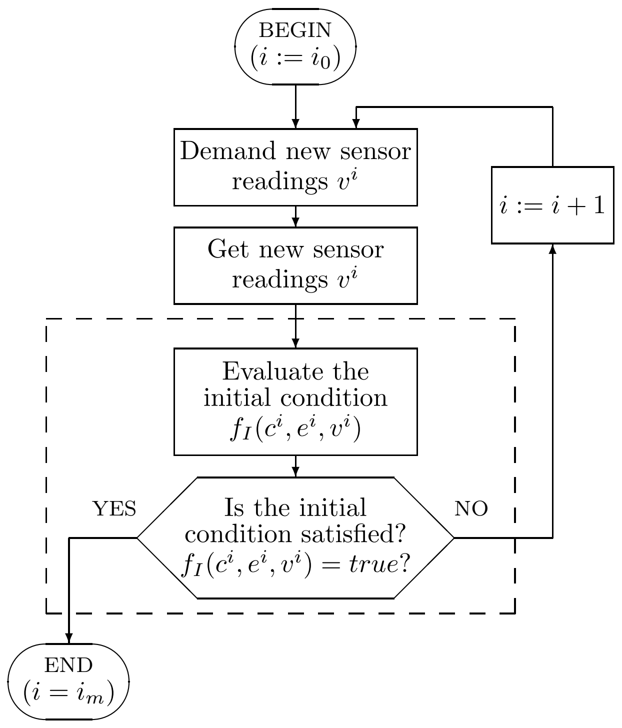
$$\begin{cases} e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{when} \\ & f_T(c^i, e^i, v^i) = false \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_m} & \text{when} \\ & f_T(c^i, e^i, v^i) = true \bigwedge \\ & f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_{m*}} & \text{when} \\ & f_E(c^i, e^i, r^i) = true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

where:

$f_e^*(c^i, e^i, v^i)$	–	effector transfer function
$f_T(c^i, e^i, v^i)$	–	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.

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.

Semantics specified by (4) can be represented graphically by the flow chart in (Fig. 1). Its implementation results in a `Wait` instruction. Definition of semantics (5) can be transformed into the flow chart in (Fig. 2). This can be used as an implementation vehicle for a `Move` instruction. It should be noted that error condition monitoring is not included in the flow charts (this can be 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.



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

IV. MRROC++

The primary task of a system designer is to come up with the structure of the system, i.e. its components and the interconnections between those components. The necessary interconnections between the system components can be deduced from the general forms of effector and control subsystem transfer functions. The semantics (4) of the initial condition monitoring instruction (`Wait`) shows that the effector transfer function in that case is an identity, as the effectors do not change their state throughout the execution of this instruction, but in the case of the instruction controlling future intermediate states (`Move`), the general semantics (5) implies an effector transfer function with a value depending on all three system components.

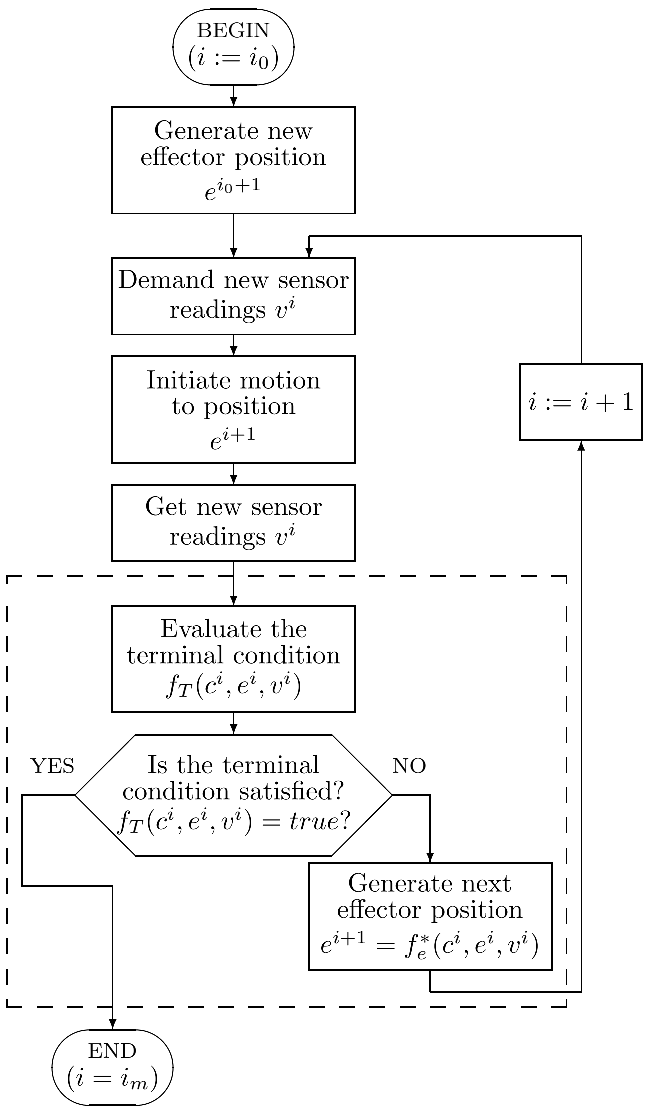
$$e^{i+1} = f_e^*(c^i, e^i, v^i) \tag{6}$$

Here it is assumed that the list of arguments of the transfer function f_e^* is the same as the lists of arguments of condition monitoring functions f_I and f_T, so we can concentrate our considerations only on the transfer function f_e^*.

Any computations have to be done within the control subsystem, which due to that changes its internal state. Those computations, in the most general situation, involve the information from all the system components.

$$c^{i+1} = f_c(c^i, e^i, v^i) \tag{7}$$

The transfer function in the form (7) implies that there exists a single centralised control subsystem that changes its state basing on the information obtained from all the



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

Fig. 2
`Move` INSTRUCTION FLOW CHART

effectors and all virtual sensors. Rarely in complex systems the situation is such that the state of each component of the system depends on every other component. We need a finer granularity of system decomposition, so the system S, as described by (1), is further divided by taking into account that there are n_e effectors $E_j, j = 1, \ldots, n_e$. For this situation still a centralised control subsystem can calculate the next effector state for all of the effectors, 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 effector contollers C_j each responsible for control of the effector E_j, $j = 1, \ldots, n_e$.

Usually virtual sensors are grouped into bundles (`MRROC++` does not impose this constraint, but experience does). Each bundle is associated with a separate effector. The actions of effector E_k rely on n_{v_k} virtual sensors.

Moreover the coordinator reads its own n_{v_0} virtual sensors. In consequence the system has n_v virtual sensors:

$$n_v = \sum_{k=0}^{n_e} n_{v_k} \tag{8}$$

As the result (1) can be transformed into:

$$S = < C_0, C_1, \ldots, C_{n_e}; E_1, \ldots, E_{n_e}; V_0, \ldots, V_{n_e} > \tag{9}$$

where each V_k, $k = 0, \ldots, n_e$ represents a bundle of virtual sensors. It is composed of virtual sensors V_{k_l}, $l = 1, \ldots, n_{v_k}$. The reading of each of those sensors is represented by v_{k_l}.

As the system has been subdivided into even smaller subsystems than in (1), new transfer functions for those subsystems have to be specified. The designer by deciding what should be the arguments of those functions determines what connections will be present between those subsystems. 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 the coordinator C_0 was chosen to evolve in the following way:

$$c_0^{i+1} = f_{c_0}(c_0^i, c_1^i, \ldots c_{n_e}^i, e_{c_1}^i, \ldots, e_{c_{n_e}}^i, v_{0_1}^i, \ldots, v_{0_{n_{v_0}}}^i) \tag{10}$$

where $e_{c_j}^i$, $j = 1, \ldots, n_e$ symbolises that C_0 obtains the information about the state of each of the effectors E_j indirectly through C_j. $e_{c_j}^i$ is the image of the state of the effector. This implies that no direct connection between C_0 and each E_j needs to be implemented. The connection between C_0 and all C_j suffices.

The state of each effector E_j and of each control subsystem part C_j was chosen to to evolve in the following way:

$$e_j^{i+1} = f_{e_j}^*(c_0^i, c_j^i, e_j^i, v_{j_1}^i, \ldots, v_{j_{n_{v_j}}}^i) \tag{11}$$

$$c_j^{i+1} = f_{c_j}(c_0^i, c_j^i, e_j^i, v_{j_1}^i, \ldots, v_{j_{n_{v_j}}}^i) \tag{12}$$

The general structure the Multi-Robot Research-Oriented Controller `MRROC++` (Fig. 3) 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 indirect 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 $v_k, k = 1, \ldots, n_v$ is implemented as a **Virtual Sensor Processes** (VSP$_k$) running concurrently to the other VSPs and ECPs. In consequence of (3)

$$v_{j_k}^i = f_{v_{j_k}}(c_j^i, e_{c_j}^i, r_{j_k}^i) \tag{13}$$

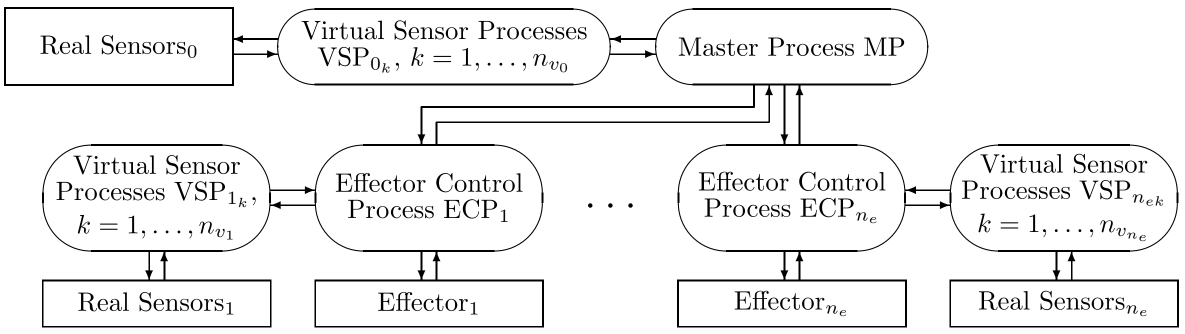
is obtained, where $j = 0, \ldots, n_e$, $k = 1, \ldots, n_{v_j}$, e_{c_j} is the image of the state of the *j-th* effector (the one associated with v_{j_k}) and $r_{j_k}^i$ is a grup of real sensors that delivers data to the virtual sensor.

As the semantics of the `Wait` and `Move` instructions formulated by (4) and (5) 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 (10) it is evident that only the coordinator (C_0, MP) perceives all of the effectors, although it does that indirectly through all C_j. From (11) and (12) 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} based on the information contained in those images. Then it sends the results of those computations to the ECP processes. 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 taking into account the the information contained in the images incorporated into those processes. The results of those computations are sent to the E_j for execution. The above mentioned 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 from the bundle associated with that process has to be created in it.

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.

V. MOTION INSTRUCTIONS

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 func-



Real Sensors$_0$
Virtual Sensor Processes VSP$_{0_k}$, $k = 1, \ldots, n_{v_0}$
Master Process MP
Virtual Sensor Processes VSP$_{1_k}$, $k = 1, \ldots, n_{v_1}$
Effector Control Process ECP$_1$
. . .
Effector Control Process ECP$_{n_e}$
Virtual Sensor Processes VSP$_{n_{ek}}$ $k = 1, \ldots, n_{v_{n_e}}$
Real Sensors$_1$
Effector$_1$
Effector$_{n_e}$
Real Sensors$_{n_e}$

Fig. 3
Structure of a MRROC++ based system

tions 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 charts of the `Move` (Fig. 2) and `Wait` (Fig. 1) 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}. The portions of the flow charts outside the bounds of the dashed lines are fixed – they do not change regardless of the task that has to be executed or the number and type of the robots or sensors used. This suggests that the programmer will have to deliver only the code dealing with: f_{I_0}, $f_{e_0}^*$ and f_{T_0}. To do that just two program entities (e.g. objects) are needed:

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

so object classes named `condition` and `generator` respectively have been introduced. The `Move` and `Wait` instructions (procedures) use within their bodies: `robot` (effector image), `sensor` (virtual sensor image), `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 programmer creates the MP by supplying adequate robot and sensor lists (images of robots and sensors), as well as initial conditions and motion generators. On the MP level the sensor list is composed of any of the n_{v_0} virtual sensor images. 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` image as one of their arguments. The sensor list for each effector E_k is created out of any of the n_{v_k} images of virtual sensors assigned to that effector.

VI. Motion Generators

An interesting point is that motion generators can be classified on the basis of the arguments of the function $f_{e_j}^*$, $j = 0, \ldots, n_e$. A separate classification is needed for the coordinator (MP) motion generators ($f_{e_{0_{MP}}}^*$) and a separate for all ECP motion generators ($f_{e_{j_{ECP}}}^*$). To distinguish them explicitly the subscripts MP and ECP have been added. From (11) it is obvious that the state of each effector depends on c_0, and that is determined by C_0 (MP) with the help of (10). That means that on the MP level the most general form of effector transfer function will depend on components c_0, e_c, v_0, where $e_c =< e_{c_1}, \ldots, e_{c_{n_e}} >$ and $v_0 =< v_{0_1}, \ldots, v_{0_{n_{v_0}}} >$. As a result it can be deduced that there are eight possible forms of that function, because each of the three arguments: c_0, e_c, v_0 can either be present or absent. Only the following five cases are meaningful: $f_{e_{0_{MP}}}^{*0}(c_0)$, $f_{e_{0_{MP}}}^{*1}(c_0, v_0)$, $f_{e_{0_{MP}}}^{*2}(c_0, e_c)$, $f_{e_{0_{MP}}}^{*3}(c_0, e_c, v_0)$, $f_{e_{0_{MP}}}^{**}() = const.$ In the first four cases c_0 is used to compute the next (demanded) effector state e^{i+1}. It must be present as an argument, because the images of effectors and sensors and other variables are contained within C_0, so e_c or v_0 cannot be used without c_0 – there would be no access to them. C_0 can also be used for memorizing other facts, which can be useful for motion generation (e.g. taught-in trajectories or functionally defined trajectories). In the fifth case c_0 is missing, so the resources of C_0 cannot be used. 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 $f_{e_{0_{MP}}}^*$, 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_c and v_0. With v_0 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_c is present in the argument list the motion generation takes into account high level effector state feedback. Substitution of e_c by $c_j, j = 1, \ldots, n_e$ broadens the capabilities of motion generation, as not only the high level effector feedback can be taken into account, but also any other information that is present on the ECP level.

Similar considerations are valid for the ECP level. Formula (11) shows that function $f^*_{e_{j_{ECP}}}(c^i_0, c^i_j, e^i_j, v^i_j)$ has four arguments, but c_j must always be present, because unlike on the MP level the resources of C_j must be employed in the determination of the next effector state – they cannot be delegated to a lower control level. This produces eight possible cases $f^{*q}_{e_{j_{ECP}}}, q = 0, \ldots, 7$ – all of them valid. When c_0 is not present in the argument list the effectors are not coordinated – 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 $f^{*q}_{e_{j_{ECP}}}$. If c_0 is present, then the effectors are coordinated by MP. Two forms of coordination are possible:

- loose – where the coordinator synchronised in time and space the effectors sporadically,
- tight – where the effectors are synchronised in each motion step or a low integer multiple of that (e.g. jointly transfer a rigid object).

In the case of loose coordination the MP (coordinator) transmits only decision information (an item from a finite set), and in the case of tight coordination numeric information (describing the next location to be attained) is being sent to the ECPs. It should be noted that even if the effectors are not coordinated by the coordinator they still can cooperate. In that case an effector perceives the actions of the other effectors with the sensors from its bundle. For the purpose of this paper coordination and cooperation should be distinguished – because they are not the same.

When e_j is present in the argument list of $f^{*q}_{e_{j_{ECP}}}$, 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 $f^{*q}_{e_{j_{ECP}}}$. 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_j. If they are present in the argument list of $f^{*q}_{e_{j_{ECP}}}$, the motion is generated on the basis of information contained in the sensor readings or this data is used to modify a predefined trajectory present in c_j. If v_j is missing from the argument list sensorless motion generation takes place.

VII. Initial and terminal conditions

Using exactly the same approach as above, a classification based on the argument lists of functions f_I and f_T can be conducted. As the classification is the same for both functions it will be carried out only once, so subscript P will stand for either I or T. As effector and sensor images are contained in C, so an adequate argument c must always be present.

On the MP level four possible combinations of arguments of the condition functions result: $f^0_{P_{MP}}(c^i_0)$, $f^1_{P_{MP}}(c^i_0, v^i_0)$, $f^2_{P_{MP}}(c^i_0, e^i_c)$, $f^3_{P_{MP}}(c^i_0, e^i_c, v^i_0)$. The presence of c_0 ensures access to the coordinator C_0 program variables. Presence of e_c ensures the possibility of checking the state of one or more effectors. Existence of v_0 enables testing the readings of the V_0 bundle of virtual sensors.

There are four possible arguments to those functions on the ECP level, but c_j must always be present, so eight possible combinations of arguments of the condition functions result: $f^0_{P_{ECP}}(c^i_j)$, $f^1_{P_{ECP}}(c^i_j, v^i_j)$, $f^2_{P_{ECP}}(c^i_j, e^i_j)$, $f^3_{P_{ECP}}(c^i_j, e^i_j, v^i_j)$, $f^4_{P_{ECP}}(c^i_0, c^i_j)$, $f^5_{P_{ECP}}(c^i_0, c^i_j, v^i_j)$, $f^6_{P_{ECP}}(c^i_0, c^i_j, e^i_j)$, $f^7_{P_{ECP}}(c^i_0, c^i_j, e^i_j, v^i_j)$.

If c_0 is missing the condition is constructed only out of local knowledge contained in c_j and the state e_j of the effector and the readings v_j of the bundle of virtual sensors V_j assigned to the effector E_j. It should be noted however, that $f^2_{I_{ECP}}(c^i_j, e^i_j)$ either reduces to $f^0_{I_{ECP}}(c^i_j)$ or is useless for the purpose of waiting, as e^i does not change – see (4). Nevertheless it can still be used for decision making – checking if the attained state e_j is the required one. The same can be said for other functions $f_{I_{ECP}}$ containing e_j.

As in the `MRROC++` system the function f_T is used in conjunction with function f^*_e within a generator, both of them have exactly the same lists of arguments.

VIII. Conclusions

The informal approach to partitioning of a multi-effector system, as described by (9) and later the formal elaboration of the internal communication links based on (11), (10) and (12), shows to the designer explicitly what are the capabilities of the designed system. If different forms of those functions would have been assumed a system with another internal structure would result. The assumed functions prohibit direct communication between effectors. Such communication is possible only indirectly through the coordinator. The benefit is the overall simplification of system implementation, but it can be argued that it limits generality. This dilemma has to be solved by the designer of the system, but he or she should be aware of the consequences of the decision. The introduced formalism shows the consequences explicitly.

This paper concentrated on showing how the introduced formalism was used to arrive at the structure of the `MRROC++`, but the same methodology can be used to derive other structures. The formalism is implementation independent, so it is not limited by the real-time operating system, the language used for coding the software or the means of communication between the processes.

The presented classification of motion generators and condition functions 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^*, f_I and f_T 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 and conditions is neglected. It also shows what are the internal communication capabilities of the system as a whole.

References

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

[2] Arkin R.C.: *Behavior-Based Robotics*. MIT Press, Cambridge, Mass., 1998.

[3] Brooks R.A.: *Inteligence Without Representation*. Artificial Intelligence. No.47. 1991. pp. 139–159.

[4] Brooks R.A.: *Intelligence Without Reason*. MIT, Artificial Intelligence Laboratory, A.I. Memo No.1293, April 1991.

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

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

[7] Blume C., Jakob W.: *Design of the Structured Robot Language (SRL)*. in: *Advanced Software in Robotics*, Eds. Danthiene A., Géradin M., Elsevier, North Holand, 1984. pp.127–143.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] Lapham J.: *RobotScript: The introduction of a universal robot programming language*. Industrial Robot, Vol.26, No.1, 1999, pp.17–27.

[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] 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.

[20] Mujtaba S., Goldman R.: *AL Users' Manual*. Stanford Artificial Intelligence Lab., AI Memo 323, January 1979.

[21] 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.

[22] 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.

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

[24] 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

[25] 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.

[26] 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.

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

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

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

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

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

[32] 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, Midzyzdroje, Poland. pp.603-608.

[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., 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.

[35] 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.

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

[37] 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.

[38] 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.

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

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

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