Bulletin of the Polish Academy of Sciences – Technical Sciences. Vol.53, No.1, 2005. pp. 57–67

Formal approach to the design of robot programming frameworks: the behavioral control case

Cezary Zieliński

Institute of Control and Computation Engineering,
Faculty of Electronics and Information Technology,
Warsaw University of Technology,
ul. Nowowiejska 15/19, 00–665 Warsaw, Poland,
C.Zielinski@ia.pw.edu.pl

Abstract

Programming frameworks [14] are application generators with the following components: library of software modules (building blocks out of which the system is constructed), a method for designing new modules that can be appended to the above mentioned library, a pattern according to which ready modules can be assembled into a complete system jointly exerting control over it and realizing the task at hand. The presented transition function based formalism can be applied to specifying programming frameworks for robot controllers executing very diverse tasks. The paper deals with systems consisting of multiple embodied agents, influencing the environment through effectors, gathering information from the environment through sensors and communicating with other agents through communication channels. The presented code patterns pertain to behavioral agents. The formalism was instrumental in the design of `MRROC++` robot programming framework, which has been used for producing controllers of single and two manipulator systems performing diverse tasks. The formalism introduces rigor into the discussion of the structure of embodied agent controllers. It is used as the means for the specification of the functions of the components of the control system and the structure of the communication links between them. This structures the implementation of a programming framework, and that in turn makes the coding of specific controllers much easier, both from the point of view of dealing with the hardware configuration of the system and the specific task that has to be executed.

Keywords: *robot programming frameworks*

1 Introduction

The motivation for this research stemmed from the necessity of quick production of controllers for diverse robots executing significantly differing tasks. Controllers for the

following systems were designed using the methodology and tools described in this paper:

- Serial-parallel robot exhibiting high stiffness and having a large work-space [16, 21, 15, 42], thus well suited to milling and polishing tasks [17],
- Direct-drive robot without joint limits [20, 22], hence applicable to fast transfer of objects,
- Two IRp-6 robot system acting as a two-handed manipulation system [43, 29].

How fast a new controller can be produced strongly depends on the quantity of readily available software that can be reused from former projects and the extent to which this software can be modified. Programming frameworks are the current answer to this predicament.

Many behavioral [4] and deliberative [26] robot control methods have been developed. Besides investigating task execution by a single robot, multi-agent systems are being developed. Unfortunately only some of those control methods have a formal description. Moreover, those control methods that do have formal specifications use different formal means of description. A unified formal description of systems composed of many embodied agents using behavioral or deliberative control methods has been obtained [39]. This paper focuses on the description of behavioral controllers of individual agents, treating multi-agent systems as a composition of individual agents communicating among themselves either directly through communication channels or indirectly by stigmergy [6].

Formerly frameworks used to be called simply robot programming libraries or languages, but both of those terms are not adequate. A library does not imply an associated software pattern into which the modules should be inserted, and a language is usually associated with a specific grammar (e.g., syntactic structure). As general purpose languages, such as `Pascal` or `C` have been used as the development tools for those libraries, so no new language was being defined. Although initially specialized programming languages had been favored, they lost their appeal, when it turned out that in the robotics domain the variability of equipment causes an ever greater demand for extensibility of those languages. Any extensions force a modification of the compiler or the interpreter of the language rendering the alteration more costly. Moreover, soon it became obvious that the specialized robot programming languages have to provide nearly all the capabilities of a general purpose programming language. Under those circumstances it was more reasonable to use a general purpose programming language and a library of modules specific to robot control and to define a general pattern according to which they should be assembled. Thus, although specialized robot programming languages initially gained considerable popularity (e.g., `WAVE` [23], `AL` [19], `VAL II` [1], `AML` [30], `RAPT` [3], `SRL` [5], `TORBOL` [31]) robot programming frameworks have been especially favored by the research community (e.g., `RCCL` [13], `KALI` [12, 11], `PASRO` [5], `RORC` [35, 32], `MRROC` [35, 34], `MRROC++` [36, 37], G$^{\text{en}}$oM [10, 2], `DCA` [24], `TCA` [28], `TDL` [27], `Generis` [18]), due to the variability and diversity of research tasks. Depending on the base language (e.g., `C` or `C++`) procedural

or object oriented approach to programming is fostered by those frameworks. Component based approach is also being considered (e.g., `DCOM` or `CORBA` [25]), but in this case the overhead of communication between distributed objects usually is an obstacle to the implementation of the hard real-time portion of the software, thus those problems have to be solved within a component and for that either the procedural or object oriented frameworks have to be used or the component has to be hand-coded. However, component based software is a viable alternative for implementing systems composed of cooperating embodied agents needing coordination or for implementing soft real-time portion of the software within an agent. Currently efforts are being made to produce public domain generic robot control software (e.g., the OROCOS project [9]).

Expressing ideas in natural languages tends to be inexact and somewhat superficial. Introduction of a formalism, that uses mathematical symbols, imposes rigor and precision on the discussion. Expressing our thoughts formally renders a deeper understanding of the topic and often discloses, otherwise hidden, properties of the proposed methods of solving the problem at hand. In our case the problem is formulated as: how to describe in a general and exact fashion the diverse behaviors that are necessary for the robots to adequately operate in complex environments. Moreover, we want the proposed description to be easily transformable into an implementation of the proposed ideas in the form of the control software coded in one of the programming languages.

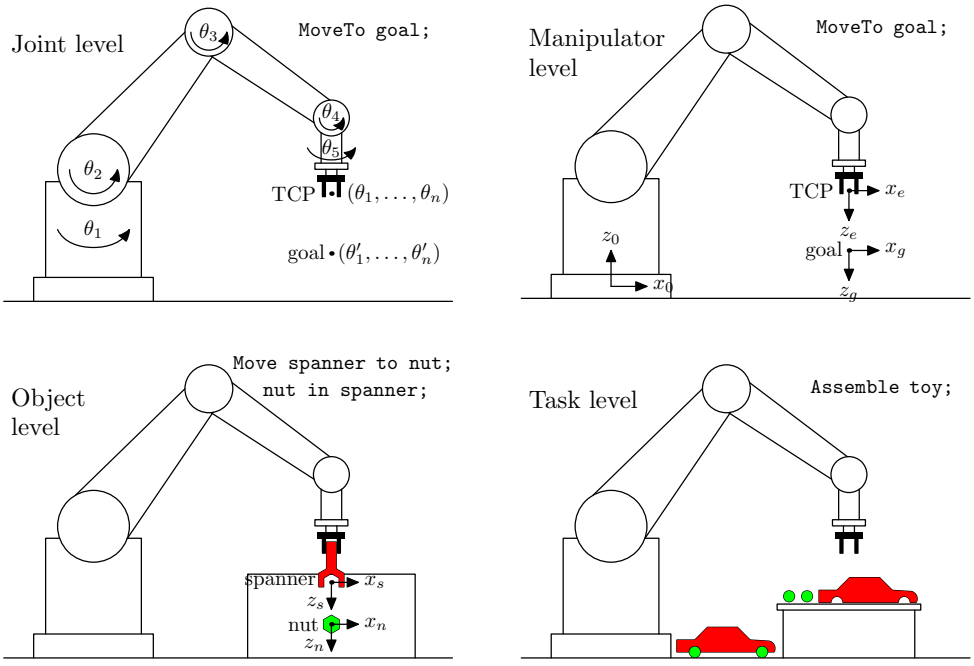

Figure 1: Robot ontology – levels of robot programming languages or frameworks

The notation presented in this paper enables a rigorous discussion and comparison of the diverse possibilities of defining robot (agent) behavior. Although this formalism is implementation independent, it suggests: the overall structure of the control system, its decomposition into modules, and enables the specification of the functions of those

modules. Control of a robot boils down to: processing the information about the current state of the robot and its environment to produce motions of the effectors, such that the task put forth before the system will be fulfilled. In a general case to do this effectively the robot's sensors have to be configured and controlled, exchange of information with other agents has to be organized, coordination of the agents in a multi-agent system has to be established, etc. All this necessitates the establishment of adequate information flow channels between the modules of the system and the definition of data processing taking place in those modules.

When designing a programming framework neither the hardware components of the system nor the task that it will have to execute are known. Nevertheless, the structure and the components of the system control software have to be designed in such a way that regardless of this lack of knowledge the implementation of such a system will be possible and fairly simple. It should be underscored that, if the functionality of the programming framework is not adequate, it might not be possible to implement some control systems and to execute some tasks. Introduction of formal notation enables the person implementing a programming framework to rigorously discuss and define its capabilities. The formalism introduced in this paper proposes a unified notation schema (symbolic denotations) and a transition-function-based approach to describing the state and control regardless of the control paradigm employed (i.e., deliberative, fuzzy, behavioral – this paper presents only the last one, but the others were presented in [39, 40]). The arguments of transition functions define the inputs of the modules, and the produced values indicate the outputs of the modules. In conjunction they define the communication links between the modules. If any of the possible arguments of the transition functions are not taken into consideration or an adequate link between modules is not established, the realization of a certain control idea will be inhibited. By using the presented formalism those flows are made evident, while, by simply following intuitive design methods, they might not become apparent until the implementation stage of a concrete control idea.

The proposed formalism has been used to define both the inter and intra agent communication structure. The communication links between the agents and their components are established in such a way that the arguments of the transfer functions computed by them are made available where necessary. In the implementation the agents are represented by several processes responsible for: the control of hardware of the effectors, effector task execution and sensor data aggregation. Each of those processes is responsible for the computation of certain transfer functions. The specific code of each transfer function is defined by the user of the framework and complies with both the available hardware and the task at hand. The main data structures within processes represent the communication packages containing the values of the arguments of the transfer functions, and the data structures representing the current and the computed (future, demanded) state of the effectors, sensor readings and configuration commands

and internal status of the controller. If object-oriented programming paradigm is used, as is the case with MRROC++, all those data structures are represented as objects. The transfer functions are embedded in the methods of those objects.

The paper is organized in the following way. Section 2 introduces the concepts of an embodied agent, and decomposes its control system into modules responsible for control of effectors, communication with other agents and aggregation of data obtained from sensors. Biologically inspired subdivision of receptors is proposed, assigning them to appropriate components of the control system. The state of effectors can be expressed in diverse ways, thus creating different ontologies. Finally the general form of transition functions is defined. Section 3 decomposes this general form of transition functions into subfunctions and introduces selection predicates. Those are the building blocks for defining behaviors recursively. The transition from the synchronous view of the system, based on the servo sampling period, into an asynchronous view, based on events, is presented as a natural consequence emerging from the introduction of predicates selecting state transition sequences of differing lengths. Both selection based and composition based behaviors are considered. Moreover, deterministic and probabilistic selection is covered by the formalism. In section 4 conclusions are formulated.

2 Embodied agents

A system consisting of n_a embodied agents is considered. The state of an agent a_j, $j = 1, \ldots, n_a$ is:

$$s_j = \langle c_j, e_j, V_j, T_j \rangle \tag{1}$$

c_j – state of the **control subsystem** of the agent (memory: variables, program),
e_j – state of the **effector** of the agent,
V_j – bundle of **virtual sensor** readings,
T_j – information from/to the other agents.

To be brief, and because of contextual obviousness, the denotations assigned to the subcomponents of the considered system and their state are not distinguished.

A bundle of virtual sensor readings contains n_{v_j} individual virtual sensor readings:

$$V_j = \langle v_{j_1}, \ldots, v_{j_{n_{v_j}}} \rangle \tag{2}$$

Each virtual sensor v_{j_k}, $k = 1, \ldots, n_{v_j}$, produces an aggregate reading from one or more hardware sensors – receptors. The data obtained from the receptors usually cannot be used directly in motion control, e.g., control of arm motion requires the grasping location of the object and not the bit-map delivered by a camera. In other cases a simple sensor 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. Thus the kth virtual sensor reading obtained by the agent a_j is formed as:

$$v_{j_k} = f_{v_{j_k}}(c_j, R_{j_k}) \tag{3}$$

where R_{j_k} is a bundle of receptor readings used for the creation of the kth virtual sensor reading.

$$R_{j_k} = \langle r_{j_{k_1}}, \ldots, r_{j_{k_{n_r}}} \rangle \tag{4}$$

where n_r is the number of receptor readings $r_{j_{k_l}}$, $l = 1 \ldots, n_r$, taken into account in the process of forming the reading of the kth virtual sensor of the agent a_j. It should be noted that (3) implies that the reading of the virtual sensor depends also on c_j. In this way the agent has the capability of configuring the sensor as well as delivering to the virtual sensor the relevant information about the current state of the agent (including its effector). This might be necessary in the case of computing the reading of a virtual sensor having its associated receptors mounted on the effector (e.g. artificial skin).

In higher living organisms receptors are divided into three categories, depending on the source of stimuli that they respond to.

- **Exteroceptors** – receptors that detect stimulus external to the body (e.g., vision, smell, touch). In technical systems they include the measuring devices gathering information from the environment. They are the source of perception of the environment.
- **Proprioceptors** – receptors that detect stimulus from inside of the limbs, i.e., muscles, tendons and joints. They enable perception of position of the limbs and body. In the case of embodied agents these are the devices for measuring the internal state of the effectors (e.g., encoders, resolvers). They determine the state of the effectors and not the environment, and thus are associated with the effector subsystem.
- **Interoceptors** – receptors that detect the stimulus from the internal organs of the body. Into this category fall nociceptors, which are sensitive to mechanical trauma, temperature and chemicals in extracellular fluids – i.e., pain receptors. In technical systems interoceptors supply information about the internal condition of the control subsystem and hence are associated with this subsystem. Error detection in the controller software can be treated as a result of monitoring interoceptors (errors as a source of pain).

All hardware sensors that take part in forming virtual sensor readings will be called, for brevity, **receptors**, although in majority of cases this will be just the exteroceptors. Interoceptors and proprioceptors that do not directly take part in the formation of virtual sensor readings will be associated either with the control subsystem itself or the effectors, so will not be treated as proper receptors in this discussion, i.e., will not be included in (4).

There are diverse methods of expressing effector state e_j, e.g., a manipulator can be perceived as (fig.1):

- a set of actuators (with angular or translational shaft positions),
- a sequence of links (with angular or translational joint positions),
- an end-effector (with a coordinate frame affixed to the end-effector).

It is worth noting that in a robot programming language or a framework the notion of a robot can disappear altogether (e.g., at object or task level). At the object level just the manipulated or avoided objects exist (other objects are irrelevant). The control system must deduce how to grasp and transfer the objects that have to be displaced (e.g., AI methods can be used or object attribute assignment and association can be utilized, as in the case of `TORBOL` [31]). At task level the actions that are to be executed are at the focus of attention.

In the case of walking machines the above mentioned items are similar, but refer to the legs. An extra item can be introduced to refer to the body of the machine (e.g., position of a frame affixed to the body). In the case of a wheeled vehicle the second item would refer to the angles of the wheels and the steering device, the third item would be unnecessary, but a coordinate frame affixed to the body would be useful. Each of those items forms a different image of the device – by using those notions the programmer creates a different model of the device that is being controlled. Sometimes we say that by creating a different view of the device we create another ontology.

The responsibility of the agent's control subsystem c_j is to: gather information about the environment through the associated virtual sensor bundle V_j, obtain the information from the other agents $a_{j'}$ ($j' \neq j$), monitor the state of its own effector e_j, and to process all of this information to produce: a new state of the effector e_j, influence the future functioning of the virtual sensors V_j, and communicate with the other agents $a_{j'}$. As a side effect, the internal state of the control subsystem c_j changes. Thus three types of components of the control subsystem must be distinguished:

- input components providing the information about: the state of the effector, virtual sensor readings and the messages obtained from the other agents (they use a leading subscript x),
- output components exerting influence over: the state of the effector, configuration of virtual sensors and the messages to be transmitted to the other agents (they use a leading subscript y),
- other resources needed for data processing within the control subsystem (without a leading subscript).

The following components are distinguished (fig.2):

- ${}_xc_{e_j}$ – input image of the effector (a perception of the effector by the control subsystem as produced by processing the input signals transmitted from the effector to the control subsystem, e.g., motor shaft positions, joint angles, end-effector location – they form diverse ontologies),
- ${}_xc_{V_j}$ – input images of the virtual sensors (current virtual sensor readings – control subsystem's perception of the sensors and through them of the environment),
- ${}_xc_{T_j}$ – input of the inter-agent transmission (information obtained from other agents),
- ${}_yc_{e_j}$ – output image of the effector (this is a perception of the effector by the control subsystem as needed to produce adequate control signals),
- ${}_yc_{V_j}$ – output images of the virtual sensors (current configuration and commands controlling the virtual sensors),
- ${}_yc_{T_j}$ – output of the inter-agent transmission (information transmitted to other agents),
- c_{c_j} – all the other relevant variables taking part in data processing within the agent's control subsystem.

From the point of view of the system designer the state of the control subsystem changes at a servo sampling rate or a low multiple of that. If i denotes the current instant, the next considered instant is denoted by $i+1$. The control subsystem uses:

$$ {}_xc_j^i = \langle c_{c_j}^i,\ {}_xc_{e_j}^i,\ {}_xc_{V_j}^i,\ {}_xc_{T_j}^i \rangle \tag{5} $$

to produce:

$$ {}_yc_j^{i+1} = \langle c_{c_j}^{i+1},\ {}_yc_{e_j}^{i+1},\ {}_yc_{V_j}^{i+1},\ {}_yc_{T_j}^{i+1} \rangle \tag{6} $$

and hence:

$$ \begin{cases} c_{c_j}^{i+1} &= f_{c_{c_j}}(c_{c_j}^i,\ {}_xc_{e_j}^i,\ {}_xc_{V_j}^i,\ {}_xc_{T_j}^i) \\ {}_yc_{e_j}^{i+1} &= f_{c_{e_j}}(c_{c_j}^i,\ {}_xc_{e_j}^i,\ {}_xc_{V_j}^i,\ {}_xc_{T_j}^i) \\ {}_yc_{V_j}^{i+1} &= f_{c_{V_j}}(c_{c_j}^i,\ {}_xc_{e_j}^i,\ {}_xc_{V_j}^i,\ {}_xc_{T_j}^i) \\ {}_yc_{T_j}^{i+1} &= f_{c_{T_j}}(c_{c_j}^i,\ {}_xc_{e_j}^i,\ {}_xc_{V_j}^i,\ {}_xc_{T_j}^i) \end{cases} \tag{7} $$

or more compactly:

$$ {}_yc_j^{i+1} = f_{c_j}({}_xc_j^i) \tag{8} $$

Formula (8) is a prescription for evolving the state of the system, thus it has to be treated as a program of the agent's behavior. For any agent exhibiting useful behaviors this function would be very complex, because it describes the actions of the system throughout its existence. The complexity of this function renders impractical the representation of the program of agent's actions as a single function. The function (8) has to be decomposed to make the specification of the agent's program of actions comprehensible and uncomplicated.

In the process of producing the output values ${}_yc_j^{i+1}$ the values of inputs ${}_xc_j^i$ and internal variables $c_{c_j}^i$ are used. The internal variables change their values, thus $c_{c_j}^{i+1}$ is created – those will be the values of internal variables at the onset of motion step $i+1$. All of the mentioned quantities are stored in the agent's memory, thus their values form the total state of the agent's control subsystem. Obviously the process of computing the output values from the values of the input and internal agent's variables takes time, so the valid

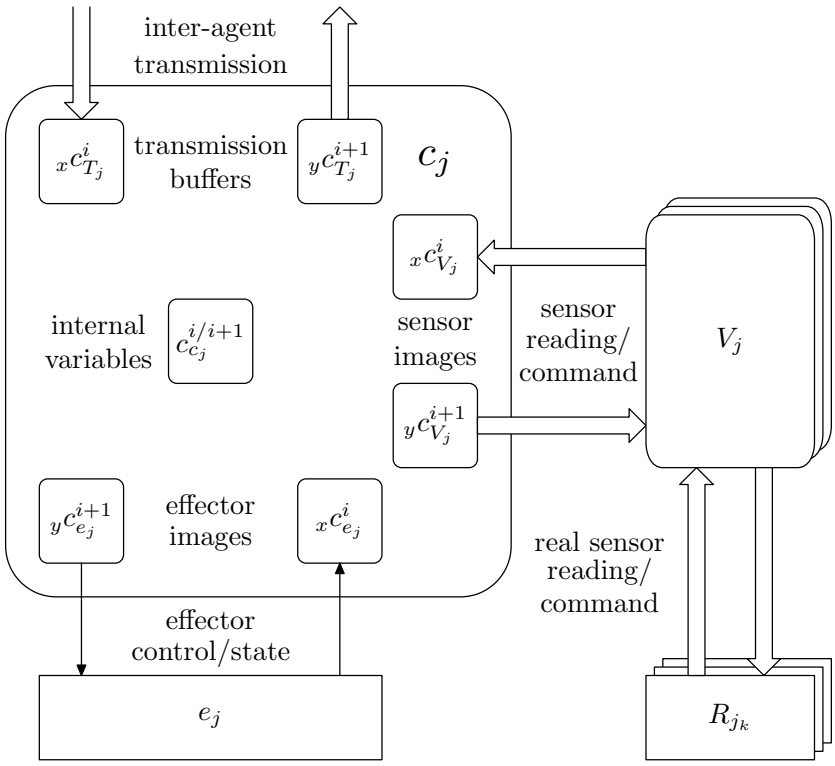

Figure 2: A single embodied agent a_j, $j = 1, \ldots, n_a$

total state is obtained after the result of those computations is ready. This result is transferred to the other components of the system. Hence, the control subsystem total state holds the data obtained from other components of the system and values of its own internal variables, both valid at the start of step i, and the output values computed during the initial part of the motion step i (those will be transmitted to the other components of the system at the end of motion step i, thus controlling the system in motion step $i+1$). All of those values are memorized within the agent during the later part of motion step i, thus they compose the total internal state of the control subsystem:

$$c_j^i = \langle c_{c_j}^{i/i+1},\ {}_x c_{e_j}^i,\ {}_y c_{e_j}^{i+1},\ {}_x c_{V_j}^i,\ {}_y c_{V_j}^{i+1},\ {}_x c_{T_j}^i,\ {}_y c_{T_j}^{i+1} \rangle \tag{9}$$

The total state contains the input values, internal variables and the produced output (control). All of those are valid after the computations of (7) are completed. The focus of our attention is the creation of ${}_y c_j^{i+1}$ from ${}_x c_j^i$, i.e., the form of transition functions (7). The symbol $c_{c_j}^{i/i+1}$ underscores the change of state of internal variables due to computations of (7) within the step i.

3 Behaviour of an agent

Internal functioning of an agent is defined by the transition functions (7) represented in compact form by (8). The flexibility of a programming framework is attributed to the ability of expressing diverse approaches to programming the actions of each agent,

and so the proposed formal description should enable easy formulation of diverse control strategies. In the case of a robot programming framework one should concentrate on the definition of motion commands. One feasible approach is: instead of providing a single function (8), describing the motion of an agent throughout its life, many simpler functions are specified, defining small motion segments, and the final result is obtained by their composition. Thus instead of a single function f_{c_j}, n_f partial functions are defined:

$$ {}_{y}c_j^{i+1} = {}^{m}f_{c_j}({}_{x}c_j^i), \qquad m = 1, \ldots, n_f \tag{10} $$

Variability of agents is due to the diversity of those functions. The more functions of this type are provided by a programming framework the more types of agents can be constructed. However, the means of selecting among those functions must be provided by the framework. In the case of a purely reactive system, sometimes also called a reflex system, the choice of the function ${}^{m}f_{c_j}$ is based on testing predicates ${}^{q}p_{c_j}$, $q = 1, \ldots, n_p$, which take as arguments only the components of ${}_{x}c_{V_j}^i$. In pseudo-code it can be expressed as:

$$ \texttt{if } {}^{q}p_{c_j}({}_{x}c_{V_j}^i) \texttt{ then } {}_{y}c_j^{i+1} := {}^{m}f_{c_j}({}_{x}c_j^i) \texttt{ endif} \tag{11} $$

Here we shall consider systems that decide, which function to choose, on the basis of all of the available information, i.e. all components of ${}_{x}c_j^i$. Moreover, in actual systems an endless loop containing the conditional instruction (11) must be constructed. Thus, for systems, where only one predicate can be true, the pseudo-code will assume the following form:

$$
\begin{array}{l}
\texttt{loop} \\
\quad \text{// Determine the current state of the agent} \\
\quad e_j \rightarrowtail {}_{x}c_{e_j}^i; \quad V_j \rightarrowtail {}_{x}c_{V_j}^i; \quad c_{T_{j'}} \rightarrowtail {}_{x}c_{T_j}^i; \\
\quad \text{// Compute the next state of the agent} \\
\quad \texttt{if } {}^{1}p_{c_j}({}_{x}c_j^i) \texttt{ then } {}_{y}c_j^{i+1} := {}^{1}f_{c_j}({}_{x}c_j^i) \texttt{ endif} \\
\quad \texttt{if } {}^{2}p_{c_j}({}_{x}c_j^i) \texttt{ then } {}_{y}c_j^{i+1} := {}^{2}f_{c_j}({}_{x}c_j^i) \texttt{ endif} \\
\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\quad \texttt{if } {}^{n_p}p_{c_j}({}_{x}c_j^i) \texttt{ then } {}_{y}c_j^{i+1} := {}^{n_p}f_{c_j}({}_{x}c_j^i) \texttt{ endif} \\
\quad \text{// Transmit the results of computations} \\
\quad {}_{y}c_{e_j}^{i+1} \rightarrowtail e_j; \quad {}_{y}c_{V_j}^{i+1} \rightarrowtail V_j; \quad {}_{y}c_{T_j}^{i+1} \rightarrowtail c_{T_{j'}}; \\
\quad i := i + 1; \\
\texttt{endloop}
\end{array}
\tag{12}
$$

where double slash precedes the comments and the symbol "$\rightarrowtail$" denotes transmission of data. Those transmissions result in: data input, execution of motion by the effectors, configuration of virtual sensors and transmission of messages to other agents. In each step i one iteration of the loop (12) will be executed. In this case in each iteration, and thus in each control step i, one and only one out of the n_p predicates ${}^{q}p_{c_j}$ must be true, hence a single function ${}^{m}f_{c_j}$ is selected as the one designating the next state of the agent. Usually in such a case $n_p = n_f$ and therefore the endless loop contains n_p instructions of the type (11).

From the point of view of clarity of the description of the task that is to be executed by the system it is useful to group the steps of the commanded evolution of control subsystem state into sequences, that will be called primitive behaviors. The system that is described in [33, 35] performs the selection of the function ${}^{m}f_{c_j}$ for several consecutive steps, i.e., the selection is less frequent than the evaluation of the function, so the reaction is composed of several steps. In that case a behavior is defined as a sequence of total states:

$$ {}^{q}b_j^i = {}^{q}b_j = \{c_j^{i+1}, c_j^{i+2}, \ldots, c_j^{i+n_s}\} \tag{13} $$

where n_s is the number of steps in a behavior (reaction) and q denotes a numeric identifier of this reaction. Each sequence of states $c_j^{i+1}, c_j^{i+2}, \ldots, c_j^{i+n_s}$ is generated by one of the functions ${}^{m}f_{c_j}$, thus this function is defining the primitive behavior. At the level of behaviors (i.e., concatenations of motion steps) the system changes its image from time driven into event driven, thus the index i is discarded. However, it should be remembered that each ${}^{k}b_j$ starts at a certain instant i and is influenced by the current control subsystem total state c_j^i, in which i appears explicitly.

The pseudo-code (11) represents a single-step behavior, i.e., $n_s = 1$. In the case of a multi-step behavior the pseudo-code assumes the following form:

$$ \texttt{if } {}^{q}p_{c_j}({}_{x}c_j^i) \texttt{ then } {}^{q}b_j({}_{x}c_j^i) \texttt{ endif} \tag{14} $$

In the case (14) the decision as to which behavior should be executed is taken once every n_s steps. Nevertheless, the transition between the state $c_j^{i+\epsilon}$ and $c_j^{i+\epsilon+1}$, where $\epsilon = 0, \ldots, n_s - 1$ is still computed on the basis of the functions ${}^{m}f_{c_j}$. The control program is composed of an endless loop containing a sequence of instructions of the form (14). In that case each iteration of the loop contains several control steps i.

```
loop
  // Determine the current state of the agent
  e_j ⇀ _x c^i_{e_j};   V_j ⇀ _x c^i_{V_j};   c_{T_j'} ⇀ _x c^i_{T_j};
  // Select and execute the next behavior
  if  ^1p_{c_j}(_x c^i_j)  then  ^1b_j(_x c^i_j)  endif
  if  ^2p_{c_j}(_x c^i_j)  then  ^2b_j(_x c^i_j)  endif
  ..................................
  if  ^{n_p}p_{c_j}(_x c^i_j)  then  ^{n_p}b_j(_x c^i_j)  endif
  //i := i + n_s;
endloop
```

(15)

Here the required computations (i.e., computation of ${}_{y}c_j^{i+\epsilon}$, $\epsilon = 1, \ldots, n_s$) and the execution of behaviors (i.e., transmissions: ${}_{y}c_{e_j}^{i+1} \rightarrowtail e_j$, ${}_{y}c_{V_j}^{i+1} \rightarrowtail V_j$, ${}_{y}c_{T_j}^{i+1} \rightarrowtail c_{T_j'}$) are bundled together within ${}^{q}b_j({}_{x}c_j^i)$, $q = 1, \ldots, n_p$. The loop can be constructed in such a way that if none of the predicates ${}^{q}p_{c_j}({}_{x}c_j^i)$ is true a default behavior, called the main reaction or a goal pursuing reaction, is executed. The other reactions deal with some abnormal situations – hindering attaining of the goal.

If we use (13) and (15) as a combined definition of a behavior a recursive definition results, where (13) defines a primitive behavior, and (15) defines a complex behavior consisting of subbehaviors. In that case within the behavior a local set of predicates can be used, producing a hierarchy of reactions with a variable granularity. One way of assigning predicates to levels of behavior is to look at the time needed to process the information from the sensors, i.e., ${}_x c^i_{V_j}$. The more time required to perform the processing the higher the level of behavior that the associated predicate triggers.

In the case (12), where the computation of the next effector state and its execution are separate, several predicates ${}^q p_{c_j}$ can be true simultaneously. In that case the values of several partial functions ${}^m f_{c_j}$ have to be composed together. Many composition operators can be conceived. Competitive methods are based on selecting one value out of the computed values, e.g.:

$$ {}_y c_j^{i+1} = \max_m \{ {}^m f_{c_j}({}_x c_j^i) \} \tag{16} $$

where the values of ${}^m f_{c_j}({}_x c_j^i)$ must be real numbers.

The mechanisms of inhibition (elimination of some components by others) and suppression (substitution of some components by others) introduced by Rodney Brooks [7, 8, 4] can be produced by supplying an adequate selection function.

Cooperative methods are based on some form of superposition (e.g., linear combination) of the computed values:

$$ {}_y c_j^{i+1} = \sum_{m=1}^{n_f} {}^m f_{c_j}({}_x c_j^i) \tag{17} $$

The pseudo-code in this case is a modification of (12):

```
loop
  // Determine the current state of the agent
  e_j ↣ x c^i_{e_j};   V_j ↣ x c^i_{V_j};   c_{T_{j'}} ↣ x c^i_{T_j};
  for q = 1, ..., n_p : clear (q_y c_j^{i+1});
  // Compute the next control subsystem state
  if 1 p_{c_j}(x c_j^i) then 1_y c_j^{i+1} := 1 f_{c_j}(x c_j^i) endif
  if 2 p_{c_j}(x c_j^i) then 2_y c_j^{i+1} := 2 f_{c_j}(x c_j^i) endif
  ..........................................                          (18)
  if n_p p_{c_j}(x c_j^i) then n_p_y c_j^{i+1} := n_p f_{c_j}(x c_j^i) endif
  // Compute the aggregate control
    for q = 1, ..., n_p : y c_j^{i+1} := composition(q_y c_j^{i+1});
  // Transmit the results
  y c^{i+1}_{e_j} ↣ e_j;   y c^{i+1}_{V_j} ↣ V_j;   y c^{i+1}_{T_j} ↣ c_{T_{j'}};
  i := i + 1;
endloop
```

The proposed program structures either rely on the evaluation of the predicates in each step (e.g., (12) and (18)) or on a fixed length of the sequence within a primitive behavior (as in (15)). A very general concept of motion instructions for multi-robot systems was

introduced in MRROC++ [36, 37, 38] (fig. 3). It can be extended even further to include direct inter-agent communication. The Move instruction has similar properties as (15). Here the function f_{c_j} (8), due to its inherent complexity, has been decomposed into a sequence of separate pairs of functions ${}^{m}f'_{c_j}$ and ${}^{m}f''_{c_j}$. Each pair influences the agent during i_f steps, where a third function ${}^{m}f_{\tau_j}$ determines the number of steps i_f. The function ${}^{m}f'_{c_j}$ defines the action of the agent in the first step – usually, from the computational point of view, it differs from all the other motion steps within the instruction execution. The function ${}^{m}f''_{c_j}$ specifies the behavior of the agent in all motion steps, but the first one. In this way, one function (f_{c_j}), defining the system evolution for whole of its lifetime, has been divided into a sequence of triplets of functions ${}^{m}f'_{c_j}$, ${}^{m}f''_{c_j}$ and ${}^{m}f_{\tau_j}$, which specify the actions of an agent for a period of time when a single Move instruction is executed. Each function ${}^{m}f_{\tau_j}$ determines when the control system should switch from one Move instruction to another. Each such instruction is governed by a different set of functions: ${}^{m}f'_{c_j}$, ${}^{m}f''_{c_j}$ and ${}^{m}f_{\tau_j}$. Thus those functions should constitute the parameters of the Move instruction. This solution was adopted in MRROC++.

MRROC++ contains an abstract class called a motion generator. The two function pairs: $({}^{m}f'_{c_j},\ {}^{m}f_{\tau_j})$ and $({}^{m}f''_{c_j},\ {}^{m}f_{\tau_j})$ are embedded within the two methods of this class. The programmer derives descendant classes from this class and simultaneously defines the code of those function pairs. The code of the Move procedure (similar to the one from fig. 3) invokes those methods causing the agent to execute a part of its task. Sequences of Move instructions in conjunction with the standard C++ loops and conditional statements enable the definition of the whole task. Thus the programmer concentrates only on two aspects of control: the motion generator of each Move instruction, and the composition of the Move instructions into a program executing the demanded task. Other aspects of program implementation are provided by the framework.

Let us limit the arguments of the Move instruction to just the three relevant to this discussion, i.e., ${}^{m}f'_{c_j}$, ${}^{m}f''_{c_j}$ and ${}^{m}f_{\tau_j}$. Now a behavior can be composed of a sequence of Move$({}^{m}f'_{c_j},\ {}^{m}f''_{c_j},\ {}^{m}f_{\tau_j})$ instructions.

$$\texttt{if}\ {}^{q}p_{c_j}(c_j^i)\ \texttt{then}\ \texttt{Move(}\ {}^{m}f'_{c_j},\ {}^{m}f''_{c_j},\ {}^{m}f_{\tau_j}\texttt{);}\ \ldots\ \texttt{endif} \tag{19}$$

Here the duration of the execution of each component (i.e., Move) in a sequence is determined by its termination function ${}^{m}f_{\tau_j}$. Each of those components has two separate functions generating the steps: one for the first step (${}^{m}f'_{c_j}$) and one for every other step (${}^{m}f''_{c_j}$). The sequence is chosen according to the value of the predicate ${}^{q}p_{c_j}$, so the decision

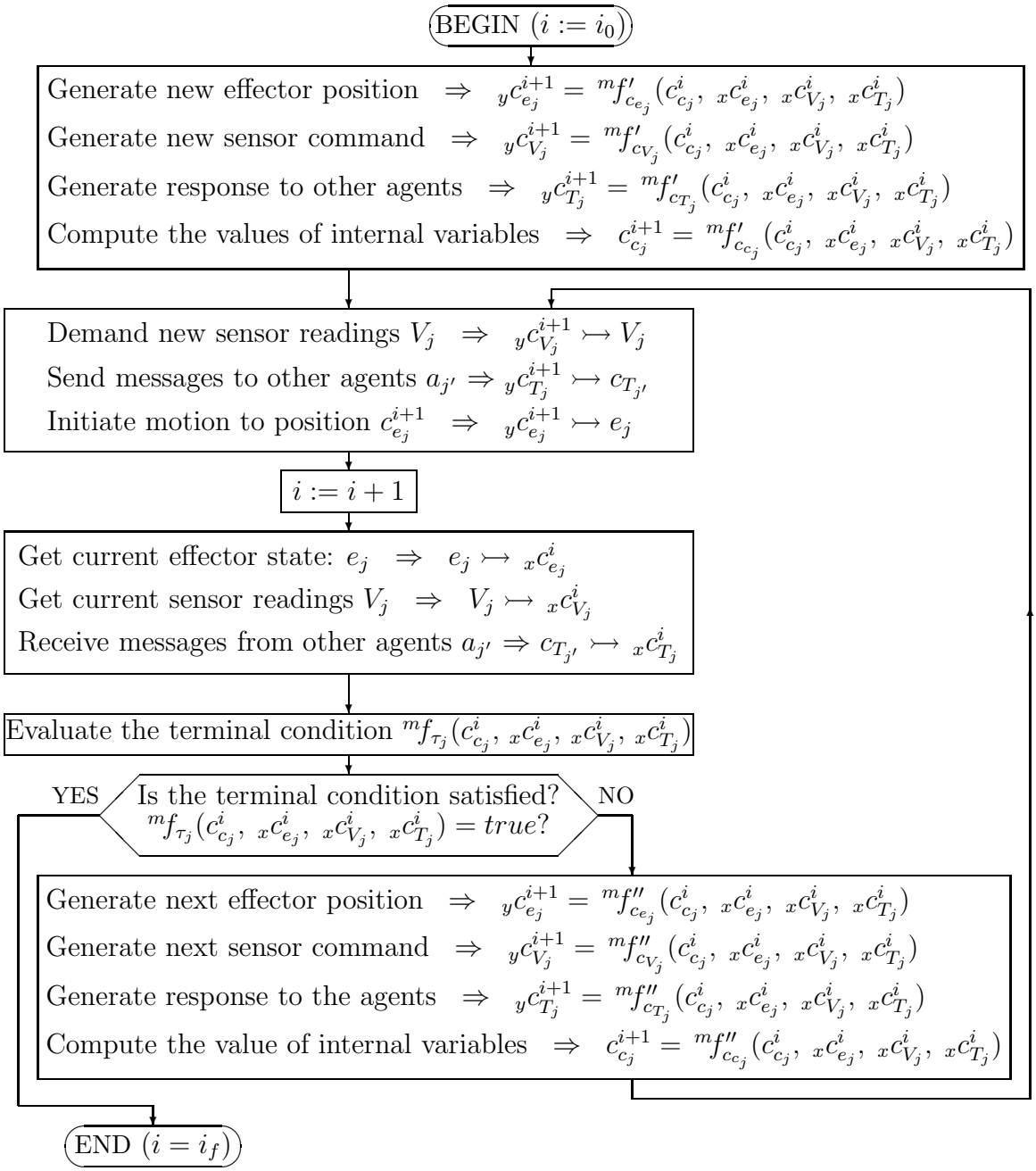

Figure 3: `Move` instruction of an agent a_j

process is not invoked too often. Hence the resulting pseudo-code is:

```
loop
 // Determine the current state of the agent
 e_j ↣ xc^i_{e_j};   V_j ↣ xc^i_{V_j};   c_{T_j'} ↣ xc^i_{T_j};
 // Select and execute the next behavior
 if 1p_{c_j}(xc^i_j) then Move(1f'_{c_j}, 1f''_{c_j}, 1f_{τ_j}); ... endif
 if 2p_{c_j}(xc^i_j) then Move(2f'_{c_j}, 2f''_{c_j}, 2f_{τ_j}); ... endif
 ..........................................
 if np_{c_j}(xc^i_j) then Move(mf'_{c_j}, mf''_{c_j}, mf_{τ_j}); ... endif
endloop
```
(20)

In the case of the pseudo-code (20) only one predicate ${}^{q}p_{c_j}$ can be true at the moment the decision is being made. To enable the situation where several predicates can be true simultaneously some modifications are necessary. The `Move` instruction not only computes the value of ${}_{y}c_j^{i+1}$, but also causes its transmission to the other subsystems of the agent, thus no composition of partial results is possible within the program (20). To make this possible a slight modification of the `Move` definition is necessary and an introduction of a separate entity responsible for the composition of partial results obtained from each `Move` instruction being executed in parallel. The modification of the flowchart defining the `Move` instruction (fig. 3) consists in exchanging the contents of the operational block initiating the motion to position ${}_{y}c_{e_j}^{i+1}$ (i.e., executing: ${}_{y}c_{e_j}^{i+1} \rightarrowtail e_j$) for the transmission of partial result ${}^{q}c_j^{i+1}$ to this new entity (e.g., a thread). Once this entity has collected the partial results from all currently active `Move` instructions, it can compute the final value according to one of the formulas (16) or (17) and then transmit the result to the other subsystems of the agent for execution.

Agents can behave probabilistically too. In this case each behavior ${}^{q}b_j({}_{x}c_j^i)$, $q = 1, \ldots, n_p$, is executed only when an associated predicate is *true* and a randomly chosen number from the range $[0, 1]$ is above a threshold value θ_p. The comparison of the random number and the threshold can be incorporated directly into the predicate itself and thus the algorithm (15) or (20) can be used. An indeterministic system results.

A more realistic system can be produced, if the probability of performing a certain action is associated with the level of stimulus. Swarm intelligence systems [6] mimicking the behavior of ants or bees frequently rely on such an approach. The stimulus can come from the environment (in this case ${}_{x}c_{V_j}^i$ is used) or from the other agents (then ${}_{x}c_{T_j}^i$ is utilized). Let us assume that one of the the components of ${}_{x}c_{V_j}^i$ or ${}_{x}c_{T_j}^i$ is used. It is singled out by the symbol ζ, so ${}_{x}^{\zeta}c_{V/T_j}^i$ is being considered. The probability of executing a certain behavior can be expressed as [6]:

$$P_\theta = \frac{({}_{x}^{\zeta}c_{V/T_j}^i)^{n_\theta}}{({}_{x}^{\zeta}c_{V/T_j}^i)^{n_\theta} + (\theta_p)^{n_\theta}} \tag{21}$$

where n_θ is an appropriately chosen positive integer – the simulation experiments in [6] were carried out with $n_\theta = 2$. In such a system, when the stimulus is low (i.e., ${}_{x}^{\zeta}c_{V/T_j}^i \ll \theta_p$) the probability of executing an associated action is next to nil, but if the stimulus is high (i.e., ${}_{x}^{\zeta}c_{V/T_j}^i \gg \theta_p$) the action is executed almost certainly. In the case of ants, if an individual detects a high pheromone level along some path the probability of following it (executing the behavior of running along this path) is high. On the contrary, if the pheromone level is low the ant is much less likely to follow that trail. An adequately located threshold enables the adjustment between following standard routs and foraging in an unknown or little known territory. The threshold θ_p does not have to be kept constant. It can be adapted according to certain other factors, so the inclination of an agent to behave in some way can vary according to those factors. Nevertheless, the

primary source of variability in the agent's behavior is the stimulus intensity. Performance of certain actions might reduce this intensity and so those actions are less likely to be repeated (negative feedback). On the other hand some other actions can produce the opposite effect. Performing those actions increases the level of stimulus, thus the agent is motivated to repeat them (positive feedback). In a way the agent becomes addicted to those actions.

Sometimes actions are performed only when stimulus is within a certain range. The probability of executing an action when the stimulus is below a threshold or above a limit is very low. For example, on a cold day, when we are far away from the fire or very near it we are not likely to extend our hands to warm them up. In the former case, because they would get even colder, and in the latter, because we would get burned. Only at a distance within a certain range the hands would be pleasantly warmed, thus the probability of executing an action consisting in extending the hands would be high only within that range. To express this fact Gauss function can be used:

$$P_\theta = e^{-[\psi(\sum_x c^i_{V/T_j} - \theta_p)]^2} \tag{22}$$

where θ_p is the mid value in the range and ψ governs the steepness of the rise and fall of probability P_θ and the size of the range. The stimulus is the temperature.

4 Conclusions

Transition function based formalism introduces rigor into the design and implementation of multi agent systems. It decomposes a large system into components that can be designed and implemented by providing the code for the specified functions. Each of the control subsystem components (9) can be treated as an object in an object-oriented programming sense. Thus the communication with the other agents, effectors or virtual sensors can be handled internally by the methods of these objects (`MRROC++` uses this method). The objects provide, through their public interfaces, only the data that is necessary for the computation of the control of the agent. Those objects provide data that is utilized by transition functions (7) resident in the control subsystem to compute the next state of this subsystem.

The pieces of code (12), (15), (18) and (20) are the general patterns that can be used in creating behavioral controllers. Those are the patterns of the programming framework, but they can be optimized for efficiency of execution. Moreover, certain hints have been given how to create predicates both for deterministic and probabilistic systems. On the one hand the patterns are very general, but on the other hand the programmer has only to focus on coding elementary behaviors in the form of functions (10), sequences (13) or sequences of `Move` instructions. Both single function and simultaneous multiple function computation has been considered. Moreover, both competitive (selection based) and cooperative (superposition based) methods of computing the final value have been investigated. This paper concentrated just on behavioral control, but the presented formalism was also used used to describe fuzzy [41] and deliberative [39] systems.

Acknowledgements

This work was supported by Polish Ministry of Science and Information Technology grant: 4 T11A 003 25.

References

[1] User's guide to VAL II: Programming manual ver.2.0. Unimation Incorporated, A Westinghouse Company, August 1986.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab M., and Ingrand F. An architecture for autonomy. *Int. J. of Robotics Research*, 17(4):315–337, 1998.

[3] A. P. Ambler and D. F. Corner. RAPT1 user's manual. Department of Artificial Intelligence, University of Edinburgh, 1984.

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

[5] C. Blume and W. Jakob. *Programming Languages for Industrial Robots*. Springer-Verlag, Berlin, 1986.

[6] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, Oxford, 1999.

[7] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[8] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, (47):139–159, 1991.

[9] H. Bruyninckx. Orocos – open robot control software. http://www.orocos.org/, 2002.

[10] S. Fleury and M. Herrb. Genom user's guide. Report, LAAS, Toulouse, December 2001.

[11] V. Hayward, L. Daneshmend, and S. Hayati. An overview of KALI: A system to program and control cooperative manipulators. In K. Waldron, editor, *Advanced Robotics*, pages 547–558. Springer-Verlag, Berlin, 1989.

[12] V. Hayward and S. Hayati. KALI: An environment for the programming and control of cooperative manipulators. In *Proc. American Control Conference*, pages 473–478. 1988.

[13] V. Hayward and R. P. Paul. Robot manipulator control under unix RCCL: A robot control C library. *Int. J. Robotics Research*, 5(4):94–111, Winter 1986.

[14] M. E. Markiewicz and C. J. P. Lucena. Object oriented framework development. *ACM Crossroads*, 7(4), 2001.

[15] K. Mianowski. Parallel and serial-parallel robots for the use of technological applications. In *Proceedings of the Parallel Kinematic Machines PKM'99, November, Milano, Italy*, pages 39–46. 1999.

[16] K. Mianowski and K. Nazarczuk. Parallel drive of manipulator arm. In *Proceedings of the 8th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy 8, Cracow, Poland, 2–6 July*, pages 143–150. 1990.

[17] K. Mianowski, K. Nazarczuk, M. Wojtyra, and S. Ziętarski. Application of the unigraphics system for milling and polishing with the use of rnt robot. In *Proceedings of the Workshop for the users of UNIGRAPHICS system, Frankfurt, November*, pages 98–104. 1999.

[18] E. R. Morales. Generis: The ec-jrc generalised software control system for industrial robots. *Industrial Robot*, 26(1):26–32, 1999.

[19] S. Mujtaba and R. Goldman. AL users' manual. Stanford Artificial Intelligence Lab., January 1979.

[20] K. Nazarczuk and K. Mianowski. Polycrank – fast robot without joint limits. In *Proceedings of the 12-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'12, Vienna, 6–9 June*, pages 317–324. Springer-Verlag, 1995.

[21] K. Nazarczuk, K. Mianowski, A. Olędzki A., and C. Rzymkowski. Experimental investigation of the robot arm with serial-parallel structure. In *Proceedings of the 9-th World Congress on the Theory of Machines and Mechanisms, Milan, Italy*, pages 2112–2116. 1995.

[22] K. Nazarczuk, K. Mianowski, and S. Łuszczak. Development of the design of polycrank manipulator without joint limits. In *Proceedings of the 13-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy 13, Zakopane, Poland, 3–6 July*, pages 285–292. 2000.

[23] R. Paul. WAVE – a model based language for manipulator control. *The Industrial Robot*, pages 10–17, March 1977.

[24] L. Petersson, D. Austin, and H. Christensen. Dca: A distributed control architecture for robotics. In *Proc. Int. Conference on Intelligent Robots and Systems IROS'01.* 2001.

[25] J. Pritchard. *COM and COBRA Side by Side: Architectures, Strategies, and Implementations.* Addison-Wesley, Reading, 1999.

[26] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, N.J., 1995.

[27] R. Simmons and D. Apfelbaum. A task description languagefor robot control. In *International Conference on Itelligent Robots and Systems IROS'98. Victoria, Canada*. October 1998.

[28] R. Simmons, R. Goodwin, C. Fedor J., and Basista. Task control architecture: Programmer's guide to version 8.0. Carnegie Mellon University, School of Computer Science, Robotics Institute, May 1997.

[29] W. Szynkiewicz. Motion planning for multi-robot systems with closed kinematic chains. In *Proceedings of the 9th IEEE International Conference on Methods and Models in Automation and Robotics MMAR'2003, Międzyzdroje*, pages 779–786. August 25-28 2003.

[30] R. H. Taylor, P. D. Summers, and J. M. Meyer. AML: A manufacturing language. *International Journal of Robotics Research*, 1(3):842–856, 1982.

[31] C. Zieliński. TORBOL: An object level robot programming language. *Mechatronics*, 1(4):469–485, 1991.

[32] C. Zieliński. Flexible controller for robots equipped with sensors. In *9th Symp. Theory and Practice of Robots and Manipulators, Ro.Man.Sy'92, Udine, Italy, Lect. Notes: Control and Information Sciences 187*, pages 205–214. Springer-Verlag, Berlin, September 1–4, 1992 1993.

[33] C. Zieliński. Reaction based robot control. *Mechatronics*, 4(8):843–860, 1994.

[34] C. Zieliński. Control of a multi-robot system. In *2nd Int. Symp. Methods and Models in Automation and Robotics MMAR'95, Międzyzdroje, Poland*, pages 603–608. 30 Aug.–2 Sept. 1995.

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

[36] C. Zieliński. Object–oriented programming of multi–robot systems. In *Proc. 4th Int. Symp. Methods and Models in Automation and Robotics MMAR'97, Międzyzdroje, Poland*, pages 1121–1126. August 26–29 1997.

[37] C. Zieliński. The MRROC++ system. In *1st Workshop on Robot Motion and Control, RoMoCo'99, Kiekrz, Poland*, pages 147–152. June 28–29 1999.

[38] C. Zieliński. By how much should a general purpose programming language be extended to become a multi-robot system programming language? *Advanced Robotics*, 15(1):71–95, 2001.

[39] C. Zieliński. A unified formal description of behavioural and deliberative robotic multi-agent systems. In *Proc. 7th IFAC International Symposium on Robot Control SYROCO 2003, Wrocław, Poland*, volume 2, pages 479–486. September 1–3 2003.

[40] C. Zieliński. Specification of behavioural embodied agents. In K. Kozłowski, editor, *Fourth International Workshop on Robot Motion and Control, RoMoCo'04, Puszczykowo, Poland*, pages 79–84. June 17–20 2004.

[41] C. Zieliński. Formalization of programming frameworks for multi-robot systems. In *8-th National Conference on Robotics, Polanica Zdrój*. June, 23–25 2004 (in Polish).

[42] C. Zieliński, K. Mianowski, K. Nazarczuk, and W. Szynkiewicz. A prototype robot for polishing and milling large objects. *Industrial Robot*, 30(1):67–76, January 2003.

[43] C. Zieliński and W. Szynkiewicz. Control of two 5 d.o.f. robots manipulating a rigid object. In *IEEE Int. Symp. on Industrial Electronics ISIE'96, Warsaw, Poland*, volume 2, pages 979–984. June 17–20 1996.