POLITECHNIKA WARSZAWSKA

ISSN 0137–2319

PRACE NAUKOWE • ELEKTRONIKA • z. 107

Cezary Zieliński

ROBOT PROGRAMMING METHODS

Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1995

Publishing House of Warsaw University of Technology, Warsaw 1995

PRACE NAUKOWE POLITECHNIKI WARSZAWSKIEJ
z. 107 Electronics 1995

Cezary Zieliński

Institute of Control and Computation Engineering
Warsaw University of Technology

ROBOT PROGRAMMING METHODS

Manuscript received 21.04.1995

This monograph describes different robot programming methods from the point of view of the programmer and the one who has to devise the structure of the system and to implement it. This work gives some insight into the theory that should underlie robot programming. This dissertation shows how the science of robot programming evolved. The author's own experience, gained by defining and implementing different robot programming and control systems, led to a generalisation and formalisation which is described in this work. The formalism is based on the decomposition of a robot system into three parts: effectors, receptors and control sub-system. Influence of instruction execution on the state of each of the system components is considered. The method of utilising sensors during the execution of an instruction is dealt with. The completeness of the language is checked by looking at the possibility of controlling through instructions each part of the system. Different types of instruction semantics are considered. Languages with few but complex instructions are compared with languages with large sets of instructions but simple semantics. The intermediate solution is regarded as the most promising. Robot languages are classified into joint, manipulator, object and task level. Examples of languages of these classes are presented, e.g. WAVE, VAL II, AL, RAPT, TORBOL. On-line, off-line and hybrid programming techniques are discussed. Different methods of implementing robot programming languages are dealt with. In the case of robot programming for research purposes, the implementation of robot specific libraries of procedures and processes submerged in a universal programming language is considered superior to the implementation of specialised languages. Implementation of libraries by using structured (e.g. ROPAS, PASRO) and object-oriented (e.g. ROOPL) programming paradigms are also discussed. The issue of robot programming is inseparable from the problems of robot controller design. Industrial controllers have a closed structure and are well suited to simple industrial tasks. Complex, often changing tasks, especially those requiring the incorporation of many diverse sensors, need controllers with open structures. Research-oriented controllers with an open structure for single– (e.g. RORC) and multi-robot (e.g. MRROC) systems are dealt with. The structures of those systems were deduced using the above-mentioned formalism. Moreover, it proved instrumental in designing new control algorithms (e.g. sensor-based reactive robot control).

Notation

b_j – instance of reaction B_j

B_j – j–th reaction

B_0 – main reaction – task goal pursuing reaction

c – control subsystem state

C – control subsystem state space

c^i – control subsystem state in instant i

c_d – state of instruction designator (in the control subsystem)

c_p – state of the part of the control subsystem containing the executed program

c_v – state of variables (in the control subsystem)

c_{vv} – state of variables not containing the virtual sensor reading

e – state of effectors

E – effector state space

e_* – single planned effector state

e_*^* – sequence of planned effector states

e^i – state of effectors in instant i

e_*^i – i–th planned effector state

E^E – effector state sub-space in which the error condition is satisfied

E^T – effector state sub-space in which the terminal condition is satisfied

f_{e_j} – $j = 0, \ldots, 7$ — effector transfer functions

f_{d_j} – $j = 0, \ldots, 8$ — program designator transfer functions

f_{v_j} – $j = 0, \ldots, 7$ — all variables transfer functions

f'_{v_j} – $j = 0, \ldots, 7$ — only c_{vv}^i variables transfer functions

f''_v – real sensor reading aggregating function

i_0 – label of the initial state

i_k – label of the state in which the initial condition is satisfied

i_m – label of the terminal state

$\ddot{\imath}$ – instruction

$I\!I$ – set of instructions

j_R – j_{R+1} is the number of reactions

r – state of real sensors

R – real sensor reading space

r^i – state of the real sensors in instant i

r_{int} – internal receptor reading

R^E_{int} – internal receptor reading error space

s – state of the system

S – system state space

s^i – state of the system in instant i

s^* – sequence of states

S^* – set of all sequences of states

s^{**} – evolution of state

S^{**} – set of all evolutions of state

sem – semantics of an instruction

v – state of virtual sensors

V – virtual sensor reading space

V^E – sub-space of virtual sensor readings satisfying the error condition

$\overline{V^E}$ – closure of V^E

v^i – state of virtual sensors in instant i

V^I – sub-space of virtual sensor readings satisfying the initial condition

V^T – sub-space of virtual sensor readings satisfying the terminal condition

V_j – virtual sensor reading sub-space associated with reaction B_j

V_0 – neutral reading sub-space

δ – search direction marker

2^Q – set of all subsets of set Q

1 Introduction

1.1 Robot software

In 1920 Karel Čapek wrote his famous drama "R.U.R" (Rossum's Universal Robots), in which he described a factory mass-producing biomechanical people. They were supposed to blindly obey humans, but eventually they rebelled, seized power, and ultimately wiped out the human race. Not only did Karel Čapek coin the term *robot* – derived from Czech words: *robota* and *robotnik*, the former meaning labour and the latter worker – but also he associated with this term anthropomorphic features. Anthropomorphism of robots has been stressed by the majority of science fiction novelists since then. That is why encyclopedic dictionaries tend to underscore humanlike features in robots, e.g. in [195] the following definitions are given:

– a machine that resembles a man and does mechanical routine tasks on command as though it were alive,
– any machine or mechanical device that operates automatically with humanlike skill.

In the early sixties the first industrial robots were produced by Unimation and Versatran. They resembled human upper limbs very vaguely, if at all. Robots created industrially or in scientific laboratories very rarely resemble the whole human body. On the contrary, only a slight similarity with some parts of the human body can be detected. Sometimes, especially in the case of walking machines [135, 156, 96], lower animal extremities (mammal or insect) are used as models. In the case of manipulators, besides human arms, the elephant's trunk and man's spine [98] have been the inspiration for designers. Moreover, the kind of locomotion the robot utilises strongly influences its shape (e.g. wall climbing [58], brachiation[1] [121]). In the scientific community there is no uniformity of views as to the exact definition of a robot. Nevertheless, the majority of scientists dealing with robots would agree that a **robot** should have:

– either a manipulative[2] or a translocative[3] capability, or both,
– the ability to change its behaviour.

The second postulate demands that there should be some external means of changing the robot's actions (e.g. motions). This can be achieved either by changing the program of these actions or, if the robot is to some extent capable of autonomous operation, by detecting through its sensors the changes in the environment and automatically undertaking modifying actions. In the first case the robot is treated as a reprogrammable device, and in the second its internal program of functioning is flexible enough to modify its behaviour according to external circumstances.

The job that a robot is designed to perform has a decisive influence on the definition. Currently the diversity of robot applications is so vast, that the number of different definitions is abundant. Robots are put to such unusual tasks that even the Robot Olympics are held. The First International Robot Olympic Games were hosted by the Turing Institute, and were held in Glasgow, United Kingdom, 27–28 September 1990. The

[1] Dynamic swinging motion that enables long-armed apes to move from branch to branch.
[2] The ability to manipulate objects in its environment.
[3] The ability to translocate itself in the environment.

following events took place: wall climbing; biped race; four-, six-, eight-legged race; obstacle avoidance, wall following, talking, pole balancing, javelin throwing; and many other strange behaviour presentations. Nevertheless, primary robot tasks are industrial, although the emergence of service robots in recent years might cause in the near future their predominance over industrial robots.

The Robot Institute of America (RIA) specifies an **industrial robot** as *a reprogrammable multi-functional manipulator designed to move materials, parts, tools, or specialised devices, through variable programmed motions for the performance of a variety of tasks* ([38]). There the robot is treated in a narrower sense[4] – rather as a reprogrammable manipulator than as a self-translocating or an autonomous device. The RIA definition underscores reprogrammability, while the above postulates either implied it or assumed a single, but intelligent, program of actions. Currently all modern robots are computer controlled, so programming is done by software means.

Robots besides being used in manufacturing industry are currently being employed in, e.g.: agriculture [124], construction industry [95], outer space [87], health care [36], and the service sector [35]. As it was mentioned above, each of these applications is reflected in the definition of an adequate robot. Fraunhofer-Institute for Manufacturing Engineering and Automation (IPA) defines a **service robot** [122] as *a free programmable system to execute movements (mostly handling tasks) with 3 or more axes of movement. Such a robot typically is movable and performs partly or fully automatically service tasks. Service tasks are tasks not used for the manufacturing of products but for the execution of services for men or installations and public utilities*. In this definition, as in all the others, again, the ability to program robots is of primary importance.

Regardless of the exact wording of the definition, it is obvious that the software component of a robot is of paramount importance. In a robot, treated as a system, usually three subsystems are distinguished: effectors, receptors and control subsystem (e.g. [93]). The control subsystem, hierarchically in the upper layer, is responsible for controlling both the effectors and receptors, and moreover for reasoning and external communications. Each of these functions is at least partially realised by software – either controlling the hardware (e.g. manipulator, sensors) or realising adequate functions (e.g. reasoning, communication with an operator). Reprogrammability can either mean supplying a new program of actions through the communication channel to an unalterable robot system or a modification of the internal robot system structure (robot software). In the first case the program of actions is usually coded in a robot programming language and delivered through the communication channel to the control subsystem for interpretation and – ultimately – execution. In the second case the internal robot software is altered directly. Once this modification is done, the robot functions in a different manner. Although both methods of reprogramming need different implementation techniques and effort, they can be equivalent in outcome.

This is a very broad view of robot programming. The following survey of robot software will show the vastness of this field of knowledge (mainly due to its interdisciplinary character). In consequence, the dissertation will focus only on robot programming treated as a method of specifying actions that the robot has to perform. Nevertheless, the survey

[4] Although this is the definition of an industrial robot, not a robot in general, the main difference is the neglect of the translocative capability.

will locate the area of interest of this monograph in relation to robot software treated as a whole.

Each of the afore-mentioned functions has its own proprietary software:

– effector control software:
  - manipulator or pedipulator dynamics control (including link drives),
  - arm or leg kinematics control (co-ordinate transformations),
  - manipulator or pedipulator trajectory generation,
  - tool (e.g. gripper) control,
  - integration of sensor data into motion control,
– sensor software:
  - sensor initialisation, termination, and control,
  - data acquisition software (reading hardware sensors),
  - data aggregation software,
– reasoning and application specific software:
  - world model (maintaining and updating),
  - action plan generation,
  - task specific software,
– operator communication software:
  - control of input/output devices,
  - robot programming language interpreter,
  - operator command interpreter.

1.1.1 Effector control software

The science of **kinematics** deals with geometry of motion regardless of the forces which cause it. The forces that are the cause of the motion and the internal forces in a system are accounted for by the science of **dynamics**. The kinematic model includes both the solution of direct[5] and inverse[6] kinematics problems. Dynamics considerations are also partitioned into the direct problem[7] and inverse problem[8]. Both kinematics and dynamics models of robots are highly nonlinear. The former is a system of algebraic and the latter of differential equations.

The closed form solution of direct kinematics problem for the serial kinematic chains always exists [24, 110]. The inverse kinematics solution for these chains exists in a closed form only for some kinematic configurations of manipulator arms [24, 110]. For the parallel manipulators (Stuart platforms) the reverse is true [61, 103]. If the closed form solution is not found, an iterative numerical method has to be applied. Closed form solutions are superior, as they are much less time demanding, and so better suited to control purposes.

[5] Calculation of the position and orientation of the end-effector in relation to a global reference frame when the joint angles (for revolute kinematic pairs) and positions (for prismatic kinematic pairs) are given.

[6] Calculation of the joint angles and positions when position and orientation of the end-effector is known.

[7] When the generalised forces driving the joints are known, generalised co-ordinates, velocities and accelerations are computed.

[8] When the joint generalised co-ordinates, velocities and accelerations are known, joint generalised forces are computed.

There are several methods of formulating robot arm dynamics: Lagrange-Euler, recursive Lagrange-Euler, Newton-Euler, and d'Alambert principle [80, 72, 74, 75]. Depending on the approach followed, the computational efficiency of the obtained model is different. As the robot arm dynamics equations are highly nonlinear coupled differential equations, for the purpose of control the obtained model usually has to be simplified. The simplifications are also introduced while developing the model. If the arm motions are slow or the accuracy of trajectory following can be low, PID[9] controllers for each joint will suffice. New control laws and structures are the subject of on-going research. Robot arms are not only position-controlled but also force-controlled (e.g. compliance, exerting forces and torques) [147, 50]. Besides the manipulator dynamics, the actuator dynamics has to be taken into consideration too.

The motion of the robot is defined as an evolution of positions in time – a trajectory. Trajectories can be expressed in different spaces[10]. The most common are Cartesian and joint spaces. The trajectory of an end-effector is usually specified as a sequence of co--ordinate frames in space that the frame affixed to the end-effector has to follow. These co-ordinate frames are specified in relation to the global reference frame by: the X, Y, Z co-ordinates of their origins and a suitable set of angles (e.g. Euler angles; roll, pitch and yaw angles) describing their orientations. Other methods of specification are also used (e.g. homogeneous transforms [28, 110], quaternions [57, 39]). Regardless of the space that the trajectory is expressed in, different methods of interpolating between the specified positions are employed (e.g. linear interpolation, splines). Also, different velocity profiles along the trajectory have to be attained. Not all trajectories are executable: some of them may violate the work space boundary, some pass through manipulator singularities[11], yet for some others the actuators cannot develop enough power to overcome dynamical interactions. The obstacles that are in the work space have to be avoided too. The trajectory generator should take into account all of the above facts.

The trajectory generator produces consecutive trajectory positions. If the positions are not expressed in joint space, then the inverse kinematics problem is solved, and the resulting joint angles are in turn transformed into drive increments, which are the set points for the drive servos. The servo can incorporate the dynamics model in the feedback loop or the dynamics model can be outside the feedback loop, depending on the assumed structure of the controller. The degree of simplification of the employed dynamics model also varies considerably with the method of implementation of the controller (analog or digital). In the case of digital implementation the computational power of the processor used is of fundamental importance.

The software associated with kinematics and dynamics of the robot usually remains unaltered by the user of the system, unless a more efficient control algorithm has to be employed. There is no provision for that in industrial robots, but investigative setups sometimes possess this feature. The ability to change the method of interpolation between the intermediate positions of the trajectory is much more useful, but only rarely in industrial robots can the user choose between more than a few, let alone devise their

[9] The letters stand for: Proportional-Integral-Derivative.

[10] In mathematics the term *space* is defined as a set with a metric, but in robotics the term *space* is usually associated with the robot work space, i.e. the set of attainable positions and orientations, and rarely a metric is defined for this set.

[11] In these positions the manipulator loses a degree of freedom.

own methods of interpolation. As research-oriented robot controllers are dealt with in this dissertation, effector control software must be mentioned.

1.1.2 Sensor software

Sensor software is responsible for the proper initialisation and termination of sensor hardware operation, reading sensor data and transforming it into a form useful for current or future robot motion control and decision making. The utilisation of sensors allows a robot to control its motions and to interact with the environment in a flexible way. Robot sensors are divided into two groups: internal sensors and external sensors (e.g. [38]). **Internal sensors** are used for low-level actuator control (e.g. joint position encoders, joint velocity sensors) – sensing components of joint servo systems. They sense the internal state of a robot. The software dealing with internal sensors is usually a part of effector control software. **External sensors** are used for detecting the state of the environment external to the robot itself.

Usually **sensors**[12] are divided into categories by looking at the living creature senses their operation resembles. Most of the human senses have been mimicked in robots: vision [38, 4, 25, 133, 132], touch [118, 26, 19, 115, 152] (including heat sensing [26, 19]), smell [29], and hearing [46, 150]. Animal senses, that are not present in humans, are employed in robots too (e.g. ultrasonic sonar systems[13], infrared proximity sensors[14], range-finders [148, 59]). Another classification takes into account the distance from which the measurements are taken. In this case contact and non-contact sensors are distinguished. **Contact sensors** need to come into contact with an object to detect it (e.g. touch and force/torque sensors [38, 59]). **Non-contact sensors** acquire the information about the state of the environment by remote measurements. These sensors are further subdivided into **long-distance sensors** (e.g. cameras, range finders, ultrasonic sensors) and **short-distance sensors** (e.g. infrared proximity sensors [21], visual light proximity and colour sensors [97]).

Each of the afore-mentioned groups of sensors has its proprietary software for taking measurements and utilising the obtained results. Especially the second function varies with the task that the robot has to perform. This is why, while programming a robot, there should be means for interference with how the sensor data is collected (e.g. sample time) and how it is utilised (used for motion modification or generation). There should be a provision for the trajectory generator to take into account in real-time the data obtained from sensors.

The sensors that the system uses may vary with the task it has to execute. It is difficult to foresee in advance what sensors will have to be used by the system, as the tasks that will have to be executed are not known at the moment when the robot is produced. Usually, addition of new hardware sensors to the system brings about either a modification of the existing or an addition of entirely new software to the system. That is why sensor software causes so many problems in robot programming. Robot programming means must be such that the incorporation of new sensors into the system will allow an easy access to and utilisation of the data obtained through them. Sensors in the aspect of robot programming are dealt with comprehensively in this monograph.

[12] External sensors.
[13] Bats, dolphins.
[14] Snakes.

1.1.3 Reasoning and application software

If a robot is to function autonomously it should have a considerable knowledge of its surroundings. As the environment is not static, the changes have to be detected by sensors, and the world model has to be updated accordingly. If a discrepancy arises between the state of the environment and the world model, inadequate decisions will be made by the reasoning subsystem. The application software is highly dependent on the class of tasks the robot has to execute (e.g. assembly, table tennis playing). The action plan for the execution of a task can either be delivered by a user in the form of a program or it can be generated by a plan generator. In the latter case artificial intelligence methods are used to code such a generator. Even if the robot reasoning subsystem is incapable of generating the action plan, the application software has to include huge amounts of data concerning the realised complex task. In complex tasks very diverse situations may arise, and they have to be handled appropriately, so the software has to include code reacting to each of such situations. This software directly interacts with the trajectory generator – translating task actions into robot motions.

As reasoning and application software is task specific, and the tasks that a robot can be put to vary immensely, only some aspects of robot programming regarding this issue are dealt with in this work. The concept of sensor-based reactive robot control [184, 185, 188] are at the focus of interest.

1.1.4 Operator communication software

A robot has to obey a human operator who supplies it with either a detailed program, that it has to execute, or a general description of the task at hand. In both cases the operator communication software must input the description of the task, interpret it, and command its execution. Moreover, while the task is being executed the operator may want to abort or suspend its execution. This also is the job of the communication software. On the lower level this software has to control the input/output devices that enable the robot-operator communication. This kind of software is central to the implementation of robot programming means. This is the reason why this subject is mentioned in this dissertation.

1.2 The aim and subject of the dissertation

As different aspects of robot programming have been described in hundreds of papers, not necessarily primarily devoted to this subject, the aim of this dissertation is to bring this dispersed knowledge into one place. There exists a very good textbook [12] on robot programming languages. It briefly surveys programming as perceived by computer science (i.e. data types, procedures, functions, execution control flow, etc.) and points out extensions to computer programming that have to be introduced due to specificity of controlling robots. Several existing robot programming languages are described and compared. The methods of implementing these languages and structures of the systems they run on are not dealt with. Moreover, the limitations and extensibility of these languages and systems are not discussed.

Most books devoted to robotics (e.g. [5, 24, 25, 38, 72, 99, 110, 154]) are written by authors having mechanical engineering or control sciences background, so although

kinematics and dynamics models, position and force control, and trajectory generation are treated comprehensively, they cover the subject of robot programming superficially, if at all. Nevertheless, most of the knowledge regarding robot programming is distributed over conference reports and journal papers. The majority of them describe new robot programming languages (RPLs) (e.g. AL/POINTY [100, 101, 10, 41], AML [134, 51, 86], AUTOPASS [83, 13], HELP [51, 25, 12], LM [78, 86], MCL [149, 25, 13, 51], PAL [25, 88], PASRO [11, 12], RAPT [2, 112, 113, 153], RCCL/KALI [53, 84, 6, 136, 54, 55], SRL [12], VAL/VAL II [193, 194], WAVE [109, 25, 86], and tens of others). This multitude of RPLs caused the emergence of survey works (e.g. [12, 164, 167, 157, 158, 25, 86, 51, 13]), trying to classify the languages and describe their intrinsic properties.

The papers concerned with RPLs usually describe, using natural language, the robot specific instructions, the hardware that runs the executable code obtained by compiling or interpreting the language, and the programming environment (e.g. operator interface, editing facilities, program execution simulation), and are very moderate on the subject of implementation. Moreover, no theoretical background is given as to the choice of the instruction set. As it has been mentioned, the instruction semantics is stated in natural language, rather than formally. For languages submerged in universal computer languages, that universal language is the means of expressing robot specific semantics. Some insight into RPL semantics is given in [42, 164].

Quite a considerable robot programming effort has been associated with motion trajectory generation, task planning and world modelling (e.g. [110, 24, 57, 40]). Robot action and motion planning is inherently associated with robot programming, so a considerable research effort has been put into solution of these problems (e.g. [64, 65, 62, 63, 68, 117]). In [64, 62, 63, 117, 65] automatic synthesis of programs for robots operating in a flexible manufacturing cell is presented. Program generation takes place in two stages. In the first stage a work-plan is generated in a robot-independent manner. In the second stage this plan is transformed into sequences of instructions causing the execution of adequate robot motion trajectories. Finally, the execution of the program is simulated to select the most effective variant of its realisation.

Although some work has been done on diverse computing architectures for single– and multi-robot systems (e.g. [120, 106, 107, 20, 67, 136, 144, 104, 45, 9]), work on software architectures of upper levels of robot system controllers is rather scarce. In this field the best known is KALI [54, 55, 6, 105, 136], and Brooks' behavioural approach to robot control [15, 16, 17]. Although a lot of work has been done on specific sensors (e.g. proximity [21, 89, 97], tactile [118, 26, 115], force/torque [147, 50], ultrasonic [151] or vision [38, 4, 25, 133]) and the incorporation of single or groups of homogeneous simple sensors into a robotic system, much less effort has been concentrated on open control architectures integrating many diverse sensors. Quite considerable knowledge regarding a utilisation of diverse sensors can be gained by studying the literature concerning mobile robots (Automatically Guided Vehicles) (e.g. [151]).

The issue of robot programming is inseparable from the problems of robot controller design. Many research teams treat both problems jointly. Industrial robot controllers have a closed structure and are well suited to simple industrial tasks. Complex, often changing tasks, especially those requiring the incorporation of many diverse sensors, need controllers with open structures. Both hardware and software aspects of open robot

controllers are investigated by many university and industrial research teams around the world (e.g. [53, 136, 6, 23]). Moreover, several universities in Poland, besides Warsaw University of Technology [181, 182, 187, 178, 179, 169], have undertaken the effort of building robot controllers for research purposes (e.g. Technical University of Poznań [33, 32, 34, 145], Technical University of Rzeszów [81, 82, 140] and Silesian University of Technology [141]). Traditionally the servo-control level has received much attention. Both position and force control is of utmost importance (e.g. [88, 44, 47, 48, 49, 50, 126, 127, 108, 77]). Technical University of Wrocław deals with upper layers of robot control, i.e. work-plan generation, work-scheduling and trajectory generation [64, 65, 62, 63, 68, 117].

Most robot programming systems have been constructed as certain extensions to existing computer programming systems, without trying to establish a theory similar to computer science theory (especially regarding semantics). The aim of this monograph is to describe different robot programming methods, not only from the point of view of the programmer, but also from the point of view of the one who has to devise the structure of the system and to implement it. This work tries to give some insight into the theory that should underlie robot programming. Last but not least, the aim of this dissertation is to show how the science of robot programming evolved. Author's own experience gained by defining and implementing different robot programming and control systems led to a certain generalisation and formalisation which is described in this work.

To be brief: **the subject of this monograph is the programming of robots as perceived by a user and implementer of robot systems**. As it was pointed out above, the scope of such a dissertation can be vast, depending on the degree of interference with the functioning of the system its operator has to possess. Moreover, due to a high variety of tasks that the considered system can be put to, it is not defined precisely – it is an open system. Nevertheless, this work presents a method of describing such systems. This method was instrumental in defining a general structure, as well as programming means for such systems.

This dissertation reflects a certain line of thought which underlies the evolution of robot programming. Initially considerable effort was put into devising new robot programming languages [157, 158, 163, 164, 167, 176, 160]. Later specialised languages gave way to robot specific libraries of functions coded in universal programming languages [11, 53, 23, 84, 54, 6, 105, 178, 179, 181]. This was due to the limitations of the former, especially in the research environment.

The author's own research is presented with the achievements of other scientists in the background. Initially the author defined and implemented on two different systems an object level robot programming language TORBOL [162, 164, 168, 173, 176, 165, 166]. This work pointed out the limitations of specialised robot programming languages, regardless of their level. The main drawback of such systems is their lack of flexibility when the class of tasks that they were initially designed for changes considerably – and this is usually the case when new research is undertaken. Nevertheless, TORBOL proved that there is a method of devising languages operating on notions comprehensible to humans, and yet that can be translated into robot motions without using artificial intelligence techniques. When the class of tasks that have to be executed by the robot system does not vary considerably this approach can be followed.

The degree of robot autonomy depends on the possibility of acquiring diverse information from the environment and executing a decision making process. As the tasks that the robot will be put to are *a priori* unknown, it is difficult and uneconomic to define a closed structure system that will execute all possible jobs, hence open structure systems are investigated. It is much easier to implement extensible libraries of control functions than to define either an extensible specialised language or a very universal specialised language taking into account all possibilities. Due to that the author defined several such libraries: ROPAS [177, 183], ROOPL [177, 180] and the Research-Oriented Robot Controller library RORC [178, 179, 175, 181, 187].

While working on robot programming languages the author has come up with a classification of their instructions and a method of their formal specification [159, 161, 164, 170, 174]. The formalism is based on the decomposition of a robot system into parts. It is an extension of operational semantics [27] defined for computer programming languages. Influence of instruction execution on the state of each of the system components is considered. Depending on the parts of the system that are influenced by the instruction, it is assigned to a certain class. Moreover the method of utilising sensors during the execution of an instruction is dealt with herein. The completeness of the language is checked by looking at the possibility of controlling through instructions each part of the system. Different types of instruction semantics are considered. Languages with few but complex instructions are compared with languages with large sets of instructions but simple semantics. As usual, an intermediate solution looks most promising.

Implementation issues are at the focus of this dissertation. As robot specific libraries of functions coded in a universal programming language are preferred, both the base languages (e.g. Pascal, C) as well as the programming methodology are discussed (e.g. object-oriented or structured programming). The incorporation of diverse sensors into a robotic system is treated comprehensively. Both the sensor data fusion and the environment data utilisation in control are dealt with.

Different system structures are described [169, 181, 188]. These structures are implied by the formalism devised by the author for specifying instruction semantics. A single-robot structure is enhanced to a multi-robot one [189, 186]. The same formalism was used by the author to specify sensor-based reactive robot control, in which a robot reaction is associated with each sensor reading sub-space [188, 185, 184]. During execution of the primary task the sensors are constantly monitored. Whenever the sensor readings enter a specific sub-space, an adequate reaction is activated to avoid an undesirable event. When a corrective action is successively accomplished, the primary task is resumed. If not, other reactions are invoked.

2 Model of a robot system

At the focus of attention of this dissertation is a robot system as it is perceived by a programmer or an implementer of the human interface with such a system. This model has to take into account the fact that a robot system actively influences its environment, gathers data from this environment, and changes its own internal state.

2.1 Decomposition of a robot system

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

$$s = < e, r, c >, \quad s \in S, \quad e \in E, \quad r \in R, \quad c \in C, \tag{2.1}$$

where:

s	– the state of the system,	S	– the system state space,
e	– the state of the effectors,	E	– the effector state space,
r	– the state of the real sensors,	R	– the real sensor reading space,
c	– the control subsystem state,	C	– its state space.

The values of s, e, r, c are usually expressed as vectors.

The raw data obtained from real sensors usually cannot be utilized directly to control the system. It has to be transformed into a useful form. This transformation is called **data aggregation**. As a result of this a **virtual sensor reading** v is obtained:

$$v = f_v''(r), \quad v \in V \tag{2.2}$$

Vector function f_v'' is called an **aggregating function**. V is the virtual sensor reading space.

2.2 Description of the state of a robot system

The description of the state of a robot system s consists of the descriptions of the state of each of its three parts.

The description of the state of receptors r is strongly influenced by the type of real sensors connected to the system. The readings of these sensors vary from simple binary values to complex multidimensional bit maps. At the most complex end multidimensional arrays of real values (e.g. matrices) are the most appropriate abstract notions describing the state of real sensors. Real vectors and Boolean values are much more common in industrial systems.

The state of the effectors e is expressed in terms of: the state of the joints of the manipulator – **joint specification**, state of the end-effector e.g., tool mounted on the manipulator – **tool specification**, or the state of the objects of the work space – **object specification**. The motivation for treating objects that are in the work space as effectors is the following. If the robot is equipped with a gripper, it can pick an object and use it as a tool in its own right, so this object has to be treated as an effector. If all the objects that can be transferred would not be included in the description of the state of the effectors, the description would vary whenever the robot picks a new object. That would highly complicate this description. Obviously, not all objects can be used by the robot as tools, so not all real objects need to be included in the description. As the transferred objects can be placed on immobile objects, the latter also can be included in the

specification. In the case of inclusion of some of the static objects, the objects that remain immobile simply do not change their state, but the description remains unaltered. Another reason for treating all objects from the work space as effectors is that devices cooperating with the robot are also objects out of its work space, and those usually can influence the environment actively, as effectors can. Which objects are included in the description as effectors and which are excluded from this specification to a certain extent is a matter of choice. All the objects that are transferred by the robot directly or indirectly must be treated as effectors. Others, depending on their role in the executed task, can either be treated as external agents, and so their existence is detected by sensors causing changes of r only, or can be treated as effectors and so their state change would directly, or indirectly through r, be reflected in e.

In the case of joint specification of effector state, usually the generalized position and velocity of each joint are used. In the case of tool specification, position is usually expressed in terms of Cartesian co-ordinates and orientation in terms of a set of three adequately chosen angles (e.g. Euler angles [110]) or a rotation angle and axis of rotation [24]. Homogeneous co-ordinates, although redundant, are commonly used too [28, 110, 24, 25]. Velocity is expressed as an adequate time derivative of position and orientation. Quaternion description is rarely used [39, 57]. Moreover, the torque or force exerted by the end-effector on the environment can be included.

Object specification of effector state is rather complex, because not only must the position and orientation of each object be included, but also relations between these objects are important. Furthermore, objects are characterized by many properties which have to be reflected by the description of their state. A rather complicated solution to this problem is the attribute graph presented in [170, 174]. Inclusion of objects other than the robot into the system caused the system to be also called the virtual environment.

The description of the state of the control subsystem c must include the description of the state of all relevant variables (that will include data bases) and the description of the state of execution of the control program (program itself and the pointer to the currently executed instruction and the next one that is to be executed):

$$c = < c_p, c_d, c_v >, \tag{2.3}$$

where:

c_p – the state of the part of the control subsystem containing the executed program – **control program state**,

c_d – the **state of instruction designator**,

c_v – the **state of variables**.

The control program state remains unaltered during the execution of the program. Systems with self-modifying programs are not considered here, as they are very difficult to manage and debug. Because of this, self-modifying programs are considered by computer scientists to be a bad programming practice. Only c_d and c_v change during the execution of a program.

2.3 Instructions and the description of their semantics

Once the decision has been made as to how the state of a robotic system will be described, another important decision has to be made. This decision concerns: how, under the influence of a program, the change of the system state in time will be described. The system changes its state under the influence of the program that is executed by its control subsystem and any disturbance external to the system. This disturbance is caused by external agents changing the state of the system (including effectors). The detection of these state changes is only possible through receptors or by explicitly informing the system about the anticipated state change by including in the program an instruction containing the information about the modification. In both cases an adequate instruction has to be issued to the system. Although the robot itself is not the cause of the effector state change (an external agent causes the change), in both cases the instruction will be treated as the cause of this change – for reasons of uniformity. For example, if a robot is force-controlled and the tool should be compliant in all directions, then pushing the tool with a finger (external agent) will cause the arm to retreat from the applied pressure (i.e. change the effector state). Obviously, for the robot to do this, it has to be instructed to be compliant. The same is even more evident when an object that is treated as an effector is transferred by an external agent (e.g. conveyor).

The program consists of instructions. Execution of each program instruction causes a certain change in the state of the system. It is important to know what influence an instruction will exert on the state of the system, i.e. what its semantics will be.

An obvious solution to the above-stated problem is to associate with an instruction a state transition pair: *initial state, terminal state*. Let $\ddot{u}$ be an instruction and $\ddot{I}$ the set of instructions, $\ddot{u} \in \ddot{I}$. Semantics of an instruction is defined as the mapping *sem*:

$$sem : \ddot{I} \rightarrow [S \rightarrow S] \tag{2.4}$$

where $[S \rightarrow S]$ is a set of operators that is assigned to the set of instructions $\ddot{I}$. This is the way the operational semantics of computer instructions is stated. Unfortunately, in the case of a robot system this is an oversimplification, because it is important to know how the system will behave in between the initial and the terminal states (e.g. not only are the end-points of a trajectory important, but also the path that the manipulator follows, because it can collide with obstacles). Another solution to this problem is to determine the initial state and the continuum of states that follow while the instruction is being executed.

A **sequence of elements of a set** (e.g. of set A) will be denoted by a^*, and the set of such sequences by A^* .

Definition 1

If T is a linearly ordered continuous set and T' is its 1–connected subset and there exists a function $f : T \rightarrow A$, $t \in T, a \in A, a = f(t)$, then $f(T')$ is denoted by a^{**} and called the **evolution** of the elements of set A.

The set of all the evolutions of a will be denoted by A^{**}, $a^{**} \in A^{**}$. Hereafter, T' will be the time interval.

Now the mapping *sem* is defined as:

$$sem : \ddot{I} \rightarrow [S \rightarrow S^{**}] \tag{2.5}$$

Usually, complete knowledge of the evolution of state that results from the execution of an instruction is unnecessary, and moreover the description becomes too complicated. An intermediate solution seems most appropriate.

The quantity upon which bounds were imposed is called the **transition parameter** $tp \in TP$. The set of all bounds imposed on the transition parameters is denoted by Q. The set of all subsets of Q is denoted as 2^Q, so the mapping *sem* is defined as:

$$sem : \Pi \rightarrow [S \rightarrow 2^Q \times S] \tag{2.6}$$

The bounds can be imposed on any component of the state of the system. This notation specifies both the initial and the terminal states precisely, but it only imposes constraints on the continuum of intermediate states. Possibly many evolutions of state can satisfy these constraints. The most useful form of this specification of system state transition is the one in which some intermediate states are specified explicitly. In this case the bounds are expressed in terms of a finite number of discrete time states of the system. In such a case the mapping *sem* is defined as:

$$sem : \Pi \rightarrow [S \rightarrow S^*] \tag{2.7}$$

This definition can also be viewed as a discrete form of (2.5). This specification of instruction semantics has one more advantage. Most robot systems are computer-controlled, and so the set values for the joint controllers are updated at discrete instants. In this way the specification is closely related to the method of implementation. This definition of RPL instruction semantics is favoured in this dissertation.

3 Robot programming

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

3.1 On-line programming

On-line programming is based on teaching a robot the trajectories it has to follow. **Teaching** is done by leading the robot arm through a sequence of motions and recording these motions, so that later they can be replayed automatically. The arm during teaching can be propelled either manually (i.e. the operator uses his muscles to shift the arm from one position to the other) or by its drives (i.e. the operator uses the joystick, the keyboard of the teach pendant, or a scaled down replica of the manipulator to command the drives to appropriate positions). Regardless of the way of propelling the arm during teaching, there are two ways of recording the trajectory of the arm motion. In the **PTP**

(Point To Point[1]) method the arm is transferred to each characteristic point of the trajectory, stopped there, and by pressing a special button on the control panel, this position is memorized by the control system. During playback the robot arm goes through these points using some form of interpolation between them. In the **CP** (Continuous Path) method, as the arm is transferred, the positions are being recorded automatically at constant intervals of time. As the points are very near each other no special interpolation routines are necessary. Moreover, the motions can be played back with different speeds by changing the time base (the interval of time allowed for reaching the next point).

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

3.2 Off-line programming

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

To make a robot more productive, the phase in which it is required for programming has to be as short as possible. In other words, robot programming has to be made independent of the robot. The program is developed off-line and later only loaded to the control system for execution. The problem with this approach is that although currently manufactured robots feature high repeatability, they exhibit low accuracy. This necessitates the calibration of the program created off-line. As the solution to this is an open research problem [60], the industrial robots used currently cannot be programmed strictly off-line. Nevertheless, RPLs draw quite a lot of attention. The solution to the calibration problem is one cause of this and the other is the simplification of coding programs in these languages. In addtion, only RPLs fully solve the problem of sensor integration.

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

The main instructions of RPLs are the ones causing the motion of the effectors, i.e. **motion instructions**. The abstract notions that these instructions refer to are: the ma-

[1] A more adequate name, although rarely used, is: *Pose To Pose.*, where *pose*, unlike *point* which in mathematics specifies only the position, includes both position and orientation. In this work *pose* is equivalent to *location*.

nipulator joints, the end-effector or the objects of the work space. These notions were used in subsection 2.2 to express the state of the effectors[2].

Each of the enumerated notions creates a certain **virtual environment** [161, 170, 164], in which the instructions of the language operate. In other words, those elements which had been considered important were selected from the real environment to constitute the virtual environment. Only some elements of the real environment (including the robot) are the basis for creating abstract notions that compose the virtual environment. The virtual environment is a simplified model of the real environment. On the degree of this simplification and on the abstract notions that were chosen to make up the virtual environment depends the complexity of the control system of a robot. The programmer through each level of the control structure perceives the real environment as a simplified model – he perceives the virtual environment. In the case of RPLs the above-mentioned abstract concepts are hierarchically related and so these languages can be classified into levels [43]. It should be noted that the virtual environment is a certain abstraction of the robot system.

The languages of the lowest level are called **joint level languages**. The instructions of those languages cause the generation of sequences of signals controlling the drives of the manipulator. So in this case the manipulator joints form the virtual environment. The design of a control system accepting these instructions is quite routine, but to forecast how the tool will behave when all the drives are in motion is not as simple. For simplification of the design we have to pay the price of the programming complexity.

The languages of the next level free the users from this disadvantage. The main concept of the virtual environment of this level is the manipulator's end-effector, so these languages are called the **manipulator level languages**. Although it is easy to predict the trajectory of the robot tool when using languages of this level, the programmer still has to be concerned with the description of all the motions of the manipulator instead of simply stating what actions have to be performed to accomplish the task.

The instructions of **object level languages** operate in virtual environments composed of models of objects existing in the work space. The programmer states only which objects should be transferred, so that the task will be accomplished. The robot control system, using its knowledge of the objects and the relations between them, will relocate the manipulator in such a way so as to complete the job. From this level onward the programmer does not have to busy himself with the motions of the robot arm, but can concentrate on the operations that have to be executed.

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

Till now over a hundred robot programming languages have been implemented. A comprehensive survey of RPLs can be found in [164, 12, 51, 167, 163, 157, 158, 13, 79, 125]. Unfortunately, most of them have only single-site implementations, and their manuals are unavailable. Only some out of this multitude will be described briefly

[2] The virtual environment is a model of a robot system as perceived by the programmer through the programming language he uses.

in the following sections (either better known or more characteristic ones have been chosen).

3.2.1 Joint and manipulator level robot programming languages

In the case of joint level robot programming languages, the motions of the robot are expressed in terms of relative positions (translation for prismatic joints and angle of rotation for revolute joints) of consecutive links of the manipulator. Manipulators are multi-degree of freedom devices, so it is extremely difficult to judge where the end-effector will be located when an adequate set of joint angles and displacements is given (e.g. for articulated robots). As joint level languages are very difficult to use, there are not many examples of them. Currently such languages are mainly used by Cartesian robots[3]. On the other hand, languages for Cartesian robots can also be treated as manipulator level languages, as the joint co-ordinates are equivalent to end-effector co-ordinates.

As examples of manipulator level robot programming languages two have been selected: WAVE and VAL II; the former – because it was one of the first well-developed manipulator level languages and had a great impact on quite a few languages defined later (e.g. VAL, AL), and the latter – because it is widely used with PUMA industrial robots.

WAVE

WAVE [109, 163] was developed in the early 1970s at Stanford Artificial Intelligence Laboratory for research purposes. It operates on three data types: co-ordinate frames (TRANS), vectors (VECT) and loop counters. Arithmetic on those types is very limited.

There are special instructions for incrementing and decrementing loop counters. They are used in conjunction with the JUMP instruction to construct program loops. Moreover, SKIPE and SKIPN skip (or not) to the next instruction if the error number argument is equal to the program error state number – those instructions can be used for conditional execution.

OPEN and CLOSE instructions deal with the gripper. MOVE, CHANGE and CENTER instructions move the arm. The MOVE instruction transfers the end-effector to the goal position through two intermediate points using cubic spline interpolation in joint co-ordinates. The CHANGE instruction is used for incremental motion of the arm. The CENTER instruction causes closing of the gripper. When one of the fingers makes contact with the object, the position of the arm is modified in such a way that eventually the gripper is fully closed without shifting the object. Monitoring of exerted forces during motion is possible, if prior to the motion instruction the STOP instruction is issued. Straight line and circular motion is possible by using the DRAW instruction, which performs functionally defined motion through a series of positions.

FREE and SPIN are used to select the direction and axis of rotation respectively for which the arm is compliant. Out of this data the control system will select the joints which will be force-controlled and those which will be position-controlled. FORCE causes the exertion of a specified force or torque. WOBBLE is an instruction causing small sinusoidal motions to be superimposed on the main motion.

[3] In this case the joint space is equivalent to end-effector space.

VAL II

VAL II [194, 14] evolved from VAL [193]. VAL was developed in 1975. In 1978 VAL had been rewritten as VAL II, and was offered commercially with PUMA robots. VAL II is a BASIC like interpreted language, so it is composed of a mix of instructions necessary for robot control and monitor commands for creating and manipulating robot programs (e.g. editing, loading, storing, deleting, printing, listing, executing, aborting, debugging). Only robot control instructions are within the scope of this dissertation.

VAL II, besides integer and real arithmetic, has geometric data types necessary in robot programming (e.g. co-ordinate frames – named **locations**; and joint co-ordinates – named **precision points**). All of its variables are global.

It has goto, if then else, case, for, while, repeat and call, return program execution control constructs. Moreover, the program can be made to wait for an operator reaction (PAUSE), stopped (STOP, HALT), delayed by a certain time (DELAY) or forced to wait for an external event (WAIT). VAL II enables asynchronous processing of process control programs. REACT, REACTI, REACTE instructions cause constant monitoring of external input signals or errors, and if a certain event occurs and its priority is high enough, they invoke a subprogram handling it.

MOVE is the basic motion instruction. It causes the robot arm to move to a specified location or a precision point. Joint (MOVE) or Cartesian (MOVES) interpolation can be used during motion. If during motion the gripper has to be opened or closed, MOVET or MOVEST with an adequate argument are used. Intermediate locations can be specified too (e.g. APRO, APROS, DEPART, DEPARTS). Either waiting for the termination of motion or simultaneous execution of the following non-motion instructions can be caused. Speed of motion can also be determined (SPEED). Arm configuration control is also possible (e.g. RIGHTY, LEFTY, ABOVE, BELOW, FLIP, NOFLIP). The accuracy of attaining path segment transition points can also be controlled (COARSE, FINE, NONULL, NULL, INTOFF, INTON). A single joint can be moved by the DRIVE instruction.

The RS–232C interface can be accessed from VAL II by reading/writing to a suitable port address. There are special instructions for reading binary inputs and A/D converters, to which sensors can be connected.

Because VAL II is implemented on an LSI–11 based microcomputer, its computational capabilities are rather limited. Most of its computational power is used up by inverse kinematics and arm control. Moreover, VAL II is an interpreted language. This leaves very little computational power for sensor data processing.

3.2.2 Object level robot programming languages

Only three object level RPLs are chosen for the presentation in this dissertation. For the same reason only some features of these languages will be described. From the point of view of robotics three features of these languages seem most important. These features are:

– description of motion (i.e. motion instructions),
– operation of the data base holding the state of the virtual environment (i.e. world model instructions),
– program structure.

The three languages are: AL (Assembly Language) [10, 12, 100, 101] developed at Stanford University, RAPT (Robot Automatically Programmed Tool) [1, 2, 112, 113] implemented at the University of Edinburgh and TORBOL (Transformation Of Relations Between Objects Language) [162, 168, 173] elaborated at Warsaw University of Technology (later also implemented at Loughborough University of Technology). Each of these languages has been chosen for its distinct approach to object level robot programming.

AL

AL has a Pascal-like syntax. Its principal data types are: SCALAR, VECTOR, ROTATION, FRAME and TRANSFORM. Each variable, besides its name and type, has a unit associated with it (e.g. m/s for velocity). This enhances the possibility of error checking. A notion of object does not exist in this language. Usually, FRAMEs are given names that make us associate them with objects or their distinct parts. When two objects are bound together or two frames describe two parts of the same object, the system is instructed by an AFFIX command that this is the case. There are two kinds of this command distinguished by an extra keyword: RIGIDLY or NONRIGIDLY. Rigid affixture is symmetric, that is if two objects (frames) are affixed rigidly, no matter which of them is transferred the other will follow its movement. In the case of the non-rigid affixture the motion of one of the frames brings about the motion of the other but not vice versa. Two frames can be disjoined by an UNFIX command. From the point in the program where the system encounters this instruction, it subsequently treats the two frames as separate. The AFFIX and UNFIX instructions are the only means of specifying relationships occurring in the virtual environment. The values of variables, including frames, can be changed by an assignment statement. In this case the programmer can exert totally unlimited influence over the world model – including the introduction of discrepancies between the state of the world model and the real environment, so utmost care has to be taken.

The basic motion statement has the following form:
MOVE <controllable frame> TO <destination> [<modifying clauses>];
where the angle brackets contain the non-terminal symbols and the square brackets enclose optional parts of the instruction. The <controllable frame> is the name of the robot arm or the name of the frame (object) affixed to the arm directly or indirectly. The <destination> is a frame expressing the goal position for the <controllable frame>. As the current position of the manipulator specifies the initial position and the <destination> specifies the goal position of the motion, when we want to influence the intermediate motion, we use the <modifying clauses>. They determine how the motion is to be performed. The <modifying clauses> can specify intermediate points on the trajectory of motion (VIA) or approach to the goal position (APPROACH) or departure point in the vicinity of the initial location (DEPARTURE). Moreover, the velocity of motion can be defined (VELOCITY) or its duration (DURATION). One aspect of motion is transferring objects. The other is exerting forces. AL enables the specification of force along a vector (FORCE) or a torque about a vector (TORQUE) to be exerted during the motion. Force components must be orthogonal to avoid incompatible requests. Applying force of magnitude zero causes the arm to avoid exerting force in the specified direction (i.e. causing compliance). If the motion is to be performed with zigzag imposed on the main trajectory, a WOBBLE clause can be used (this kind of motion is

necessary in the case of welding). As the AL system evolved, many other clauses have been introduced into it.

RAPT

RAPT has a somewhat obsolete syntax due to the fact that it evolved from a language for programming numerically controlled tools (APT). The structure of the program is composed of the following elements:

– body definition statements which use:
  - construction features (e.g. POINT, LINE, CIRCLE) to define:
  - external features of bodies (e.g. FACE, SHAFT, HOLE, EDGE, VERTEX, SPHFACE, i.e. spherical face),
– relational statements describing the relationships between external features of the body (e.g. AGAINST, COPLANAR, FITS, ALIGNED, TIED, UNTIED, ISSUB, i.e. is a subassembly, NOTSUB),
– motion statements (e.g. MOVE, TURN) causing the motion of AGENTs (e.g. robot),
– situation description statements which use the same keywords and syntax as the relational statements.

First, all the bodies (objects) existing in the virtual environment have to be defined by using body definition statements and relational statements. Each body is a 3-D object with a name, a set of features and a position associated with it. Once this is done the program executing the task can be written. It is composed of descriptions of situations which must come into existence during the execution of the task (e.g. assembly). The situations can be treated as snapshots of important phases of the task. Situations are expressed in terms of spatial relationships between objects. Between each situation must come a movement. Motion is caused by agents. Only agents (e.g. robot) or bodies connected to agents can move. The connections between bodies and the agents are specified by TIED/UNTIED and ISSUB/NOTSUB statements. Ties are strong links between bodies, so no relative motion can occur between them. Subassemblies are weaker links between bodies, so they permit constrained relative motion. By using these statements the robot and its tools can also be defined. Subassemblies can be permanent or temporary.

The RAPT system possesses its rules of motion. It knows that any body TIED to the world cannot move and that any agent or body TIED (directly or indirectly) to an agent can move. Moreover, any body which is related to a subassembly of a body which can move can also move. The system starts inferring the intermediate positions in a path from a known position. It can either be the initial or the goal situation. Forward or backward inference is used (artificial intelligence methods are employed). The world model is constructed by using body definition and relational statements. As the execution of the task proceeds, the world model is updated in such a way that it will match each situation. Simultaneously, changes are performed in the real world. Motion statements are only to simplify the process of inferring actions which will transfer the system state from the initial to the goal situation. As all motions are defined in relation to some features of the objects, the necessity of defining virtual (nonexistent) objects arises sometimes. The following few lines of code should give the flavour of programming in this language. This program shifts a bottle with a square cross-subsection from one place on the table to another. The goal location of the bottle is expressed in terms of the position of a certain point (vertex) on the bottom circumference of the bottle.

```
REMARK lift the bottle off the table
MOVE/bottle,PERPTO,table;
REMARK describe the situation: the bottle is above the table
AGAINST/VERTEX of bottle, virtual_plane;
REMARK transfer the bottle to the new location
MOVE/bottle,PARLEL,table
MOVE/bottle,PERPTO,table
REMARK describe the goal situation
AGAINST/VERTEX of bottle, new_vertex
```

MOVE statements cause the robot and objects to move, but the range of motion is determined from the description of the situation following the MOVE instructions.

TORBOL

In the following sub-sections the basic notions that are used in TORBOL will be outlined.

Virtual and real environment

The real environment is composed of the robot itself, its surroundings, and the cooperating devices. A full and exact description of such an environment is neither necessary nor possible. From the point of view of the class of tasks that have to be executed (even by a very universal robot) only specific features of the real environment are important. So the model of the real environment, that takes into account all the necessary details (from the point of view of the realized task), is called the virtual environment. The programmer, through a programming language, perceives only the virtual environment. In other words, in a programming language only some features of the real environment can be described. A real environment reduced in such a way is called the virtual environment.

Objects

The virtual environment, in which TORBOL instructions operate, is constructed out of models of real objects. Each object possesses certain properties which characterize it. These can be such features as: the position of its base, its shape, or the description of the place most suitable for getting hold of it. An important characteristic is the path of safe approach to the object. The set of features is described by the attributes of the object.

Two kinds of objects were introduced in TORBOL: indexed and non-indexed objects. A non-indexed object is identified only by its name, and an indexed object is identified by both its name and the values of the indices. Up to three indices can be used. It was assumed that all the objects with the same name (the values of indices can vary) possess the same features and thus the same attributes. This characteristic of the language is very useful in the case of objects placed on pallets or in containers.

Attributes of objects

The language contains a pre-defined constant set of possible attributes. Each object is characterized by a non-empty subset of the set of attributes. The choice of attributes featuring an object is left to the programmer and is made according to the requirements imposed by the executed task.

The geometric properties of the objects are described by co-ordinate frames (or in short, simply, frames), that is: orthogonal, right-handed sets of Cartesian basis vectors with common origin. The primary co-ordinate frame affixed to an object is called BASE. The other four pre-defined frames are named TOP, BOTTOM, INSIDE and HANDLE. Their **locations** (position and orientation) are relative to the BASE frame. They specify certain characteristic places on or in the object. In principle:

TOP – a place where another object can be placed,
BOTTOM – a place where contact between the object and its support can occur,
INSIDE – the interior of the object (in case of hollow objects),
HANDLE – is the location best suited for grasping the object.

Moreover, an attribute named PATH was introduced. It determines a route of safe approach to the object. PATH is an ordered list of frames specified relative to BASE. When an object is to be grasped, the frame affixed to the tool centre point (TCP) must pass through all the frames listed in the PATH. This attribute allows an implementation of elementary collision avoidance.

The attribute GRASPED decides whether the object is held by the gripper.

The attribute INPUT specifies the signal inputs to the object (e.g. input to a cooperating device – turning it on or off). To such an input an output of the robot control system has to be connected. The last attribute is called SHAPE. This attribute was introduced only for the sake of simulation of the execution of a program written in TORBOL. It enables a graphical representation of an object. The shape of the object can be represented as a concatenation of the following primitive shapes:
CUBOID,
PRISM with an equilateral polygon in its base,
PYRAMID with an equilateral polygon in its base.

Object classes

Essentially an object class is a set of objects described by the same set of attributes. Various objects of the same class will have different values of their attributes. Nevertheless, the language makes possible the specification of several classes of objects possessing the same sets of attributes. This possibility was introduced because the set of attributes defined in the language is not very abundant and it increases the readability of programs. Moreover, for the sake of automatic verification of semantic validity of a program it may be important that the objects with the same sets of attributes belong to classes with differing names.

Relations

Since a program in TORBOL is treated as a sequence of situations (relations) which the objects have to attain, a mechanism for defining such relations has to be created. A higher level relation[4] is defined by supplying lower level relations[5]. The compiler will automatically generate such motion commands for the robot, so that the lower level relation becomes valid. The occurrence of the lower level relation is equivalent (by definition) to a situation in which the pertinent high level relation takes place. The above considerations will become much clearer after reading the following example.

[4] A relation between objects.
[5] Relations between the values of attributes of objects.

If two classes of objects named bolts and plates respectively are specified in a program, then between the objects of these classes a relation named “in” can be defined. A situation in which a bolt will be in relation “in” with a plate occurs when the bolt is placed inside the plate hole (Fig. 3.1). The relation “in” is a higher order relation. Now

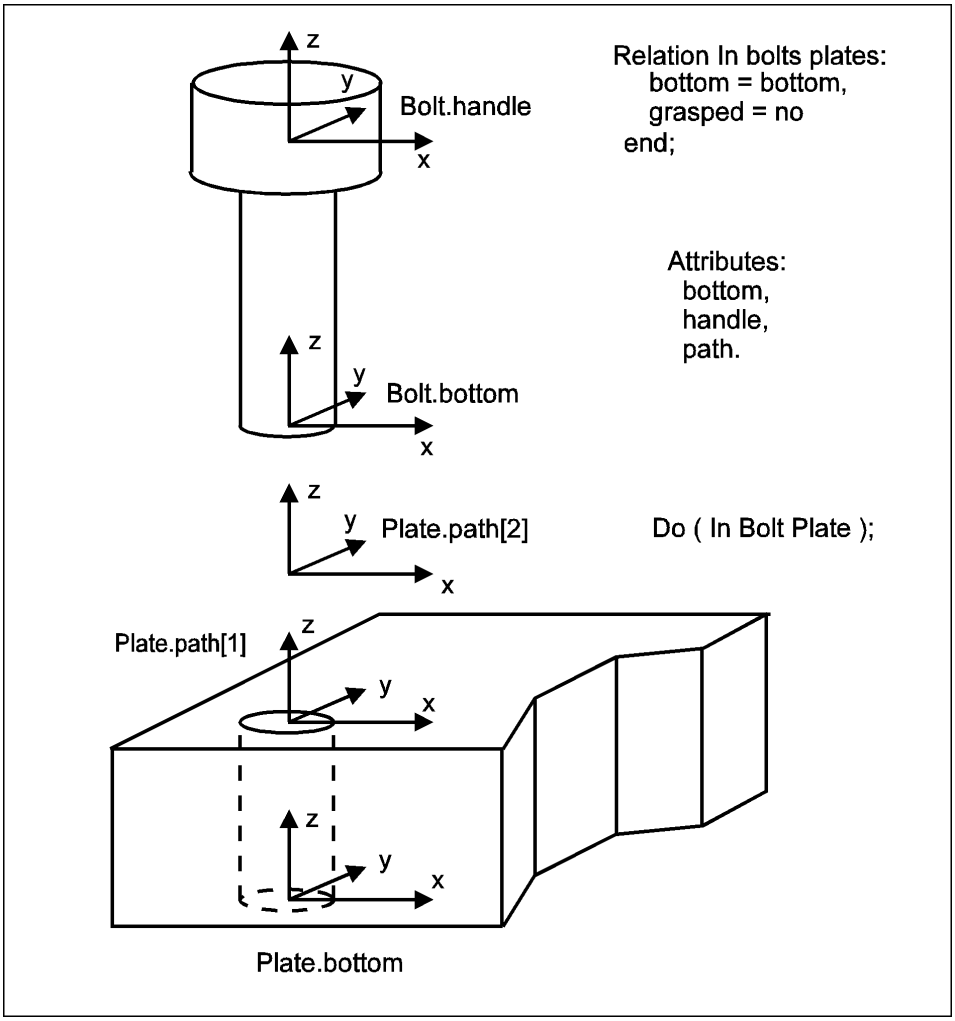

Figure 3.1. Definition of the relation In between a Bolt and a Plate

it has to be described in terms of its equivalent lower level relations. A lower level relation can be represented by an “equal to” relation between the values of the BOTTOM attribute of the bolt and the BOTTOM attribute of the plate. Of course, the values of both attributes (frames) have to be expressed in the same co-ordinate system, that is in the global co-ordinate frame. A formal method of defining high level relations in terms of low level (mathematical) relations is presented in [164, 170, 174].

In TORBOL both unary and binary relations can be defined, where the latter can be of two types: symmetric and asymmetric. A unary relation determines the feature which we want the object (being an argument of this relation) to acquire. For instance, if we consider a lathe to be our object, then we may demand the lathe to be activated (ACTIVATED LATHE). ACTIVATED is the name of a unary relation, and the LATHE is the object being the argument of this relation. Whether the LATHE is activated or not is its feature. The higher order relation (ACTIVATED) must be defined in terms of lower order relations [194] between the value of the attribute INPUT (of an object named the LATHE) and a constant value (here a binary value). The robot control system will output this value causing the activation of the lathe.

An example of a binary relation has been cited above. Only the symmetric and asymmetric binary relations have to be differentiated. If two objects named OB1 and OB2 are in a relation REL with each other, what can be written down as (REL OB1 OB2), then in the case of a symmetric relation the displacement of object OB1 causes the displacement of object OB2, and vice versa. When the relation REL is an asymmetric one, the motion of object OB2 brings about the motion of object OB1, but not the opposite. Symmetric relations take place when two objects are fixed to each other in some way (e.g. glued or screwed together). Asymmetric relations occur when one object is placed on top of the other or inside the other (with a loose fit).

Signals and events

The robot control system may obtain information from the environment through signal lines. This information can be utilized to synchronize the actions of the robot with the production process. With every signal line (IN_SIGNAL) we can associate one or more names of events. If the signal attains the value specified in the definition of the event (EVENT), then we consider that this event took place. TORBOL has an instruction waiting for an event to take place. In the system a signal named TIME was distinguished. It is responsible for measuring the elapsed time. Usually, this signal indicates that a certain amount of time has passed.

Supplementary data types

TORBOL has four supplementary data types: INTEGER, REAL, LOGICAL and FRAME. The application of the former three is obvious. The last data type is used for specifying positions and orientations of certain characteristic places of an object and points (locations) along a path. On these data types all mathematical operations appropriate to them can be performed.

Initialization directives

Initialization directives assign initial values to supplementary variables and attributes of objects, and moreover characterize the initial relations taking place between the objects of the virtual environment. A supplementary variable can be assigned an initial value by a variable initialization directive (<<). To assign initial values to attributes of objects, an OBJECTS initialization directive must be used. To enter into the database the information about the initial relations occurring between the objects, a RELATIONS initialization has to be employed. All the initialization directives can only be used inside the initialization subsection of a program (starting with the INITIALIZATION and ending with ENDINIT keyword).

Instructions

The DO instruction is the principal command of the language. It enforces a relation. In other words, it causes a situation in which a binary relation between two objects (being its arguments) takes place, or a situation in which an unary relation occurs, causing a specified object to acquire a particular feature.

The WAIT instruction causes the robot to wait for an event to occur. While the robot is waiting, other devices may make some changes in the real environment. These changes have to be reflected in the data base holding the state of the virtual environment. That is why the definition of the event has to contain all the information about the changes in the values of attributes of objects and new relations that will take place after the occurrence of the event. Due to such a definition of the event, the updating of the internal data base is possible.

To the supplementary variables as well as to the attributes of objects new values can be assigned by an assignment instruction (:=). Nevertheless, the programmer is warned against doing so, because it is easy to introduce false data into the data base in such a way. The values of attributes of objects being moved by the robot are updated automatically, but sometimes, when the robot is not the cause of motion of the object, the programmer may have to change the values of attributes using this instruction. Practice shows that these are rather rare cases.

Finally, four instructions (well-known from the general purpose high-level languages, e.g. Pascal) controlling the execution of a program have been introduced into the language. These are: IF THEN ELSE ENDIF, WHILE PERFORM ENDWHILE, REPEAT UNTIL ENDREPEAT, and FOR ENDFOR loop. The first three are executed depending on the condition being their argument. The condition may be constructed out of logical expressions, binary relations (the fact of a binary relation taking place between the indicated objects is being checked) and events (the fact that an event took place is being checked).

DO instruction

The basic instruction of the language causing changes in the virtual environment is the DO instruction. This is why it will be dealt with in more detail here. Its syntactic form is as below:

DO (<relation_name> <object_name> [<object_name>]) [<modification>];

where the angle brackets contain the non-terminal symbols and the square brackets enclose optional parts of the instruction. When the specified relation is a unary relation, then the name of the second object is obviously not needed, but if it is a binary relation, the names of both objects are necessary. The optional <modification> part states how the motion is to be executed: CAUTIOUSLY, MODERATELY or QUICKLY. It influences only the speed of motion during the execution of a binary relation. In the future implementations <modification> will be the name of the modifying process influencing the execution of the DO instruction according to the information received from sensors [153] or imposing oscillations on the basic motion.

EXAMPLE:

DO (IN BOLT PLATE) SLOWLY;

The execution of this instruction will cause the BOLT to be placed IN the PLATE (Fig. 3.1). The relation IN between the specified objects will be realized. The speed of

motions will be 20% of the maximum possible speed. The details of the algorithm that executes the binary relation are shown in Fig. 3.2.

The motion of the gripper along the approach-departure trajectory is performed in the following manner. When the gripper approaches the object, then it moves from the last to the first frame in the list being the value of the PATH attribute of this object. When the gripper departs from the object, then it passes through these frames in the reverse order. If the movable object is in a symmetric relation with any other object, then this object will be transferred too, and the values of its attributes will be suitably up--dated. The same thing happens to the objects that are in an asymmetric relation with the movable object if the movable object had been the second argument of this relation when it was created. The asymmetric relations in which the movable object was the first argument are automatically deleted from the data base. In other words, the asymmetric relations are dealt with in the following manner. All the objects that are standing on top of the movable object are displaced with it, and all the objects that are supporting the movable object are left behind.

It should be explained why during the creation of a new binary relation, when the movable object was already in an asymmetric relation as its first argument, the gripper would traverse a different trajectory than when the movable object was in a symmetric relation or in no relation at all. If the movable object is in no relation with other objects and the gripper, while approaching this object, did not collide with any obstacles, then it is assumed that when the arm is retracted, it will also avoid obstacles (this reasoning does not guarantee that the retracted object will not hit obstacles, but such a situation is rather improbable in the case of relatively small objects). In the case when the movable object was in an asymmetric relation (as its first argument), and this relation had been created without collisions, then by tracing the same trajectory in the opposite direction the obstacles will probably also be avoided (an obstacle can be encountered, if from the time of the creation of this relation the immediate surroundings of the object have been changed considerably). When the object is relatively big or the surroundings have been altered, the value of the attribute PATH has to be changed. In the case of an object being in no relation with other objects the gripper will move through the same approach trajectory twice – once approaching the object and once retracting it. In the case of the movable object being already in an asymmetric relation, the gripper will traverse such a trajectory that the characteristic place of this object will pass through the approach trajectory of the object with which the movable object is in a relation. The symmetric relation does not distinguish any of the objects, so it was decided that this case would be treated in the same way as the case of the movable object being in no relation with other objects. Such an approach to the realization of a binary relation gives us a "coarse" version of a program quickly. Then, during simulation, if collisions are detected, the program can be "tuned" by adjusting appropriate PATH attributes.

The execution of a unary relation amounts to output, from the control system of the robot, of a signal with a magnitude specified in the definition of this relation. The output OUT_SIGNAL(i), where i is the number of the output, must be physically connected to the INPUT of the object.

TORBOL program structure

A program written in TORBOL consists of four parts (possibly empty):

– program heading,

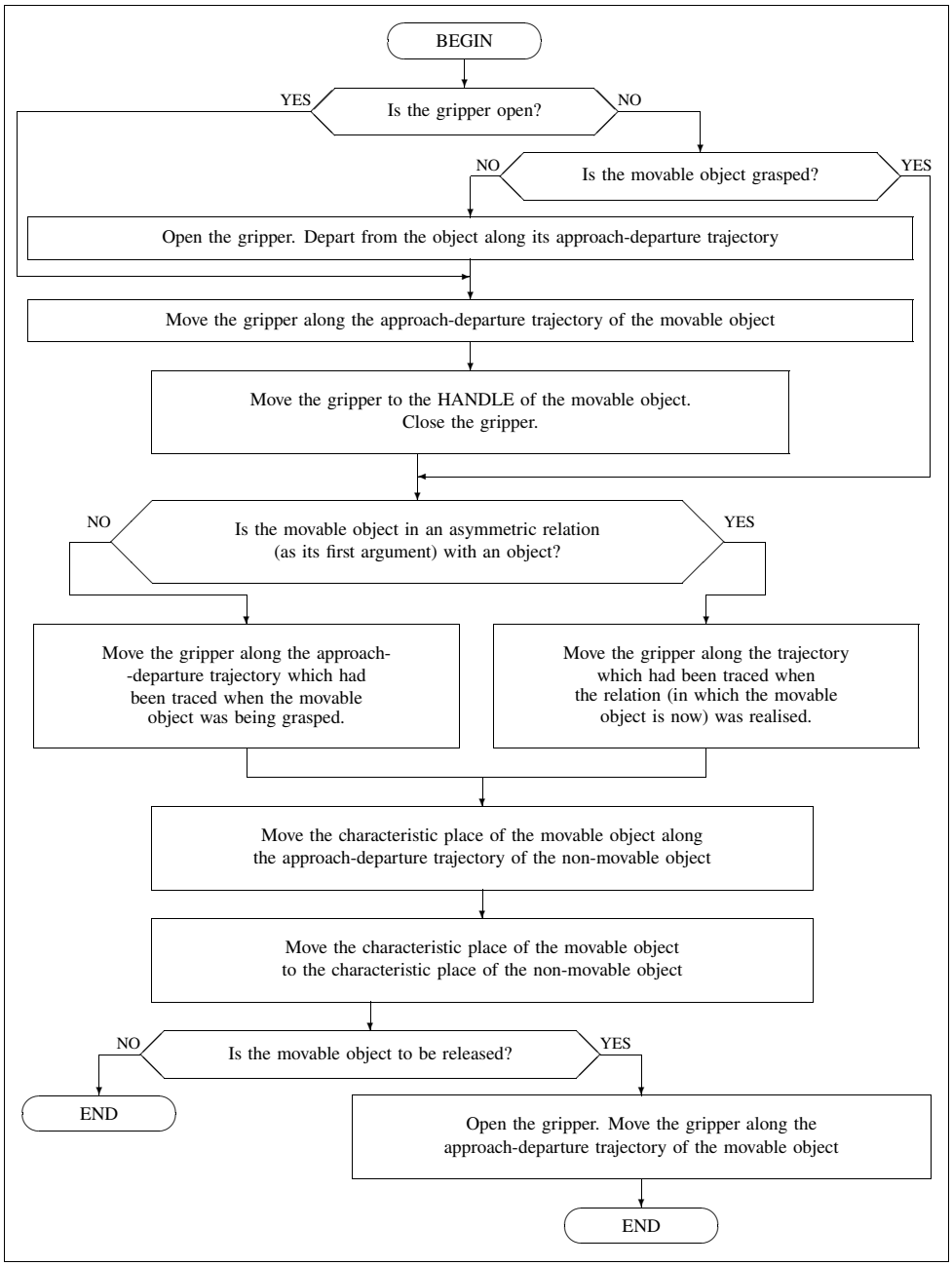

Figure 3.2. Flowchart of the execution of a binary relation

– declaration subsection,
– initialization subsection,
– program subsection.

The program starts with the keyword TITLE followed by the program name (this is the heading). The declaration subsection contains the declarations of supplementary variables, object classes, objects, relations, and events. The initialization subsection holds the initial state of the virtual environment and supplementary variables. If the system has a camera, then the state of the virtual environment can be read in by "looking" at the real environment. For the time being the initial state of the virtual environment has to be supplied by the programmer (a rather time-consuming activity). He has to assign initial values to all the attributes of objects and some of the supplementary variables. Moreover, he has to state which relations already exist in the environment. The program subsection contains the instructions which determine the task that the robot is to execute. The program is terminated by the keyword FINISH.

TORBOL example

The basic properties of the language will be illustrated by a programming example. For this purpose one of the tasks that are common to robots installed in factories has been chosen. The effects of simulating the execution of this program are illustrated by Fig. 3.3. The braces contain comments.

```
TITLE EXAMPLE;
{ Robot transfers WorkPiece from the Source conveyor to the hydraulic Press, ac-
  tivates the Press and waits for the Press to signal that its job is done. It then
  retrieves the machined WorkPiece from the Press and places it on the Receiver
  conveyor }
DECLARE
  OBJECT_CLASS Conveyor ATTRIBUTES
    BASE, TOP, SHAPE, PATH, INPUT
  END ;
  OBJECT_CLASS Machine ATTRIBUTES
    BASE, TOP, INPUT, SHAPE, PATH
  END ;
  OBJECT_CLASS Material ATTRIBUTES
    BASE, BOTTOM, HANDLE, SHAPE, GRASPED, PATH
  END ;
  OBJECT Source CLASS Conveyor
    SHAPE : CUBOID (3000 , 300 , 575 )
     ( CUBOID (100 , 100 , 100 ) AFFIXED_AT [ 0 , 1200 , 575 , 0 , 0 , 0 ] !
       CUBOID (100 , 100 , 100 ) AFFIXED_AT [ 0 , 800 , 575 , 0 , 0 , 0 ] !
       CUBOID (100 , 100 , 100 ) AFFIXED_AT [ 0 , 400 , 575 , 0 , 0 , 0 ] ),
    INPUT : OUT_SIGNAL(1)
  END;
  OBJECT Press CLASS Machine
    SHAPE : CUBOID (1000 , 600 , 130 )
    ( PRISM (10 , 60 , 1300 ) AFFIXED_AT [ 0 , 300 , 130 , 0 , 0 , 0 ]
     ( PRISM (10 , 60 , 1300 ) AFFIXED_AT [ 0 , –600 , 0 , 0 , 0 , 0 ]
```

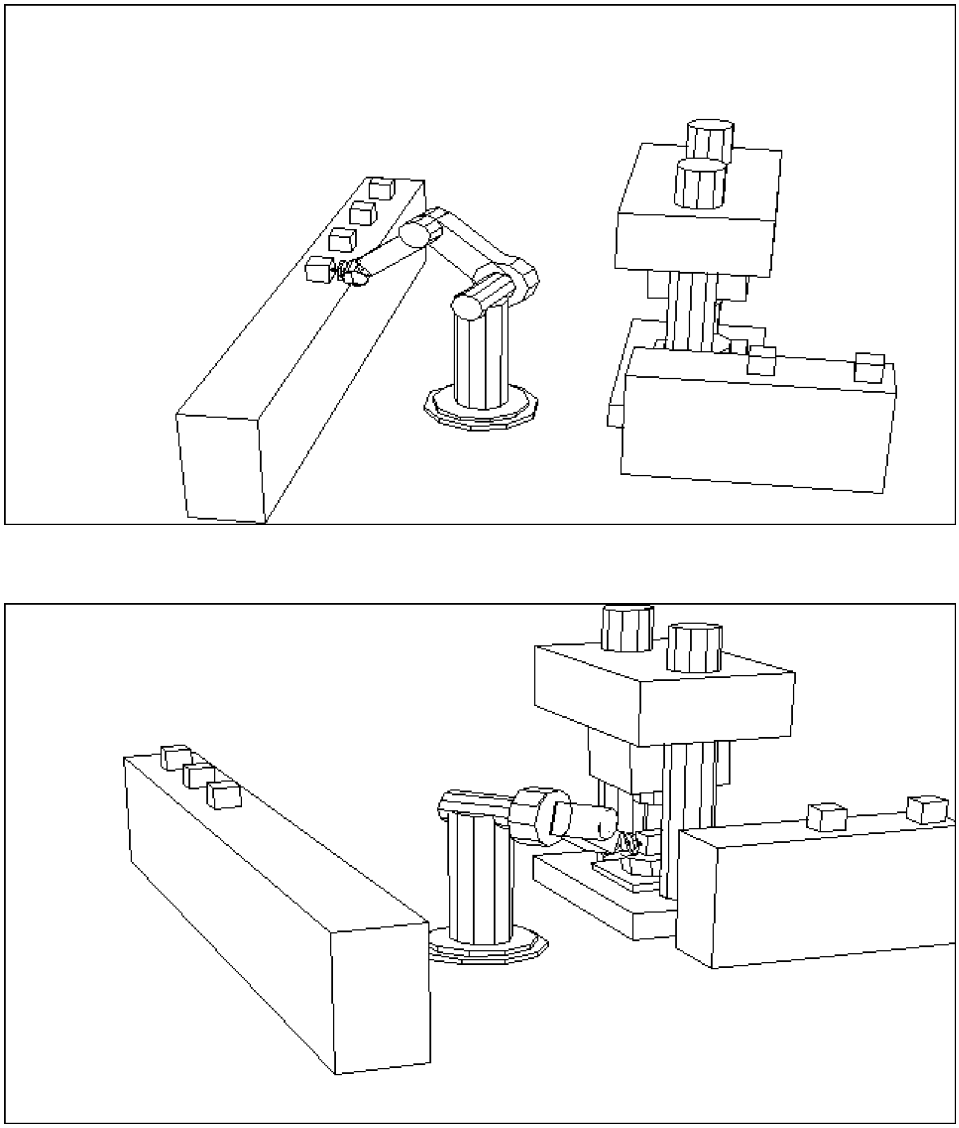

Figure 3.3. Exemplary TORBOL program simulation

```
      ( CUBOID (1000 , 600 , 300 ) AFFIXED_AT [ 0 , 300 , 800 , 0 , 0 , 0 ]
        ( CUBOID (400 , 400 , 300 ) AFFIXED_AT [ 0 , 0 , 0 , 0 , –PI , 0 ]
          ( PRISM (10 , 60 , 100 ) AFFIXED_AT [ 0 , 0 , 300 , 0 , 0 , 0 ]
            ( CUBOID (400 , 400 , 50 ) AFFIXED_AT [ 0 , 0 , 500 , 0 , –PI , 0 ]
              ( PRISM (10 , 90 , 80 ) AFFIXED_AT [ 0 , 0 , 50 , 0 , 0 , 0 ]
    )))))),
   INPUT : OUT_SIGNAL(2)
END;
OBJECT Receiver CLASS Conveyor
```

```
    SHAPE : CUBOID (1000 , 300 , 565 )
     ( CUBOID (100 , 100 , 100 ) AFFIXED_AT [ 0 , −400 , 565 , 0 , 0 , 0 ] !
       CUBOID (100 , 100 , 100 ) AFFIXED_AT [ 0 , 0 , 565 , 0 , 0 , 0 ] ),
    INPUT : OUT_SIGNAL(4)
  END;
  OBJECT WorkPiece CLASS Material
    SHAPE : CUBOID (100,100,100)
  END ;
  RELATION On Material Conveyor :
    BOTTOM = TOP ,
    GRASPED = NO
  END ; { On }
  RELATION In Material Machine :
    BOTTOM = TOP,
    GRASPED = NO
  END ; { In }
  EVENT WorkPiece_in_position : IN_SIGNAL(3) = 1
    OBJECTS
     WorkPiece = [ −750 , 300 , 575 , 0 , 0 , 0 ]
    RELATIONS
     On WorkPiece Source
  END ;
  EVENT Work_done : IN_SIGNAL(1) = 1 END ;
  EVENT Pressed : IN_SIGNAL(2) = 1 END ;
  UNARY RELATION Loaded Conveyor :
    INPUT = 1
  END ;
  UNARY RELATION Unloaded Conveyor :
    INPUT = 0
  END ;
  UNARY RELATION Working Machine :
    INPUT = 1
  END ;
  UNARY RELATION Disabled Machine :
    INPUT = 0
  END ;
ENDDECL

INITIALIZE
  OBJECTS
   Source
    BASE = [ −750 , 300 , 0 , 0 , 0 , 0 ],
    TOP = [ 0 , 0 , 575 , 0 , 0 , 0 ],
    PATH = # 2, Source.TOP∗[ 0 , 0 , 150 , 0 , 0 , 0 ],
               Source.TOP∗[ 200 , 0 , 200 , 0 , 0 , 0 ]#;
   Receiver
    BASE = [ 1150, −400, 0 , PI/2 , 0 , 0 ] ,
```

```
    TOP = [ 0 , 400 , 565 , PI/2 , 0 , 0 ],
    PATH = # 2, Receiver.TOP*[0 , 0 , 50 , 0 , 0 , 0 ],
              Receiver.TOP*[100 , −100 , 50 , 0 , 0 , 0 ] #;
  Press
    BASE = [ 850 , 400 , 0 , PI , 0 , 0 ] ,
    TOP = [ 0 , 0 , 260 , 0 , 0 , 0 ],
    PATH = # 3, Press.TOP*[ 0 , 0 , 50 , 0 , 0 , 0 ],
              Press.TOP*[ 50 , 0 , 50 , 0 , 0 , 0 ],
              Press.TOP*[ 100 , 0 , 50 , 0 , 0 , 0 ] #;
  WorkPiece
    BASE = Source.BASE * Source.TOP,
    BOTTOM = [ 0 , 0 , 0 , 0 , 0 , 0 ] ,
    HANDLE = [ −10 , 0 , 50 , PI , PI/2 , 0 ] ,
    GRASPED = NO ,
    PATH = # 2, [ 0 , 0 , 100 , PI , PI/2 , 0 ],
              [ 100 , 0 , 100 , PI , PI/2 , 0 ] #
  END { OBJECTS } ;
ENDINIT

PROGRAM { for the robot }
{ Execute the loop until signaled that the work is done }
  WHILE not ( ? Work_done ) PERFORM
    { Wait for the WorkPiece to be in position }
    WAIT WorkPiece_in_position;
    { Put the WorkPiece in the Press }
    DO ( In WorkPiece Press );
    { Signal the Source conveyor that it is empty }
    DO ( Unloaded Source );
    { Activate the Press }
    DO ( Working Press );
    { Wait for the WorkPiece to be pressed }
    WAIT Pressed;
    { Turn off the Press }
    DO ( Disabled Press );
    { Take the WorkPiece from the press and put it on the Receiver conveyor }
    DO ( On WorkPiece Receiver );
    { Signal that the Receiver conveyor is loaded (thus activate it) }
    DO ( Loaded Receiver );
    DELETE ( On WorkPiece Receiver );
    { Turn off the Receiver conveyor }
    DO ( Unloaded Receiver );
    { Activate the Source conveyor }
    DO ( Loaded Source );
  ENDWHILE
FINISH.
```

Comparison of object level RPLs

AL and TORBOL rely on frames in the description of objects. TORBOL and RAPT use shape to describe bodies. If an object changes its shape during machining (the shape has to be redefined), or if the object is flexible (e.g. rubber pads), some problems arise in the description of the task in RAPT. On the other hand, most CAD programs generate as an output descriptions of shapes of objects. RAPT could readily use this output in the description of the virtual environment, thus cutting down the program development time. AL and RAPT do not differentiate the relationships between parts of objects and between objects themselves. TORBOL uses different means to define the relationships between parts of an object (attributes) and different to specify relationships between objects (relations). In AL the definition of an object is a bit fuzzy, in RAPT it is much stricter, whilst in TORBOL an object is a very distinct entity. The systems relying on frames have problems with specifying motions of robots with less than six degrees of freedom. If, for instance, the tool is axially-symmetric and the robot has five degrees of freedom, it is very difficult to specify attainable positions, although the task can be realized without problem (when programmed by teaching). TORBOL solves this problem by defining a rotational virtual degree of freedom about the axis of the tool. RAPT relies on artificial intelligence techniques for generating motions, so these methods have also to be used to solve the above problem.

In the case of AL, motion is specified in terms of motions of frames and the system relies partly on the programmer for the proper actualization of the world model. This procedure is error prone. As in RAPT the motion of an object is specified as a motion of a body satisfying a goal situation, no discrepancy between the real and the virtual environments can arise here. The problem is that artificial intelligence methods have to be employed to deduce the motions of the arm. This cannot be done on-line. As a TORBOL program is a sequence of relations that have to be created during the execution of the task, here again no discrepancy between the real world and the world model can arise.

Only AL can exert considerable influence on the execution of the intermediate part of the trajectory (by modification clauses). When information is gathered by sensors it can readily be used if an adequate clause is present in the implementation of the language; if not – the language can be enhanced. Some enhancements have been introduced to RAPT [153] and TORBOL [172] to accommodate sensors, but they are not very natural. The problem of incorporation of sensors into object level systems is as yet an open research problem.

From what has been said, it is very difficult to judge which system is the best. Each has its advantages and drawbacks. Object level robot programming languages will have to undergo further development to be used on the factory floor. A language treating objects as distinct entities with attributes, where one of them would be a variable (flexible) shape, and enabling dual definition of relations (i.e. utilizing frames as well as the shapes of objects), emerges as the target for the future research. Moreover, such a future language must take into account the diversity of sensors that can be incorporated into a robotic system and deal properly with aggregation and utilization of data from these sensors. Last but not least, unexpected events occurring during normal execution of the program have to be dealt with, i.e. error handling and error recovery procedures should be easy to implement in such a language.

3.3 Hybrid programming

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

VAL II [194] can be treated as a hybrid robot programming language, because besides the possibility of supplying numeric arguments to the motion instructions (e.g. MOVE, MOVES[6]), the location to which the arm is commanded can be taught-in. The arm is transferred to the goal location by the teach-pendant. This location will be stored as an argument of the motion instruction when it is typed in. Usually, not many of such locations have to be taught-in. The remaining locations that the robot arm must attain either do not have to be stated very precisely or can be specified in relation to the exact ones that have been taught-in.

Until the calibration problem has been solved satisfactorily, some form of "calibration by teaching" will have to exist, so the hybrid programming will be used rather than the pure off-line method. Nevertheless, the off-line component of programming will be dominating, especially due to the tendency of incorporating sensors into modern robotic systems. Moreover, in the author's opinion, for reasons explained in chapters 5 and 6, rather robot specific libraries of procedures coded in universal computer programming languages will be used than such specialised languages as AL, RAPT or TORBOL. In the next section the influence of RPL instructions on the components of a robotic system will be discussed. This discussion will point out the necessary components of an RPL instruction set and will give some hints of its implementation.

4 Robot language instructions

The following discussion will show what influence the RPL instructions must exert on the robotic system, for the system to be fully controllable by this instruction set. The influence of RPL instructions on the system will be described in terms of system state changes.

Substitution of (2.3) into (2.1) yields the following:

$$s = < e, r, < c_p, c_d, c_v >> \tag{4.1}$$

When the description (2.7) of instruction semantics is assumed, two cases arise. For certain instructions the sequence of intermediate states is reduced to zero (only the initial and terminal states exist). Other instructions require that the number of intermediate

[6] In this case the motion instruction is followed by an exclamation mark.

states should be greater than zero. In the former case the instruction will be called a **single-step instruction** and in the latter a **multi-step instruction**. CPL instructions are considered to be single-step, unless they are compound instructions.

Regardless of whether an instruction is single– or multi-step, it is of utmost importance to know what the state s^{i+1} of the system will be after it has executed a single step from the current state s^i, where superscript i denotes the discrete state number.

In the following discussion the influence of a single instruction on the state of the system will be considered. This influence will be described as a set of certain mappings. It is assumed that c_p will not be affected by the execution of any instruction, so $c_p = const$, hence c_p will not be an argument of the mappings describing the semantics of an instruction.

4.1 Influence of instruction execution on the state of the system components

Each and every RPL instruction influences c_d, because the program counter changes its state with the execution of an instruction. It designates both the next step and the subsequent instruction that will be executed, so it has to be incremented with the execution of each step and instruction in the program. The next instruction to be executed can be determined by the value of an expression computed out of all or any combination of the terms: e, r, c_d, c_v. If attention is concentrated on the changes in the state of c_d, then the following cases are possible:

$$c_d^{i+1} = \begin{cases} f_{d_0}() = const & - \text{ unconditionally jump} \\ f_{d_1}(c_d^i) & - \text{ execute next step or instruction} \\ f_{d_2}(c_d^i, c_v^i) & - \text{ loop or conditionally branch on variable expression} \\ f_{d_3}(c_d^i, r^i) & - \text{ wait, loop or conditionally branch on expression receptor state} \\ f_{d_4}(c_d^i, e^i) & - \text{ conditionally branch on effector state expression} \\ f_{d_5}(c_d^i, c_v^i, r^i) & - \text{ wait, loop or conditionally branch on receptor state and variable expression} \\ f_{d_6}(c_d^i, c_v^i, e^i) & - \text{ loop or conditionally branch on effector state and variable expression} \\ f_{d_7}(c_d^i, r^i, e^i) & - \text{ wait, loop or conditionally branch on effector and receptor state expression} \\ f_{d_8}(c_d^i, c_v^i, r^i, e^i) & - \text{ wait, loop or conditionally branch on effector and receptor state and variable expression} \end{cases} \tag{4.2}$$

Functions $f_{d_j}, j = 0, \ldots, 7$ are special cases of function f_{d_8}. It is important to note that from (4.2) two distinct cases result: f_{d_0}, f_{d_1} – where the next step or instruction to be executed is determined only on the basis of the information contained in the program designator and the executed instruction itself (i.e. program counter is incremented by a constant value or a constant value is assigned to it); and all the other cases – where the next instruction depends on the value of an expression formed by variable values, sensor readings and the current state of effectors. In the latter case the value of the expression can be used to control the execution of a loop, conditional branch or a delay. The above

distinction divides the instruction set into two categories: **unconditional instructions**, and **conditional instructions**.

If the state of variables (memory) changes due to the execution of an instruction, then the following cases are possible:

$$c_v^{i+1} = \begin{cases} f_{v_0}() = const & - \text{ store constants} \\ f_{v_1}(c_v^i) & - \text{ store value of variable expression} \\ f_{v_2}(r^i) & - \text{ store aggregated receptor state} \\ f_{v_3}(e^i) & - \text{ store processed effector state} \\ f_{v_4}(c_v^i, r^i) & - \text{ store aggregated value of variable expression and receptor state} \\ f_{v_5}(e^i, r^i) & - \text{ store processed effector and receptor state} \\ f_{v_6}(c_v^i, e^i) & - \text{ store processed effector state and variable values} \\ f_{v_7}(c_v^i, r^i, e^i) & - \text{ store processed variable values, effector and receptor state} \end{cases} \tag{4.3}$$

An instruction that is executed implies the choice and form of the functions $f_{v_j},\ j = 0, \ldots, 7$. Instructions are stored in the memory of the control part of the system. This is the only influence that c_d has on the future state of variables. That is why it is not expressed explicitly as an argument of functions $f_{v_j},\ j = 0, \ldots, 7$.

In general, either the value of a certain expression is computed and stored (all possible forms of expressions are denoted by (4.3)) or no evaluation of expression takes place and no modification of c_v is executed by an instruction. In the former case the instruction is called the **storing instruction** and in the latter – the **non-storing instruction**.

If the state of effectors changes due to the execution of an instruction, then the following cases are possible:

$$e^{i+1} = \begin{cases} f_{e_0}() = const & - \text{ absolute motion to a position defined by a constant} \\ f_{e_1}(c_v^i) & - \text{ absolute motion to a position defined by a variable expression} \\ f_{e_2}(r^i) & - \text{ absolute motion to a position defined by receptor readings} \\ f_{e_3}(c_v^i, r^i) & - \text{ absolute motion to a position defined by variable and receptor reading expression} \\ f_{e_4}(e^i) & - \text{ incremental motion defined by a constant} \\ f_{e_5}(e^i, r^i) & - \text{ incremental motion defined by receptor readings} \\ f_{e_6}(e^i, c_v^i) & - \text{ incremental motion defined by a variable expression} \\ f_{e_7}(e^i, c_v^i, r^i) & - \text{ incremental motion defined by variable and receptor reading expression} \end{cases} \tag{4.4}$$

The instruction, and so c_d, implies the choice and form of $f_{e_j},\ j = 0, \ldots, 7$.

Effectors are influenced by **motion instructions**. The following possibilities arise:

– either **absolute motion** (in the computation of the next effector position the current position is disregarded – functions: $f_{e_j}, j = 0, \ldots, 3$ are used) or **incremental motion** (next position is computed on the basis of current position – functions: $f_{e_j}, j = 4, \ldots, 7$ are utilized) takes place. In the former case the instruction is called

the **absolute motion instruction**, and in the latter case – the **incremental motion instruction**;

– either receptors are used – **sensory motions** (functions: f_{e_2}, f_{e_3}, f_{e_5}, f_{e_7} are used) or sensors are not used – **sensorless motions** (functions: f_{e_0}, f_{e_1}, f_{e_4}, f_{e_6} are utilized). In the former case the instruction is called the **sensory motion instruction**, and in the latter – the **sensorless motion instruction**.

Formulae (4.2), (4.3), (4.4) point out the factors that can influence the execution of an instruction. They show by what factor each of the elements of the system state can be modified. The only element of the system state that cannot be modified in this way is the state of receptors r. The state of receptors can be influenced indirectly by changing the state of effectors or by events external to the system (external agents). It should be remembered that in subsection 2.3 instructions taking into account the changes in effector state due to the actions of external agents were treated as motion instructions, and consequently so are now.

Formulae (4.2), (4.3), (4.4) show what can influence each component of the system. The arguments of **transfer functions** $f_{d_j}, j = 0, \ldots, 8$; $f_{v_j}, j = 1, \ldots, 7$ and $f_{e_j}, j = 1, \ldots, 7$ are at the same time the parameters of instructions. Notation (4.2), (4.3), (4.4) takes into consideration all possibilities of instruction semantics, disregarding the syntactical form of the instruction.

An instruction can simultaneously belong to several classes. An **instruction** is:

– either **conditional** or **unconditional**,
– either **storing** or **non-storing**,
– either **motion** or **non-motion** – in the former case the instruction is:
  - either **absolute motion** or **incremental motion**,
  - either **sensory motion** or **sensorless motion**.

The definition of an RPL should be such that all the above instruction classes should be included. In the discussion of RPL instruction semantics, besides knowing what can possibly influence each component of the system, it is also important to know at which instant in the execution of the instruction this influence is exerted.

4.2 Utilization of sensors

The job of a robot system is to execute a task supplied to it in the form of a program. Motion instructions in a program cause changes of the state of effectors e. If notation (2.7) is assumed, then the execution of a motion instruction begins in an **initial state**, ends in a **terminal state**, and traverses a sequence of **intermediate states**. The majority of robots are controlled by digital computers, so the execution of each instruction is subdivided into **steps**. Each step results in the change of the system state from one intermediate state to the next.

In each intermediate state (or while attaining it) the state of the system can be measured – **monitored** by sensors. The current state of the system can only be monitored, but the future intermediate states can be influenced – **controlled**. The initial state can be treated as a current intermediate state at the beginning of motion instruction execution. The terminal state is the current intermediate state in which the execution of

the instruction terminates. As the initial and terminal states are special cases of current intermediate states, both of them can only be monitored. These ideas are illustrated symbolically by Fig. 4.1.

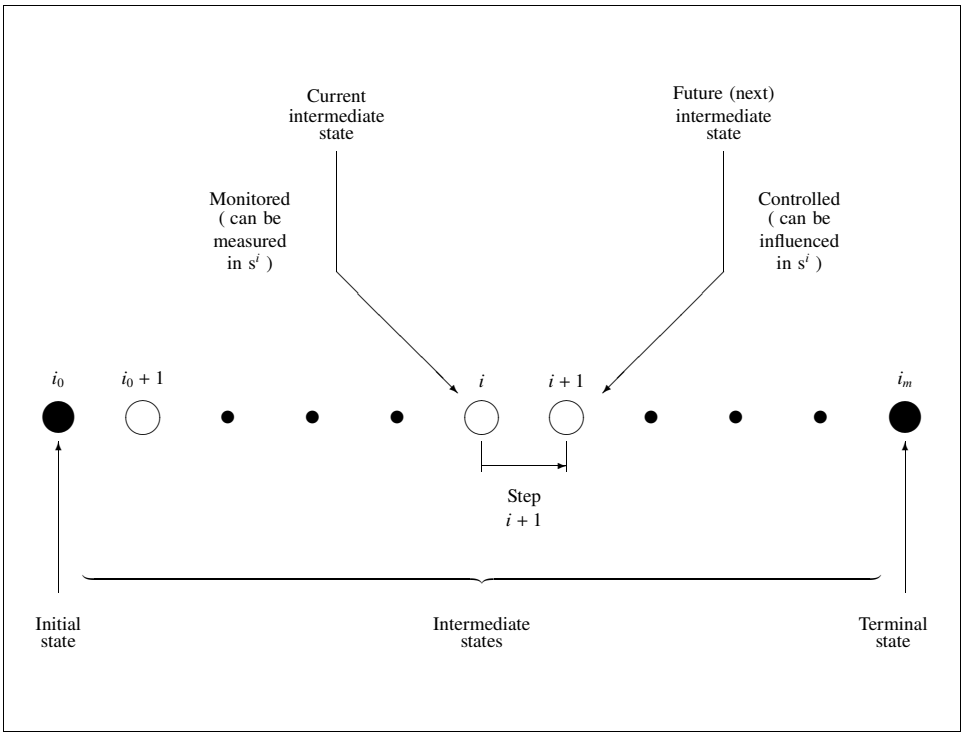

Figure 4.1. Evolution of the system state during execution of a motion instruction

The monitoring of the system state is performed by receptors. The raw data obtained from them cannot usually be utilized directly to monitor or control the system. It has to be transformed into a useful form by **data aggregation** and the result is stored in variables. In consequence a **virtual sensor reading** v is obtained (2.2).

Let the initial state of an execution of a multi-step instruction be labelled i_0 and the consecutive intermediate states $i = i_0 + 1, \ldots, i_m$, where i_m is the label of the terminal state. If the system has executed i steps, and is currently in intermediate state s^i, the next intermediate state of effectors e^{i+1} is computed by means of any of the **effector transfer functions** $f_{e_j}, j = 0, \ldots, 7$ from (4.4).

Three distinct purposes of monitoring can be named:

– initial condition monitoring,
– terminal condition monitoring,
– error condition monitoring.

The monitoring of the initial condition starts in the initial state and causes the system to execute a certain number of steps waiting for the initial condition to be satisfied. If the initial condition is satisfied upon initiation of the execution of the instruction, the number of steps done in this phase of the instruction execution is zero. Once the initial condition is satisfied, usually throughout the remaining steps both the error and the

terminal condition are monitored, until one of them is satisfied. Satisfying either the terminal or the error condition brings the execution of a motion instruction to the terminal state (Fig. 4.2). Here again, the number of executed steps depends on the moment when one of these conditions will be satisfied.

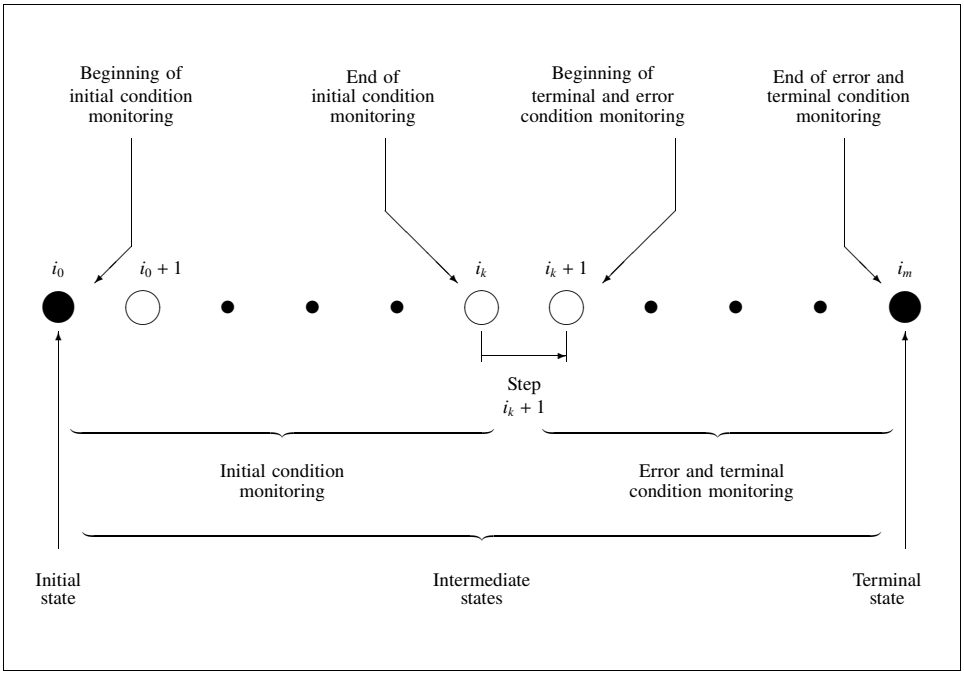

Figure 4.2. Monitoring the execution of a motion instruction

The future intermediate states can be controlled, that is either modified in relation to the planned states or generated. In both cases virtual sensor readings are utilized.

The variables are divided according to the role they play in the control of the system:

v – the state of variables containing the virtual sensor reading,

c_{vv} – the state of remaining variables.

$$c_v^i = < v^i, c_{vv}^i > \tag{4.5}$$

It follows from (4.3) that the state of variables after the execution of the i-th step is:

$$c_{vv}^i = \begin{cases} f'_{v_0}() = const \\ f'_{v_1}(c_v^{i-1}) \\ f'_{v_2}(r^{i-1}) \\ f'_{v_3}(e^{i-1}) \\ f'_{v_4}(c_v^{i-1}, r^{i-1}) \\ f'_{v_5}(e^{i-1}, r^{i-1}) \\ f'_{v_6}(c_v^{i-1}, e^{i-1}) \\ f'_{v_7}(c_v^{i-1}, r^{i-1}, e^{i-1}) \end{cases} \tag{4.6}$$

From (4.3) it is obvious that only $f_{v_j}, \quad j = 2,4,5,7$ can be used as real sensor aggregating functions. The virtual sensor reading is then:

$$v^i = \begin{cases} f''_{v_2}(r^{i-1}) \\ f''_{v_4}(c_v^{i-1}, r^{i-1}) \\ f''_{v_5}(e^{i-1}, r^{i-1}) \\ f''_{v_7}(c_v^{i-1}, r^{i-1}, e^{i-1}) \end{cases} \tag{4.7}$$

in which $f''_{v_j}, \quad j = 2,4,5,7$ is a more precise statement of (2.2).

In the case of **initial condition monitoring** the system executes consecutive steps waiting for the initial condition to be satisfied, so that the motion can proceed. The most general form of initial condition monitoring is obtained by taking into account (4.4):

$$e^{i+1} = \begin{cases} e^i = e^{i_0} & \text{when } v^i \notin V^I \\ f_{e_j}(\bullet), \quad j = 0,1,4,6 & \text{when } v^i \in V^I \end{cases} \quad \text{for } i = i_0, \ldots, i_k \tag{4.8}$$

where:
i_k is the number of the first step in which $v^{i_k} \in V^I$,
$V^I \subset V$ is the sub-space of virtual sensor readings satisfying the initial condition,
• stands for adequate arguments in functions $f_{e_j}(\bullet), \quad j = 0,1,4,6$.

Terminal condition monitoring consists in detecting a virtual sensor reading that satisfies the terminal condition. Again from (4.4) the most general form follows:

$$\begin{cases} e^{i+1} & = f_{e_j}(\bullet), \quad j = 0,1,4,6 & \text{when } v^i \notin V^T \\ e^i & = e^{i_m} \in E^T & \text{when } v^i \in V^T \end{cases} \quad \text{for } i = i_0, \ldots, i_m \tag{4.9}$$

where:
$V^T \subset V$ is the sub-space of virtual sensor readings satisfying the terminal condition,
$E^T \subset E$ is the effector state sub-space in which the terminal condition is satisfied, and
• stands for adequate arguments in functions $f_{e_j}(\bullet), \quad j = 0,1,4,6$.

Error condition monitoring consists in detecting a virtual sensor reading that satisfies the error condition:

$$\begin{cases} e^{i+1} & = f_{e_j}(\bullet), j = 0,1,4,6 & \text{when } v^i \notin V^E \\ e^i & = e^{i_m} \in E^E & \text{when } v^i \in V^E \end{cases} \quad \text{for } i = i_0, \ldots, i_m, \; i_m \le i_{m_*} \tag{4.10}$$

where:
$V^E \subset V$ is the sub-space of virtual sensor readings satisfying the error condition,
$E^E \subset E$ is the effector state sub-space in which the error condition is satisfied,
• stands for adequate arguments in functions $f_{e_j}(\bullet), \quad j = 0,1,4,6$.

It should be noted that in this case the instruction terminates its execution either when the error condition is satisfied or when a fixed number of steps $(i_{m_*} - i_0)$ is executed. Sometimes, instead of V^E, its complement $\overline{V^E}$ is used ($V^E \bigcup \overline{V^E} = V, \quad V^E \bigcap \overline{V^E} = \emptyset$).

$$\begin{cases} e^{i+1} & = f_{e_j}(\bullet), j = 0,1,4,6 & \text{when } v^i \in \overline{V^E} \\ e^i & = e^{i_m} \in E^E & \text{when } v^i \notin \overline{V^E} \end{cases} \quad \text{for } i = i_0, \ldots, i_m, \quad i_m \le i_{m_*} \tag{4.11}$$

It is sometimes easier to detect **non-error conditions** than to detect errors. When the virtual sensor reading is within bounds, the instruction execution proceeds.

Both error and terminal condition can be monitored simultaneously. In this case (4.9) and (4.10) have to be superimposed:

$$\begin{cases} e^{i+1} & = f_{e_j}(\bullet), j = 0, 1, 4, 6 \quad \text{when } v^i \notin V^T \bigwedge v^i \notin V^E \\ e^i & = e^{i_m} \in E^T \cup E^E \quad\quad\quad\quad \text{when } v^i \in V^T \bigvee v^i \in V^E \end{cases} \quad \text{for } i = i_0, \ldots, i_m \tag{4.12}$$

The **control of future intermediate states**, in the general case, can be expressed as:

$$\begin{cases} e^{i+1} & = f_{e_7}(e^i, c_v^i, r^i) \quad \text{when } v^i \notin V^E \bigwedge v^i \notin V^T \\ e^i & = e^{i_m} \in E^E \cup E^T \quad \text{when } v^i \in V^E \bigvee v^i \in V^T \end{cases} \quad \text{for } i = i_0, \ldots, i_m \tag{4.13}$$

The control is usually combined with monitoring either error or terminal condition or both. Otherwise, a fixed number of steps has to be executed ($i_m = const$, i_m does not depend on v^i).

If (4.8), (4.9), (4.10), and (4.13) are combined, the execution of the most general motion instruction assumes the following form (Fig. 4.2):

$$\begin{cases} e^{i+1} & = e^i = e^{i_0} & \text{when } v^i \notin V^I, & \text{then } i = i_0, \ldots, i_k \\ e^{i+1} & = f_{e_7}(e^i, c_v^i, r^i) & \text{when } v^i \notin V^E \bigwedge v^i \notin V^T, & \text{then } i = i_k, \ldots, i_m \\ e^i & = e^{i_m} \in E^T & \text{when } v^i \in V^T, & \text{then } i = i_m \\ e^i & = e^{i_m} \in E^E & \text{when } v^i \in V^E, & \text{then } i = i_m \end{cases} \tag{4.14}$$

The following subsection will utilise the above formalism to describe the semantics of some motion instructions taken from several well-known RPLs.

4.3 Examples of RPL motion instructions

The most general RPL instruction semantics assumes the following form:

$$\begin{cases} c_d^{i+1} & = f_{d_8}(c_d^i, c_v^i, r^i, e^i) & & \\ c_{vv}^{i+1} & = f'_{v_7}(c_v^i, r^i, e^i) & & \\ v^{i+1} & = f''_{v_4}(c_v^i, r^i) & \text{until } v^i \notin V^I, & \text{then } i = i_0, \ldots, i_k \\ v^{i+1} & = f''_{v_7}(c_v^i, r^i, e^i) & \text{after } v^i \notin V^I, & \text{then } i = i_k, \ldots, i_m \\ e^{i+1} & = e^i = e^{i_0} & \text{when } v^i \notin V^I, & \text{then } i = i_0, \ldots, i_k \\ e^{i+1} & = f_{e_7}(e^i, c_v^i, r^i) & \text{when } v^i \notin V^E \bigwedge v^i \notin V^T, & \text{then } i = i_k, \ldots, i_m \\ e^i & = e^{i_m} \in E^T & \text{when } v^i \in V^T, & \text{then } i = i_m \\ e^i & = e^{i_m} \in E^E & \text{when } v^i \in V^E, & \text{then } i = i_m \end{cases} \tag{4.15}$$

This is a multi-step instruction. The program that is to be executed consists of a sequence of instructions, and so c_p implies the form of: f_{d_8}, f_{e_7}, f'_{v_7}, f''_{v_4}, f''_{v_7}.

The semantics of instruction (4.15) are as follows. Initially the system waits for the initial condition to be satisfied. The structure of this condition depends on the form of f''_{v_4}. Whilst waiting for an initial condition to be satisfied the state of effectors does not change ($e^{i+1} = e^i = e^0$), hence f''_{v_4} is used and not f''_{v_7}. During the waiting period, as well as during the execution of later steps, certain measurements can be processed and stored.

This depends on the form of f'_{v_7}. Once the initial condition is fulfilled, the monitoring of the terminal and error condition commences. As the future effector state depends on sensor readings (f_{e_7}), the effector trajectory is either generated or modified in relation to the stored value c_{vv}. The next step to be executed is designated by f_{d_8}. Once the instruction terminal or error state is detected, f_{d_8} determines the the next instruction to be executed.

The generic form of functions $f_{d_j}, j = 0, \ldots, 8$ from (4.2) can be constrained to the following:

$$f_{d_j} : \begin{cases} \text{increments or modifies the step counter} & \text{for } i_0 \leq i < i_m \\ \text{designates the next program instruction} & \text{for } i = i_m \end{cases} \tag{4.16}$$

where i_0, i_m are the labels of the initial and terminal states respectively.

Although a single instruction with such complex semantics would suffice to build robot programs, its syntax would be very intricate. Usually, RPLs introduce several instructions with much less complex semantics. The semantics of several instructions from different existing RPLs will be defined using the introduced notation.

The VAL II MOVES <location> command, described earlier in subsubsection 17, is a multi-step, unconditional, non-storing, sensorless absolute motion instruction. This instruction causes the tool to assume the location (position and orientation) specified by its argument. The intermediate locations are computed by straight line interpolation between the initial and terminal location. Location of the tool is specified by the X, Y, Z Cartesian co-ordinates and O, A, T angles[1]. The maximum speed of motion depends on a certain global variable, set by the programmer with the SPEED <value> instruction. The semantics of this instruction is as follows:

$$\begin{cases} c_d^{i+1} & = f_{d_1}(c_d^i) & & \\ c_v^{i+1} & = c_v^i & & \\ e^{i+1} & = f_{e_1}(c_v^i) & \text{when } r_{int}^i \notin R_{int}^E & \text{for } i = i_0, \ldots, i_m \\ e^i & = e^{i_m} & \text{when } r_{int}^i \in R_{int}^E & \end{cases} \tag{4.17}$$

where r_{int} is an internal receptor reading and R_{int}^E is its error space. The VAL II system monitors the arm position and certain other safety sensors. If the system detects an error (e.g. violation of work space), the arm is stopped. These sensor readings are inaccessible to the programmer directly, so they cannot be used for virtual sensor reading evaluation. Although internal sensors, for safety reasons, are always used by motion instructions, this fact does not change the attitude towards instructions of the MOVES kind – they are sensorless instructions, because external sensors do not influence their execution, when no fatal situation[2] occurs. Function f_{d_1} increments the step counter and after executing the MOVES instruction designates the next program instruction for execution. Function f_{e_1} causes the tool tip to move along the straight line in Cartesian space and its smooth rotation. No arm configuration changes are allowed. Unfortunately, the VAL II manual gives no hint of how should function f_{e_1} be specified.

The VAL II WAIT <expression> command is a multi-step, conditional, non-storing, non-motion instruction. The execution of this instruction causes the system to wait

[1] Mutation of Euler angles [194].

[2] A situation in which program execution is aborted – the system halts.

for the <expression> to become TRUE. The form of <expression> implicates both v^i and V^I. It can contain information about several external signals (receptors), bit-wise binary and Boolean operators, relational operators, constant values and variables. The next step is designated using function f_{d_3} or f_{d_5}:

$$\begin{cases} c_d^{i+1} & = f_{d_5}(c_d^i, c_v^i, r^i) & & \\ c_v^{i+1} & = c_v^i & & \\ e^{i+1} & = e^i = e^{i_0} & \text{when } v^i \notin V^I & \text{for } i = i_0, \ldots, i_m \\ e^i & = e^{i_m} = e^{i_0} & \text{when } v^i \in V^I & \end{cases} \tag{4.18}$$

Function $f_{d_5}(c_d^i, c_v^i, r^i)$ checks if $v^i = f''_{v_4}(c_v^i, r^i) \in V^I$. If yes, then $i = i_m$ and waiting terminates, otherwise the next waiting step is executed. The state of the effectors does not change during waiting.

The AL MOVE instruction, described earlier in subsubsection 3.2, has a complex syntax, which can include motions with simultaneous application of forces, e.g.

MOVE < movable frame > TO < non-movable frame >
WITH FORCE = < value > < unit >
ALONG < versor > OF < co-ordinate frame >

This is a multi-step, conditional, non-storing, sensor, incremental motion instruction:

$$\begin{cases} c_d^{i+1} & = f_{d_7}(c_d^i, r^i, e^i) & & \\ c_{vv}^{i+1} & = c_{vv}^i & & \\ v^{i+1} & = f''_{v_7}(c_v^i, r^i, e^i) & & \text{for } i = i_0, \ldots, i_m \\ e^{i+1} & = f_{e_7}(e^i, c_v^i, r^i) & \text{when } v^i \notin V^E \bigwedge v^i \notin V^T & \\ e^i & = e^{i_m} \in E^T & \text{when } v^i \in V^T & \\ e^i & = e^{i_m} \in E^E & \text{when } v^i \in V^E & \end{cases} \tag{4.19}$$

The next step is designated according to $f_{d_7}(c_d^i, r^i, e^i)$. The termination of the execution of this instruction depends on the tool (effector – < movable frame >) reaching the vicinity of the < non-movable frame > or detection of errors. The variables do not change their state. Effector state is computed using $f_{e_7}(e^i, c_v^i, r^i)$, which takes into account the specified force that has to be exerted. The AL manual does not enable the exact specification of this function.

The TORBOL DO (relation Object1 Object2) instruction, described earlier in subsubsection 19, is a multi-step, conditional, storing, sensorless absolute motion instruction. Object1 and Object2 are treated as effectors. Figure 3.2 shows the flow diagram of the execution of this instruction, and its formal specification can be found in [164, 170, 174]. As this specification is very complex and tedious, only its general

outline will be mentioned here:

$$\begin{cases} c_d^{i+1} & = f_{d_6}(c_d^i, c_v^i, e^i) & & \\ c_{vv}^{i+1} & = f'_{v_1}(c_v^i) & & \text{for } i = i_0, \ldots, i_m \\ e^{i+1} & = f_{e_1}(c_v^i) & \text{when } v^i \notin V^E \bigwedge v^i \notin V^T & \\ e^i & = e^{i_m} & \text{when } r_{int} \in R^E & \end{cases} \tag{4.20}$$

where, as in the case of (4.17), r_{int} is an internal receptor reading and R_{int}^E is its error space. For the same reasons the DO instruction is treated as a sensorless motion instruction. This instruction designates the next step on the basis of information contained in the database storing the state of the environment c_v^i and the state of the effectors e^i – namely the gripper – $f_{d_6}(c_d^i, c_v^i, e^i)$. The execution of this instruction brings about a change in the contents of the database, so c_{vv} is modified, hence it is a storing instruction – $f_{v_6}(c_v^i)$. Finally, no sensors (besides internal sensors) are employed and each motion step is computed in absolute co-ordinates, so $f_{e_1}(c_v^i)$ is used.

The exact form of functions $f_{e_j}, j = 0, \ldots, 7$ depends on how the effector state is expressed and on the employed space and method of interpolation. Usually, cubic or quintic splines or a linear function with parabolic blends is used as a reference trajectory [24].

The introduced formalism takes into account all of the system components (i.e. effectors, receptors and the control subsystem), and states the instruction semantics without predetermining any particular implementation technique. Nevertheless, each of the functions $f_e(\bullet)$, $f_v(\bullet)$, $f_d(\bullet)$ specifies a certain portion of the code (usually a procedure or function) that together implement the instruction. If these functions are incorporated into flow-diagrams, which show what the sequence of the execution of these portions is, the implementation of the instruction is simple and less prone to errors. This technique has been used by the author in the implementation of: ROPAS, ROOPL, TORBOL and RORC. It is especially well suited to the implementation of robot specific libraries of procedures and run-time systems of specialised robot languages.

5 Implementation of robot programming languages

5.1 Methods of implementing robot programming languages

There are three methods of implementing RPLs:

- as a specialised language,
- as an enhancement of an existing CPL,
- as a robot specific library of procedures coded in a universal CPL.

Implementation of a specialised language is a very laborious task. First the definition of the language (syntax, semantics) has to be elaborated. Usually, it turns out that such a language has to possess all the properties of a CPL plus robot specific instructions and data types, which renders it very complex, both to master and even more so to

implement. VAL II [194, 12], WAVE [109], RAPT [2, 176], AL [100, 176], TORBOL [162, 168, 173, 176], SRL [12] and many other RPLs were implemented in this way.

As an RPL has to have nearly all the constructs attributed to a CPL, it seems that the second of the listed ways of implementation would be more appropriate. Unfortunately, very seldom can the definition of a CPL be enhanced – mainly because the code of the compiler is available only in an executable (non-modifiable) form (the source code is a trade secret), and so this way is usually closed to robotics researchers.

The last method is the cheapest, both in terms of funds necessary for the development of an RPL and the time spent on this development. Only robot specific procedures have to be coded, while all the mechanisms of a CPL are still available to a programmer. Moreover, no modification to the compiler or interpreter of the language is necessary. The only drawback is that robot specific instructions are a bit more cryptic (procedures with adequate parameters have to be used instead of explicit robot instructions with appropriate arguments). If library creation is chosen as the means of implementing an RPL, a CPL that will be the foundation, and the programming methodology, still remain to be selected. For instance PASRO [11, 12], POLROB [73], ROPAS [183] and ROOPL [177, 180] are submerged in Pascal [190], and C [69, 192] is the basis for RCCL [53] and ARCL [23] and the language of the Research-Oriented Robot Controller RORC [175, 181].

Sections 5.2 and 5.3 describe two manipulator level robot programming languages implemented by the author as libraries of procedures coded in Pascal (ROPAS) and in object-oriented extension of Pascal (ROOPL). They were not described in section 3, because the definitions of these languages are inseparable from their implementation platforms. The definition of their instructions is the code (in Pascal) of the procedures implementing these instructions. In the case of specialised languages the definition of instructions can be readily separated from their implementation, so this section will only describe the implementation aspects of TORBOL, which is a specialised language.

5.2 ROPAS

ROPAS (RObot PAScal) is a library of robot specific data types and procedures coded in Pascal [190], that can be used in programs generating, modifying, and executing robot arm trajectories. The library does not have a closed form, so new procedures can be readily added.

Unlike PASRO [11], which is also a library of Pascal procedures for controlling robots, ROPAS defines two assembly specific data types:

- homogeneous transform matrix (pointed to by MatrixPtr) which can represent a co--ordinate frame in relation to some other reference frame, a transformation between co-ordinate frames, or a translation and rotation operator. As both translation and rotation are specific cases of a homogeneous transformation, no other geometric data types are necessary,
- path point (pointed to by PathPointPtr) being an element of a two-way linked list of frames. Path points are used to describe paths in space relative to a certain reference co-ordinate frame. Each path point carries information about the X, Y, Z Cartesian co-ordinates, as well as the orientation of the frame traversing the path. Moreover, pointers to the previous and next path point are supplied.

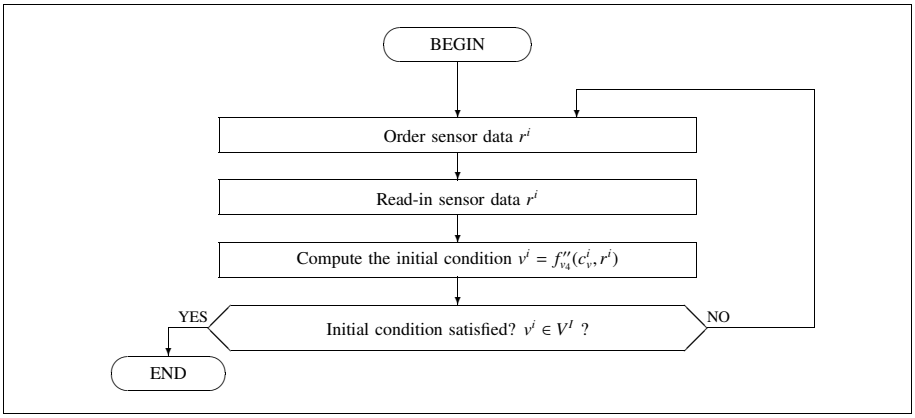

Figure 5.1. Initial condition monitoring

PASRO introduces more geometric data types (vector, rotation and frame), and so a large portion of its procedures deals with mathematical manipulation of these data types (makevector, vabs, vadd, vsub, vmul, vdiv, vrot, vdot, vcross, makerotation, rotrot, rotaxis, rotangle, makeframe, setframe, frametransl, framerot, transframe, framerel, frameinv). The sensor data processing and utilization has not been a priority of the PASRO authors. PASRO has several very simple procedures supporting communication via an interface to external devices or sensors:

sigon (ad: integer) – sets a port at address ad,
sigoff (ad: integer) – resets a port at address ad,
sigin (var bout: Boolean; ad: integer) – reads in the value in the port at address ad; if high, bout is assigned true, else it is assigned false,
anout (v, ad: integer) – transmits the value of v to a D/A converter via the port at address ad,
anin (var v: integer; ad:integer) – stores the data from an A/D converter at port ad in variable v.

In the case of ROPAS both sensor data aggregation and utilization procedures form the bulk of the library.

ROPAS has three distinct elementary motion instructions, each implemented as a Pascal procedure. The first causes the robot to wait for the initial condition to be satisfied (Fig. 5.1). Its semantics is given by (4.8).

```
procedure Wait ( InitialCondition: ConditionTemplate;
                 Trajectory:       PathPointPtr );
```

where ConditionTemplate is a procedural type defining a template which decides whether the initial condition is satisfied ($v^i \in V^I$). It takes into account the designated virtual sensors. Once the initial condition is satisfied, the sequence of planned next effector states e^*_* (Trajectory) is computed (where e_* is a single **planned effector state**).

The second motion instruction monitors the error and terminal conditions (Fig. 5.2). Its semantics is given by (4.12).

```
procedure MoveMonitoring ( Tool:                MatrixPtr;
```

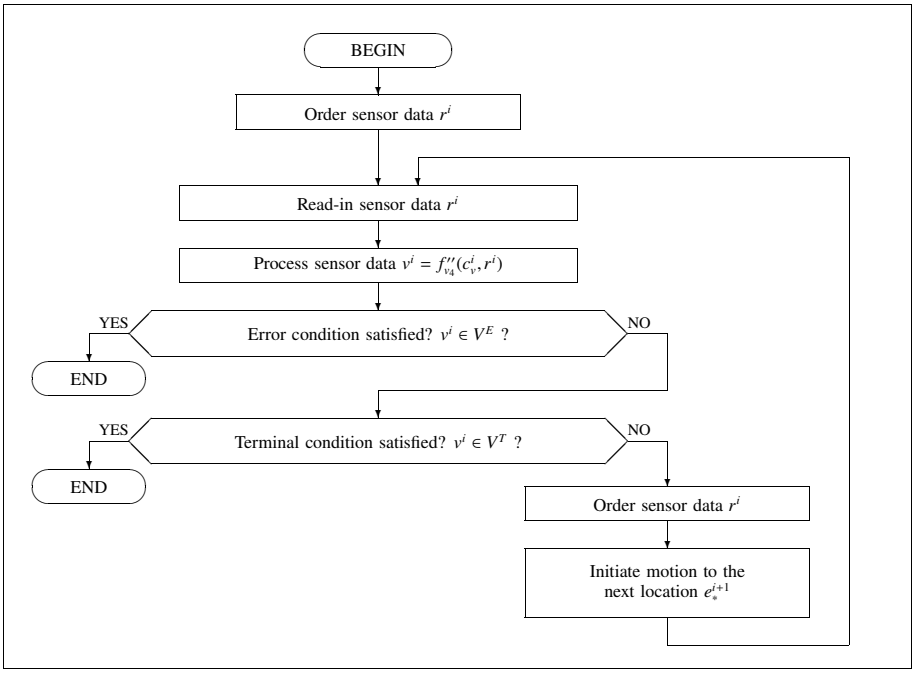

Figure 5.2. Error and terminal condition monitoring

```
                    Trajectory:          PathPointPtr;
                    TerminalCondition:   ConditionTemplate;
                    ErrorCondition:      ConditionTemplate );
```

where Tool is the pointer to the co-ordinate frame affixed to the end-effector in relation to the manipulator flange, and Trajectory is the pointer to the list of frames that define the planned end-effector state (i.e. trajectory – e^*_*). The remaining parameters are as before. This procedure causes the tool to traverse the specified trajectory. During the motion sensors gather information from the environment, so that the terminal and error conditions can be evaluated. Motion terminates if either of these conditions is satisfied or all of the trajectory is traversed.

If MoveMonitoring and Wait instructions are executed in a sequence, a compound instruction monitoring initial, error and terminal condition results.

The third instruction simultaneously monitors terminal end error conditions and controls the future intermediate state (Fig. 5.3). Very rarely is control of future states exercised without monitoring these conditions, so it was decided that future state control instruction will not be introduced in its pure form (i.e. without monitoring). Its semantics is defined by (4.13).

```
procedure Move ( Tool:               MatrixPtr;
                 Trajectory:         PathPointPtr;
                 TerminalCondition:  ConditionTemplate;
                 ErrorCondition:     ConditionTemplate;
```

TransferFunction: TransferTemplate);

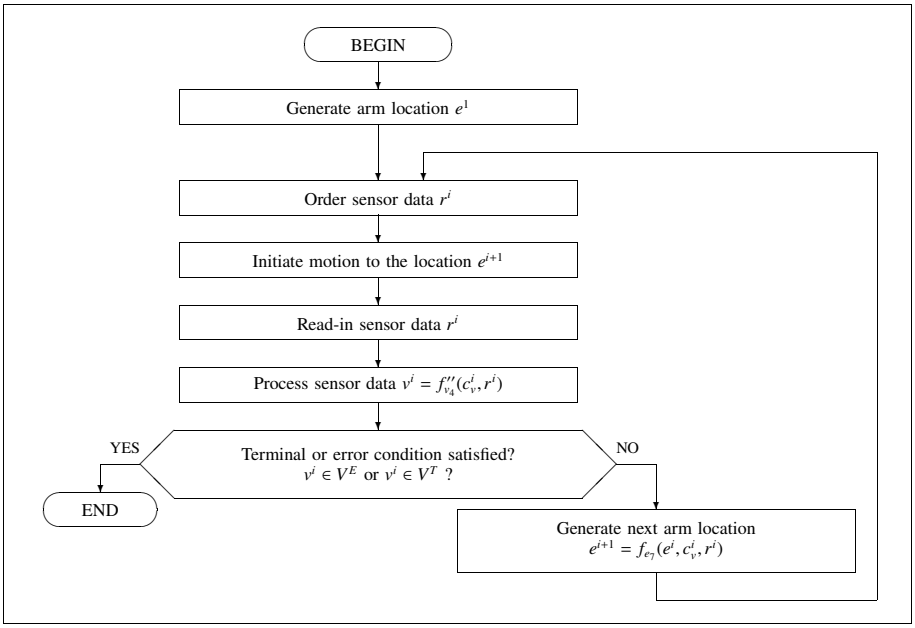

Figure 5.3. The monitoring of current states and control of future states

where TransferTemplate is a procedural type defining the transfer function computing the next state of the end-effector on the basis of Trajectory (i.e. e^*_*) and the designated virtual sensors v. The remaining parameters are as above. In this case in each intermediate path point the terminal and error conditions are evaluated and the new path point is computed using the transfer function. The transfer function can generate a path point based only on the knowledge of the current state of the effectors and sensor readings or it can take into account an off-line created trajectory (i.e. e^*_*) – modifying it using sensor data.

The introduction of procedural parameters reduces significantly the number of types of motion instructions that are defined in a robot programming language. For instance, in PASRO different types of interpolation between the current robot position and the goal position need separate instructions. In ROPAS the same instruction (Move) is invoked with different TransferFunction as its parameter, significantly increasing the readability of the resulting code.

If the Wait is combined with the Move instruction in a sequence, the most general compound motion instruction monitoring all three conditions and influencing (modifying or generating) future effector states results.

The structure of the system executing ROPAS programs is shown in Fig. 5.4. A ROPAS program is executed on an IBM-PC class host computer. It generates robot motion commands that are being interpreted by a VAL II [194] program running on a robot control system computer. Both computers are connected by an RS–232 serial interface. The VAL II control system causes the motion of a PUMA–560 robot arm. The

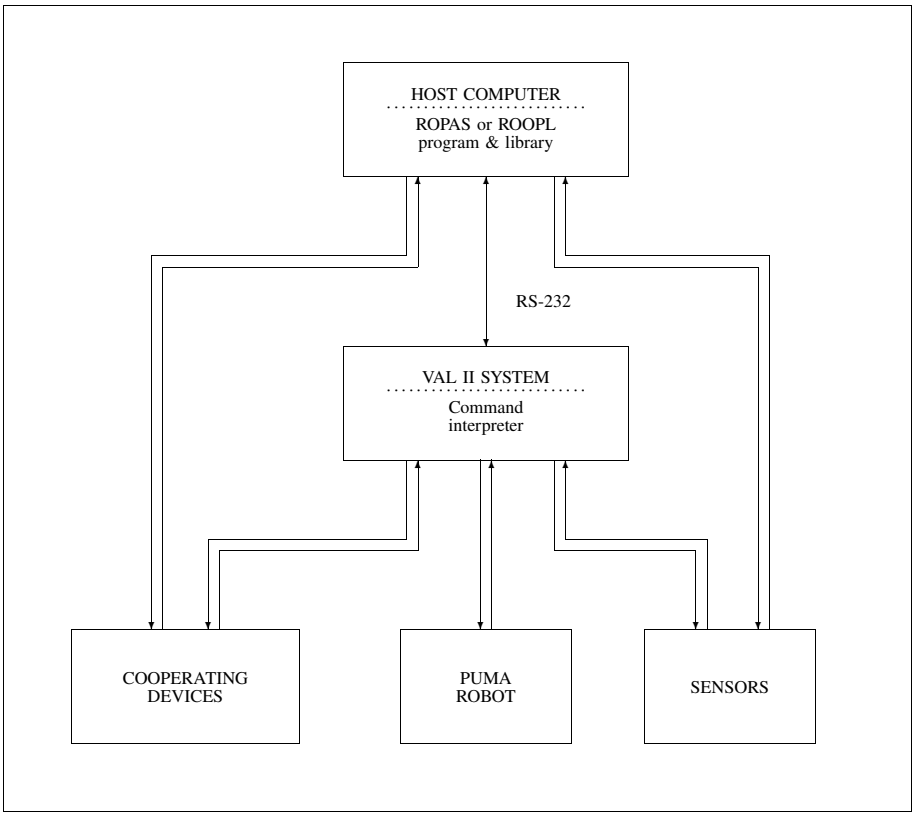

Figure 5.4. Structure of the system executing ROPAS and ROOPL programs

sensors can be directly connected either to the host computer or to the control computer. In the latter case, data obtained from sensors are transmitted through the RS–232 interface to the host computer for processing. The RS–232 is also used for transmitting to the host computer the information about the current state of the arm (e.g. about motion termination or the arm location). ROPAS programs can also control cooperating devices connected either to the host computer directly or indirectly through the control computer.

The method of processing and executing ROPAS programs is presented in Fig. 5.5. The source program is written using any text editor. Next the source program is compiled by a Pascal compiler. The resulting object code files and the ROPAS library module are linked, and so the executable code is obtained. This code is run on the host computer. Simultaneously, a program called a command interpreter, written in VAL II, is run on the control computer. After an automatic synchronization phase through the RS–232 interface, the robot task initially coded in ROPAS is executed. The command interpreter constantly waits for motion commands and status sensor requests from the host computer. In response to these commands it initiates the execution of motions and sends back the requested dat a and information about motion termination.

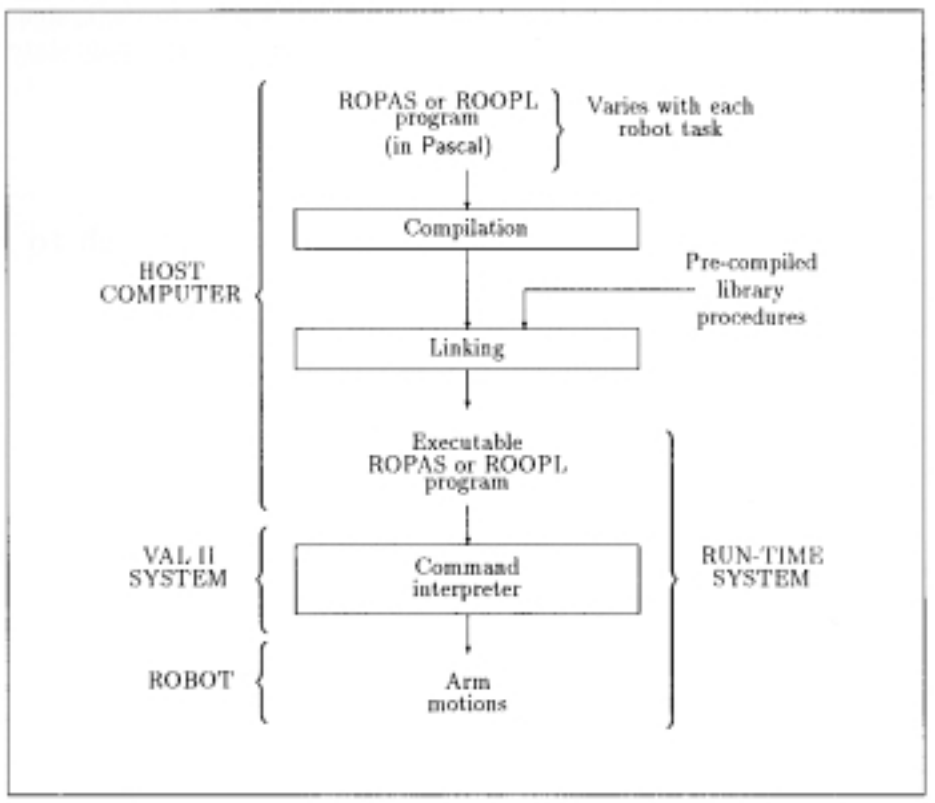

Figure 5.5. Method of processing a ROPAS or ROOPL program

In ROPAS, out of several possible sets of motion instructions, the one containing only three instructions was chosen. Nevertheless, the three instructions cover all the possible uses of sensors for motion monitoring and control. As it was shown, the set can be reduced to only one instruction, but then very complex semantics result. It was also shown that at the other extreme the set could have many more instructions but then they would be much more elementary. In either case the readability of resulting programs would decrease. So a compromise solution was taken.

All functions (i.e. f_v'' – aggregating functions, f_e – transfer functions) and all conditions were coded as separate procedures and are invoked by the Wait, MoveMonitoring and Move instructions as procedural variables further increasing the readability of the ROPAS code.

Due to the low transmission rate of the RS–232 interface, the consecutive end-effector states (positions) that were generated or modified in relation to the Trajectory list could not be too near each other. Otherwise, either very slow motions had to be executed or a jerky motion resulted. This drawback could be eliminated either by using a much faster and parallel interface or by introducing interpolation in each motion step, and reducing the number of trajectory sensory updates. In the case of the current system the second approach was followed. The end-effector state was up-dated by taking into account sensor readings only at the positions generated by the transfer function (those were

quite far apart) and between those positions either joint space or Cartesian-Euler interpolation was used. Obviously, this is one remedy to the problem, but the real solution is obtained by utilizing a fast parallel interface.

It should be noted that some robot programming systems restrict the full utilization of sensors by:

- reading sensors only between motion instructions – in effect enabling only global decision making, and so making on-line trajectory modification and monitoring impossible,
- implementing only condition monitoring – in effect making the influence over future intermediate states impossible.

Even if a language has full sensor utilization potential, programmers tend to confine themselves to monitoring only. In the case of complex tasks (e.g. assembly tasks) error condition monitoring followed by error recovery techniques are frequently used. If such a passive method of programming is followed, the system is allowed to make an error, the error is detected, and finally a recovery procedure is tried. This results in considerable loss of time. Because of that, error recovery is sometimes given up altogether, and the problem is solved by discarding the material – especially if the material causing the problem is much cheaper than the time lost on recovering from the error. It is much better to employ an active method of programming in which errors are avoided by correcting them prior to the system condition deteriorating to an error condition (i.e. future intermediate state control).

5.3 ROOPL

5.3.1 Object-oriented and structured approach to programming

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

Encapsulation — treating *data* and *code* operating on it as one entity – an *object*.

Inheritance — defining a hierarchy of *objects* in which each *descendant object* acquires all the properties of the *ancestor objects* (access to *data* and *code* of the *ancestors*) and receives some new properties specific to the newly created *object*.

Polymorphism — using the same name for an action that is carried out on different *objects* related by *inheritance*. The action is semantically similar, but it is implemented in a manner appropriate to each of the individual *objects* of the hierarchy.

Some of the CPLs were created as OOP languages (e.g. Loglan [8]), others which had been originally used only as structured programming tools were enhanced by adding OOP mechanisms (e.g. Pascal [190], C++ [192]).

OOP methodology assumes that certain abstract *objects* will be defined by the programmer. These *objects* have their properties (*data*) and exhibit behaviours (*methods*). The program is written in terms of *objects* behaving in such a way so as to change their properties, i.e. applying *methods* to change *data*. To make this clearer, an example follows. Let a screw be the *object*. One of its many characteristics is its location in space (*data*). The screw can move in space (movement is its behaviour). The programmer commands the screw to change its position, and so applies its position changing *method* to its *data*. As the result of applying this *method*, the screw will be transported to some other location.

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

This monograph describes the application of OOP methodology to the creation of a manipulator level RPL (*object* library to be strict). For this purpose a version of Pascal language possessing OOP enhancements [190] was used.

5.3.2 General information about the ROOPL library

To use ROOPL, a Pascal program invoking library *objects* and their *methods* has to be written. At its beginning it should contain the following clause: uses roopl;

A homogeneous transform matrix type representing a dextrorotatory set of orthogonal unit axes (co-ordinate frame) – matrix, and a pointer type – MatrixPtr to such a frame, are defined as supplementary data types. Since a homogeneous transform matrix can also represent transformations (translation and rotation), no other robot specific data type needs to be introduced.

Two types of *objects* are defined by the ROOPL library: frame and segment. The first represents a homogeneous transform representation of a co-ordinate frame. The later is a *descendant* type of the former and describes the approach path segment to the *ancestor* frame. The *objects* are defined in the following way:

```
frame = object
  location:     MatrixPtr;
  constructor   Create(x,y,z,fi,theta,psi: real);   { Create the frame using Cartesian-Euler
                                                      description }
  destructor    Destroy; virtual;                   { Release memory }
  procedure     Copy(F: frame);                     { Copy F.location into self.location }
  procedure     Invert;                             { Invert self.location }
  procedure     LeftMultiply(F: frame);             { left multiply self.location by F.location }
  procedure     RightMultiply(F: frame);            { right multiply self.location by F.location }
  function      Equal(F: frame): boolean;           { Compare self.location and F.location }
  procedure     WriteLocation;                      { Write out self.location }
```

```
end; { frame }


segment = object (frame)
  motion:           MotionType;   { type of motion to be executed along the segment
                                    }
  Sensor_LSB:       byte;         { VAL input number that is to be treated as a LSB }
  Sensor_reading:   integer;      { value obtained from the sensor }
  Error:            byte;         { error code (e.g. errors may occur during
                                    transmission) }

  constructor     Create(x,y,z,fi,theta,psi: real; SegMotion: MotionType;
                                                   SensLSB: byte);
                  { Create segment using the listed arguments }
  destructor      Destroy; virtual;                  { Release memory }
  function        SensorData: byte;                  { Read sensor data byte }
  function        ErrorOccurred: Boolean;            { Get error }
  procedure       WriteError;                        { Write out error }
  function        SegmentType: MotionType;           { Get segment type }
  procedure       Copy(S: segment);                  { Copy S into self }
  procedure       InitRobot(A6T: frame);             { Initialize the robot }
  procedure       QuitRobot(A6T: frame);             { Deactivate the robot }
  function        GetSensorData: byte;               { Get sensor reading }
  function        SensorNumber: byte;                { Get sensor number }
  procedure       SetSensorNumber(sn: byte);         { Set sensor LSB number }
  procedure       SetSegmentType(m: MotionType);
                  { Set segment type }
  procedure       SetLocation(x,y,z,fi,theta,psi: real);
                  { Set location field to Cartesian-Euler representation }
  procedure       Homogeneous_to_Euler(var x,y,z,fi,theta,psi: real);
                  { Transform homogeneous to Cartesian-Euler representation }
  procedure       AttractPumaTool(A6T: frame);
                  { Move robot; A6T – Flange to Tool transform }
end; { segment }
```

The *methods* defined in the frame *object* (Copy, Invert, LeftMultiply, RightMultiply, Equal) perform the obvious homogeneous matrix operations. The first argument of the operations is the location *field* of the *object* and the second (where present) is defined by the *method*'s actual parameter.

There is only one, but general, *method* of moving the robot. The AttractPumaTool *method* applied to a segment *object* causes the robot to move towards (be attracted by) the co-ordinate frame, being one of the *fields* of the *object*. The type of motion depends on the contents of the motion *field*. Eight different kinds of motions can be performed – defined by enumerated type MotionType.

PTP_JointInterpolated_NoSensors causes joint interpolated motion without using sensors; confirmation is sent when the motion terminates

CP_JointInterpolated_NoSensors causes joint interpolated motion without using sensors; confirmation is sent when the motion is initiated

PTP_JointInterpolated_WithSensors causes joint interpolated motion with sensor feedback; confirmation is sent when the motion terminates

CP_JointInterpolated_WithSensors causes joint interpolated motion with sensor feedback; confirmation is sent when the motion is initiated

PTP_Cartesian_NoSensors causes Cartesian interpolated motion without using sensors; confirmation is sent when the motion terminates

CP_Cartesian_NoSensors causes Cartesian interpolated motion without using sensors; confirmation is sent when the motion is initiated

PTP_Cartesian_WithSensors causes Cartesian interpolated motion with sensor feedback; confirmation is sent when the motion terminates

CP_Cartesian_WithSensors causes Cartesian interpolated motion with sensor feedback; confirmation is sent when the motion is initiated

In the case of **PTP** (point to point) motion, its termination is signalled, while in the case of **CP** (continuous path) motion its initiation is signalled by the VAL II command interpreter. A straight line either in the joint angle co-ordinates or in Cartesian-Euler space can be used while approaching the goal frame. If sensors are used during the motion, the confirmation byte carries the information about the obtained sensor reading. Otherwise, an asterisk is sent as confirmation token. Sensor reading is obtained from the eight consecutive VAL inputs, starting at Sensor_LSB (inclusive).

Any *method* that can finish its execution without performing its task causes the Error *field* of a segment to change its value to non-zero. This *field* should be checked whenever executing an action that can result in an error (e.g. transmission error).

Other *methods* defined in segment manipulate the *data fields* of this *object* (e.g. read them). It is easier for the programmer to describe frames as three Cartesian co--ordinates of the origin and three Euler angles of orientation, so adequate *methods* for transforming the internal format into this representation and vice versa are supplied (Homogeneous_to_Euler, SetLocation).

As can be seen from the above definitions, nearly all the actions are performed by executing appropriate *methods* on the two supplied *object* types (frame and segment). Only gripper closing and opening is done by procedures:

```
procedure Grasp ( var Err: byte );
procedure Release ( var Err: byte );
```

The robot program consists of sequences of *methods* executing actions on *objects*. Besides these the programmer is free to use any Pascal statements.

5.3.3 Example

A fragment of a program executing a path that is generated on-line will be presented as an illustration of a program coded in ROOPL. In the case of sensor guided motions each new location depends on the value obtained through a set of binary sensors connected to the VAL system. Eight kinds of motion can be used by selecting one from each of the three following pairs: (PTP, CP), (joint interpolated, Cartesian interpolated), (with sensors, without sensors). The *object* Seg is the definition of the current path segment. It is modified for each subsequent motion step. As a result of this program the operator can lead the robot by exerting forces on the sensor to a goal location that has non-decreasing co-ordinates in relation to the current location (this assumption was made to keep the example brief). The sensor can be mounted near the tool tip or be placed in any other area.

```
{ Execute 15 motion steps }
for i := 1 to 15 do
  begin

    { Create the next location taking into account the sensor data }
      case Seg.SensorData of
    { Sensor supplies a number: 0 – 7 }
       0:  begin   { do nothing }                                          end;
       1:  begin   x := x + xstep;                                         end;
       2:  begin                      y := y + ystep;                      end;
       3:  begin   x := x + xstep;    y := y + ystep;                      end;
       4:  begin                                        z := z + zstep;    end;
       5:  begin   x := x + xstep;                      z := z + zstep;    end;
       6:  begin                      y := y + ystep;   z := z + zstep;    end;
       7:  begin   x := x + xstep;    y := y + ystep;   z := z + zstep;    end;
      end { case Seg.SensorData }
      Seg.SetLocation ( x, y, z, PI/2, PI/2, 0 );

    { Execute the current step }
      Seg.AttractPumaTool ( A6T );

    { Check if an error occurred }
      if Seg.ErrorOccurred
        then
          begin
            Seg.WriteError;
            Seg.Destroy;
            halt;
          end;

  end; { for }
```

5.3.4 Implementation

ROOPL (Robot Object-Oriented Pascal Library) is a library of *objects* and *methods*, coded in an OOP version of Pascal [190], that can be used in programs generating, modifying and executing robot arm trajectories.

The structure of the system executing ROOPL programs is shown in Fig. 5.4. A ROOPL program is executed on an IBM-PC class host computer. It generates robot motion commands that are being interpreted by a VAL II [194] program running on a robot control system computer. Both computers are connected by an RS-232 serial interface. The VAL II control system causes the motion of a PUMA-560 robot arm. The sensors can either be directly connected to the host computer or to the control computer. In the latter case, data obtained from sensors is transmitted through the RS-232 interface to the host computer for processing. The RS-232 is also used for transmitting to

the host computer the information about the current state of the arm (e.g. about motion termination or the arm location). ROOPL programs can also control cooperating devices connected either to the host computer directly or indirectly through the control computer.

The method of processing and executing ROOPL programs is presented in Fig. 5.5. The source program is written using any text editor. Next the source program is compiled by a Pascal compiler. The resulting object code files and the ROOPL library module are linked, and so the executable code is obtained. This code is run on the host computer. Simultaneously, a program called command interpreter, which has been written in VAL II, is run on the control computer. After an automatic synchronisation phase through the RS-232 interface, the robot task initially coded in ROOPL is executed. The command interpreter constantly waits for motion commands and sensor status requests from the host computer. In response to these commands it initiates the execution of motions and sends back the requested dat a and information about motion termination.

5.4 Implementation of TORBOL

TORBOL programs are subjected to a two-phase compilation process (Fig. 5.6). During the first phase TORBOL programs are compiled into Pascal programs. As a result,

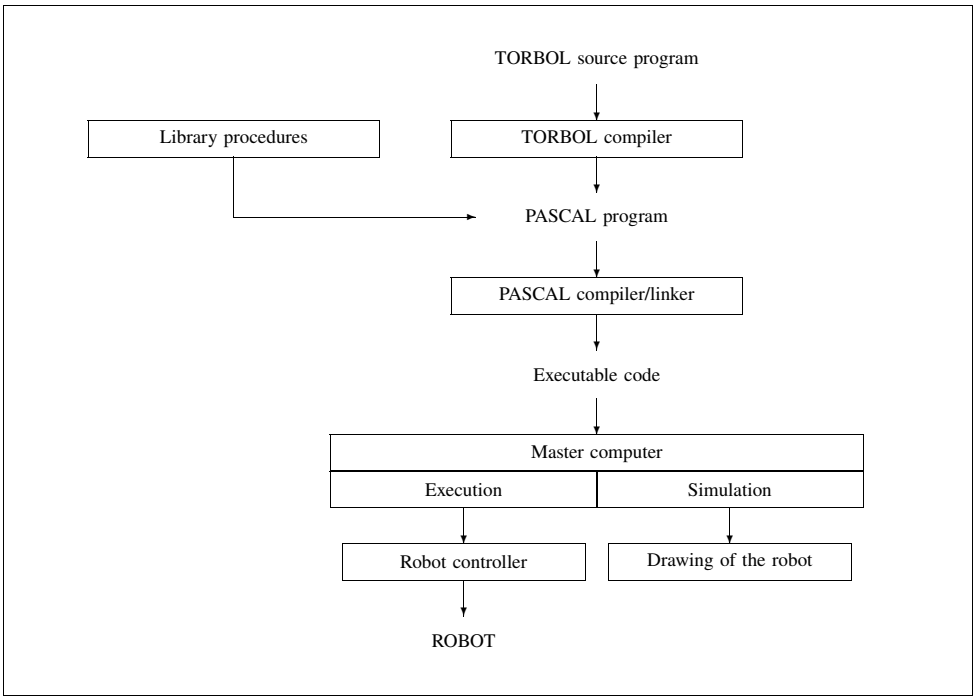

Figure 5.6. Translation of TORBOL programs

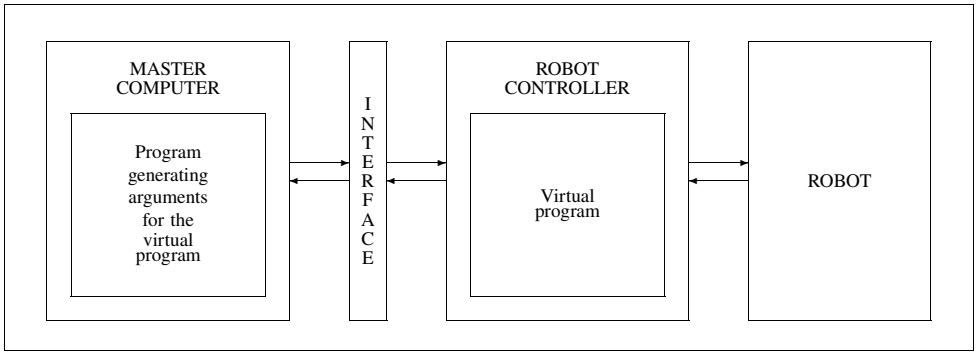

Figure 5.7. The WUT robot system executing TORBOL programs

Label	Instr. code	Argument	Comment
↓	↓	↓	↓
00	OUT	0	signal that the motion is not finished
10	MOVE	X	move the TCP to the location X
20	GRIP	Y	open or close the gripper
30	OUT	1	signal that the motion is finished
40	GOTO	L	jump to the label L
50	OUT	0	signal that the motion is not finished
60	MOVE	X	move the TCP to the location X
70	GRIP	Y	open or close the gripper
80	OUT	1	signal that the motion is finished
90	GOTO	L	jump to the label L

00	OUT	0	0	0	0	0	0
10	MOVE	X_1	X_1	X_3	X_3	X_5	X_5
20	GRIP	Y_1	Y_1	Y_3	Y_3	Y_5	Y_5
30	OUT	1	1	1	1	1	1
40	GOTO	40	50	40	50	40	50
50	OUT	0	0	0	0	0	0
60	MOVE	X	X_2	X_2	X_4	X_4	X_6
70	GRIP	Y	Y_2	Y_2	Y_4	Y_4	Y_6
80	OUT	1	1	1	1	1	1
90	GOTO	00	90	00	90	00	90
		↑	↑	↑	↑	↑	↑
		first motion	second motion	third motion	fourth motion	fifth motion	sixth motion

Figure 5.8. Virtual program and the successive modifications of its arguments in the first six motions

a Pascal program consisting of the definition of the virtual environment data-base, assignments of initial values to the attributes, and calls to the library procedures executing TORBOL instructions, is obtained. During the second phase this program is compiled by the Pascal compiler and linked with the above-mentioned library procedures. The set of library procedures that are linked-in is different when the executable code graphically simulates the execution of the TORBOL program and when this program is to be executed on the real robot system.

The robot system executing TORBOL programs consists of: an IRb–6 robot, its standard controller, a master computer (an IBM-PC/AT compatible) and a custom designed interface between the computer and the controller (Fig. 5.7). The interface enables the master computer to introduce the robot controller processor into a wait/hold state. When the robot controller processor is in the wait state, the master computer gains access to the internal memory of the robot controller. It can either read it or write into it. The program running on the master computer (a compiled TORBOL program) exerts influence on the motions of the robot by modifying the virtual program.

The virtual program is a template of a program that is being constantly executed by the robot controller. The sequence and the instruction codes of the virtual program remain unaltered. The arguments of the instructions are altered with each robot motion that is to be executed. During the modification of the arguments the processor, which executes the virtual program, is not functioning (it is in the wait state). The modification of each argument should be considered to be an indivisible operation and is undetectable by the robot controller processor executing the virtual program. The successive modifications of the arguments in successive robot motions are shown in Fig. 5.8. The virtual program consists of two sequences of instructions, each terminated by an instruction jumping to itself. While one of the sequences is being executed, the arguments of the other are being modified. When the motion is finished and all the arguments have been modified, the label in the appropriate GOTO instruction is switched, and so the other sequence begins its execution. In this way an infinite number of motions can be executed.

TORBOL was also implemented at Loughborough University of Technology, on the same system that executed ROPAS and ROOPL programs (Fig. 5.4).

6 Research-Oriented Robot Controller: RORC

Currently, investigations of robot controllers concentrate on several problems. One of the areas is the development of new methods of programming these controllers. Robot manufacturers are enhancing teach-in methods by Robot Programming Language (RPL) constructs to make their controllers even more universal (e.g. [194]). Quite a considerable effort is concentrated on developing new RPLs, both specially defined for robots [12, 173, 176], and Computer Programming Languages (CPLs) enhanced by libraries of robot specific procedures [11, 53, 23, 178, 179, 181].

Many hardware architectures have been proposed specifically for:

– solution of manipulator dynamics based usually on the more efficient Newton-Euler formulation of dynamics equations [67]. The less computationally efficient Lagrange

formulation has also received some attention, e.g. general purpose parallel processor implementation [142]. Dynamics equations in different forms imply different architectures,
- solution of manipulator kinematics. Here, specialised hardware for vector and matrix computations as well as evaluation of trigonometric and cyclometric functions is usually employed,
- Jacobian computations (e.g. [20]),
- vector and matrix computations being subtasks of the above-mentioned tasks (e.g. [120, 106]),
- general image processing (e.g. [3]),
- integration of sensor data (e.g. [155]).

All of the above-mentioned architectures are concerned with speeding up computations. Both the specialized hardware [144] and the general-purpose hardware architectures have been investigated [104]. Hardware developments are supplemented by research into more efficient computational algorithms [67, 142]. A very good overview of processor architectures tailored to robotics applications is given in [45]. As the development of silicon compilers has greatly simplified the design process of new VLSI chips, many investigators are tempted to produce custom-designed chips meeting their specific computational requirements. These requirements may arise from both the computational algorithms developed (e.g. solution of dynamics equations) and the task that the robot has to execute (e.g. robot motions utilising image processing).

IBM-PCs connected by an Ethernet network or several signal processor hardware configurations were utilised in experimental general purpose robot controllers [9]. QNX [191, 52, 119] real-time operating system was used with the IBM-PC network, but for the signal processor architecture a new operating system needed to be developed.

Experiments with task-specific controllers for executing a single but complex task are also conducted. Good examples of such complex systems are: a ping-pong playing robot [4] or a sheep-shearing robot [139]. In these cases especially the software structure of the controllers conforms to the task that the robot is to accomplish. Some researchers enhance limited computational capabilities of industrial robot controllers by adding to the system external computers executing procedures related to vision and planning. Such architectures were proposed for a robotic system solving jigsaw puzzles [18] and a system for decorating scale model cars [137].

A behavioural approach to robot control has also been utilised in several controllers. In such systems the control program is a collection of independent behavioural modules. Each of the behaviours contains some expertise concerning the task that is to be executed. The behaviours have priorities associated with them and are activated by specific situations detected by sensors probing the environment. One such system locates and retrieves empty soda cans in an unstructured environment [22], another constructs complex shapes out of *soma blocks* [92].

Some work has been done on sensor data fusion (data aggregation) [90, 31]. Different algorithms have been developed for aggregating data from hardware sensors. Two terms have been coined, independently by several authors, describing the process of data fusion and the resultant data aggregate: logical sensor (e.g. [155, 172]) or virtual sensor (e.g. [175]).

The problem of structuring a robot system in such a way so as to enable easy incorporation of multiple sensors is starting to attract the attention of the research community.

A method of integrating multiple logical sensors into a robotic system was proposed in [155]. It assumes that a hierarchy of robot control levels should be matched by a hierarchy of logical sensors. The concept of object-oriented programming is adopted. Logical sensors and robot control activities are treated as objects and inter-object communication is used as a method of transferring data. Incorporating new logical sensors is equivalent to adding new objects and ensuring interaction between the old and new ones. Other software structures of controllers simplifying sensor incorporation, regardless of the nature and complexity of the sensor, can be found in [90, 91, 175]. Such a controller is also at the focus of this dissertation.

The flexibility of conventional robot controllers is achieved by rendering the method of their programming (i.e. RPL) universal. In consequence, the specialized RPL must have all the properties of any Computer Programming Language (CPL) plus all robot specific commands – making it hard to master and difficult to implement. Moreover, we cannot be certain that the robot specific part will not have to be enhanced when a new sensor has to be incorporated into the system.

All the above drawbacks can be eliminated by changing the approach to robot programming. During investigations, and even more so during production, the robot is performing a single and a well-defined task. While executing this task, the program is not altered, so the whole universality of the language is not used (usually only a small subset of instructions of RPL is utilized). From this point of view it is rational to tailor the controller to the needs of the task. This implies that for each task a new controller has to be produced. This is economically feasible only when modifying the controller software structure. The hardware structure must remain unaltered. To make the process of creating a new controller easy, a library of ready-to-use program blocks (procedures and concurrent processes treated as construction blocks) must exist. The flexibility of the controller can be achieved in two ways: through the selection from the library and proper arrangement of blocks or through the creation of new blocks (e.g. by modifying the existing ones). As the controller software can be coded in a high-level CPL (C [69] in our case), this is a relatively simple process and, moreover, there are no limitations to what such a controller can do.

The robot program is coded in C as a sequence of calls to library procedures and, if necessary, user defined procedures and any C instructions. This program is compiled and appended to the fixed part of the controller. The fixed part contains the user interface and the concurrent processes module. The result of the compilation (the executable code) is loaded into the controller hardware and executed there, and so the robot task is executed. If the task is altered, the robot program part of the controller has to be exchanged and the compilation and loading steps have to be repeated.

The idea of using the C language extended by robot specific library routines to program robot motions originated with the implementation of RCCL [53]. Initially in RCCL a motion request was a request to the system to modify the world model, so that a position (homogeneous transform) equation became valid. The trajectory generator was an interrupt driven background process (in relation to the world modelling task) that used position specifications obtained from the world model to compute joint positions at sample rate. The programming of the system was done by invoking procedures constructing position equations. Once the equation was assembled, procedures initiating the motion of the arm in such a way so as to satisfy this equation were called. There were functions that had been specifically designed to introduce compliance, limit forces or interrupting

motions, as well as waiting for events to occur. Although motion planning and execution were performed concurrently, the programmer did not write separate processes for each of these tasks. A single program was written bearing in mind that after initiating a motion (by calling a move function) the subsequent instructions will be executed in parallel with this motion. Later synchronisation was obtained by placing a wait for motion termination or a wait for an event to occur procedure somewhere in the subsequent code.

In the RCCL system sensor incorporation was done by assigning a global variable to the sensor and updating it at sample rate. The value so obtained was used in a background function. The RCCL system mainly supported position encoders and force sensors utilised in the servo loops. The intermediate positions of the arm were obtained by interpolating in either the Cartesian space or joint space. The RCCL system relied on global variables for communication between its processes.

A similar approach was utilised in ARCL (Advanced Robot Control Library) [23]. This library also employs position equations for motion specification. The position equations are regarded as appropriate motion requests and are queued by the user program task for the trajectory generator for execution. The synchronisation between the two, when required, can be obtained through binary semaphores or by using special attributes while queuing the motion requests.

RCCL was extended to allow the control of multiple cooperating robots – as a result RCI (Robot Control Interface) [84] was designed. In RCI each robot and the transferred object has its own trajectory generator implemented as a task. The planning level task queues the motion requests to the trajectory tasks. Communication between tasks is through shared memory.

A development of RCCL, called KALI [6, 54, 55, 105], also queues motion requests and controls multiple cooperating robots. In the case of KALI, motions are treated as processes (they are created, executed and finally killed). Synchronisation between the motions is obtained through combined use of motion control flags and motion parameters (e.g. velocity, time of arrival). As in RCCL, in KALI little work has been done on complex sensor incorporation and sensor data aggregation for the purposes of higher level control. On the other hand, it should be noted that both in the case of RCCL and KALI there exists the possibility of changing the servo control algorithms. This was obtained by designing robot specific hardware.

The controller proposed in this dissertation employs full concurrency (as in KALI) and data pipelines for inter-process communication. The motion of the arm can be specified in Cartesian, joint and motor increment space, and no influence on the servo control loops exists. No motion queuing mechanism was employed. In this way an open structure of the system was obtained with regard to high level control. Moreover, the proposed structure accommodates complex sensors and deals with sensor data aggregation for the purposes of high level control.

6.1 Structure of the controller

The hardware of the system consists of a 32-bit Intel 80486 microprocessor-based computer and five 8-bit microprocessors. Each of the 8-bit microprocessor based servo--drives MA-70 controls an electric DC motor actuating one of the five degree-of-freedom manipulator axes. The 32-bit processor is interfaced directly to the servo-drives. It transmits position increments to each of them and receives feedback signals from them. The 32-bit processor bus is connected by a parallel 800 kbyte/s interface MIAT [76] to the five servo-motor controllers and I/O boards MC-42. The hardware structure of the system is shown in Fig. 6.1. The original microprocessor and its memory shown in Fig. 6.1 (MM-16 and ML-16), situated in the control cabinet of the IRp-6 robot, remain dormant throughout the functioning of the 32-bit microprocessor.

The hardware of the system is rendered invisible to the programmer by the concurrent processes module. This module enables the creation and destruction of processes, their synchronization through semaphores, data transfers through pipelines, and process scheduling. Currently, all processes of the system are executed by the 32-bit processor. The layered structure of the system is shown in Fig. 6.2.

The system (Fig. 6.3) is composed of the:

- Command Process (CP),
- Response Process (RP),
- Robot Control Process (RCP),
- zero or more Virtual Sensor Processes (VSPs).

The hierarchic dependence of the processes (creation/destruction tree) is shown in Fig. 6.4. The programmer can modify the RCP and VSPs according to certain rules, but apart from that he is free to tailor the system to his investigative needs. Once the modifications are done, the source code of these processes is compiled and linked to the code of the other processes.

6.1.1 Operator interface

The operator interface is serviced by the Command and Response Processes. The operator has a command menu at his disposal and operates the system through it. He can initialize the robot (command it to the home position); execute the Robot Control Process; abort, suspend or resume the execution of the RCP; or quit the system. All the processes communicate with the operator through the Response Process. They transfer the messages through pipelines to the RP. The RP reads the messages, formats them and displays them in appropriate windows on the screen of the monitor, or saves them in a file. Each process has its own windows on the screen.

6.1.2 Virtual Sensor Processes

As there is a multitude of real sensors, and the data obtained from them ranges from simple one-bit signals to complex bit patterns obtained from CCD cameras, a method of interfacing these sensors to the system, independent of their complexity, was devised. It was assumed that the generic structure of a virtual sensor process should enable real sensor data aggregation according to functions $f''_{v_j} \quad j = 2, 4, 5, 7$ mentioned in (4.7). It suffices to take into consideration only the most general function, out of the ones mentioned in (4.7), i.e. f''_{v_7}. Fig. 6.5 presents the generic form of the Virtual Sensor Process

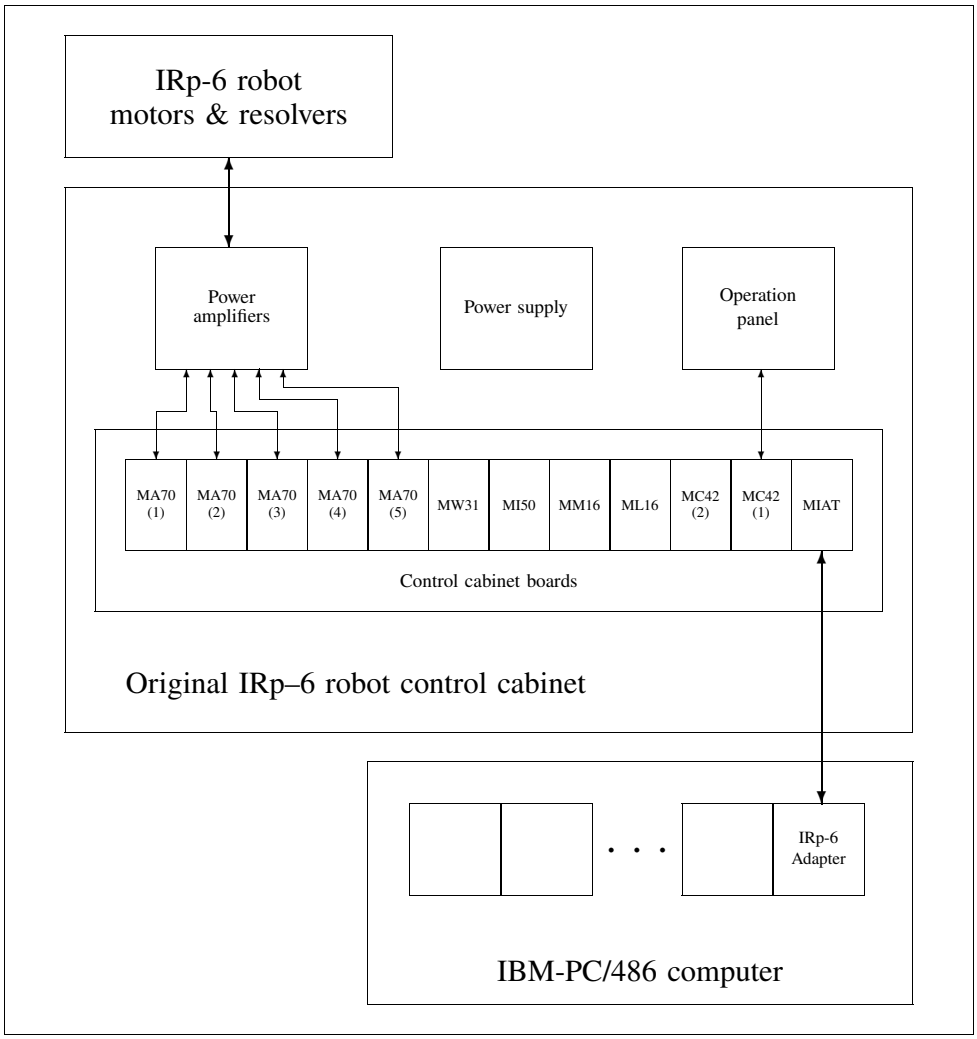

Figure 6.1. Hardware structure of the system

(VSP) flow diagram. Its main task is to obtain the readings of the real sensors (one or more) and to aggregate this data, so that the virtual sensor reading is obtained. Obviously, the aggregate can include both variable values and effector state. If the effector state is needed in the computation of the virtual sensor reading, it is transmitted through a pipeline from the Robot Control Process.

Several VSPs can access one real sensor. To obtain a virtual sensor reading, for example, readings of several strain gauges can be processed to obtain two vectors – force and torque. The VSP sends its reading through a pipeline to the Robot Control Process (RCP), either in an interactive way (data is computed when VSP is ordered to) or in a non-interactive way (data is ready whenever needed). The programmer has no limi-

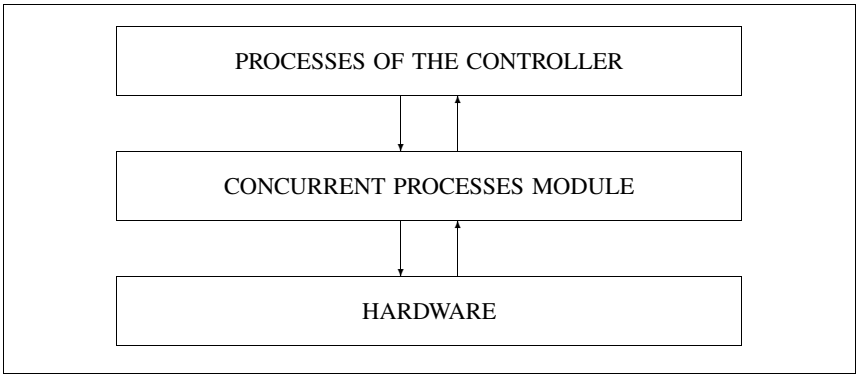

Figure 6.2. Layered structure of the system

tations as to which real sensor readings will form the virtual sensor reading and how it will be formed. In this way the aggregating function f''_{v_7} can assume any form, as it was initially postulated.

6.1.3 Robot Control Process

The Robot Control Process (RCP) is the only process directly influencing the manipulator. Its main task is to compute the set values for the five servo-controllers. It was assumed that the generic structure of the Robot Control Process should enable trajectory computations according to functions f_{e_j} $j = 0, \ldots, 7$ defined in (4.4). It suffices to take into consideration only the most general function, out of the ones mentioned in (4.4), i.e. f_{e_7}. Each motion instruction enables the computation of the next effector state e^{i+1} by evaluating the function f_{e_7} or any of its simpler cases. Currently, the only influence on the dynamics of the system can be exerted by shaping the set values to the axis servo--controllers. The set values can be computed from a trajectory expressed in Cartesian space, joint space or in motor-increment space. The next effector state e^{i+1} is then expressed in either Cartesian space, joint space or motor-increment space. In the case of Cartesian-Euler space, axially-symmetric tools were assumed, as the robot has only five d.o.f.[1] The form of function f_{e_7} implies that the trajectories can also be computed from or modified according to virtual sensor readings. Figure 6.6 presents the generic form of the **MOVE procedure** (motion instruction). The form of this instruction remains the same regardless of the space in which we express the trajectory, i.e. regardless of the space that e^i is expressed. Computation of the next motion step e^{i+1} inside the MOVE procedure is done utilizing f_{e_7} coded in C – as initially postulated.

Synchronisation, between a motion instruction and the virtual sensor processes it uses, can be described by a Petri net [111, 116], and is presented in Fig. 6.7. The figure shows a part of the RCP associated with a single motion instruction (left column of places) and a single VSP (right column of places). The places in the middle belong to the RCP process. If several VSPs are needed, then new VSP branches of the Petri net

[1] Degree of freedom.

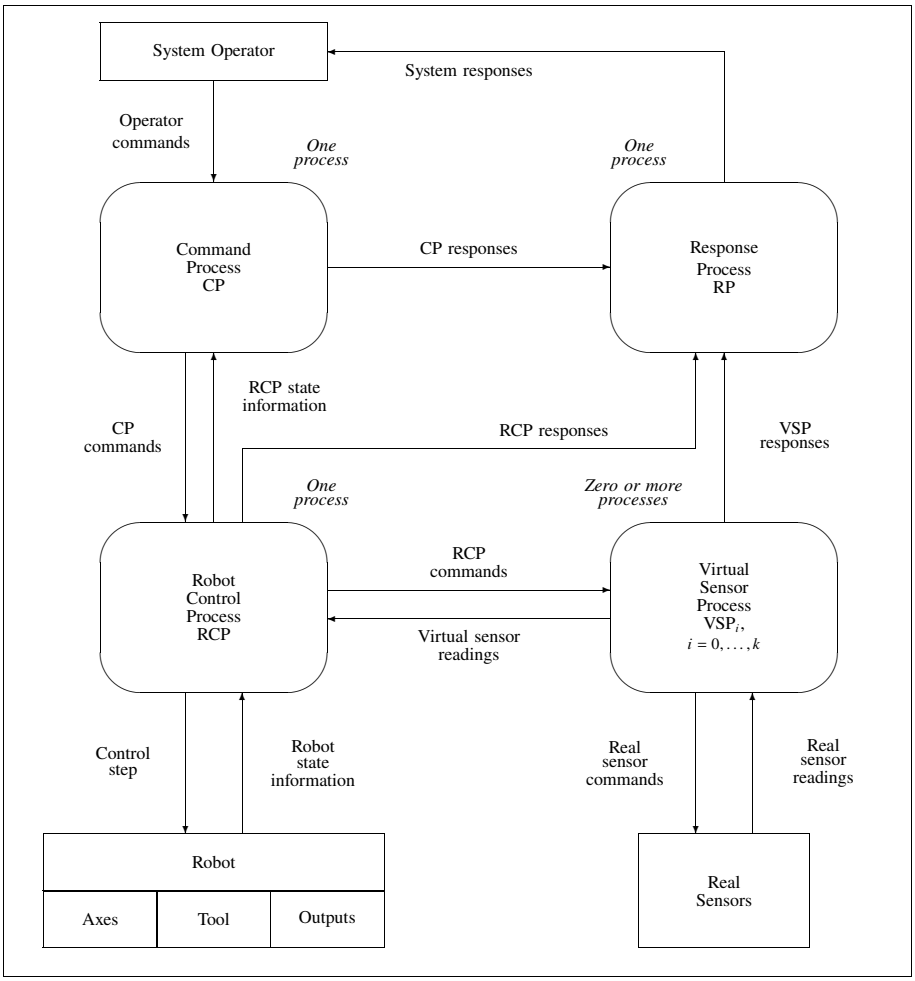

Figure 6.3. Logical structure of the system — process interactions

have to be added (in parallel to the right column of places). The concurrent program implementation of the Petri net requires semaphores and pipelines for synchronization and data transfers [178, 179, 181]. If several levels of data aggregation have to be performed in parallel, a multi-layered Petri net can be used [171, 172].

There is also a generic form of a **WAIT instruction**, waiting for an event to happen (Fig. 6.8). The event is signalled by virtual sensors. The condition is evaluated by computing the value of one of the functions from (4.2), i.e. the functions that take r^i as an argument: $f_{d_j}, \quad j = 3, 5, 7, 8$.

The RCP is composed of the MOVE and WAIT instructions and any other C language instructions, if necessary (Fig. 6.9). Each MOVE and WAIT instruction creates and kills its VSPs according to need.

Figure 6.9 presents the structure of the RCP. This structure (shell) has to be maintained whenever a new RCP has to be constructed to execute another robot task. Otherwise,

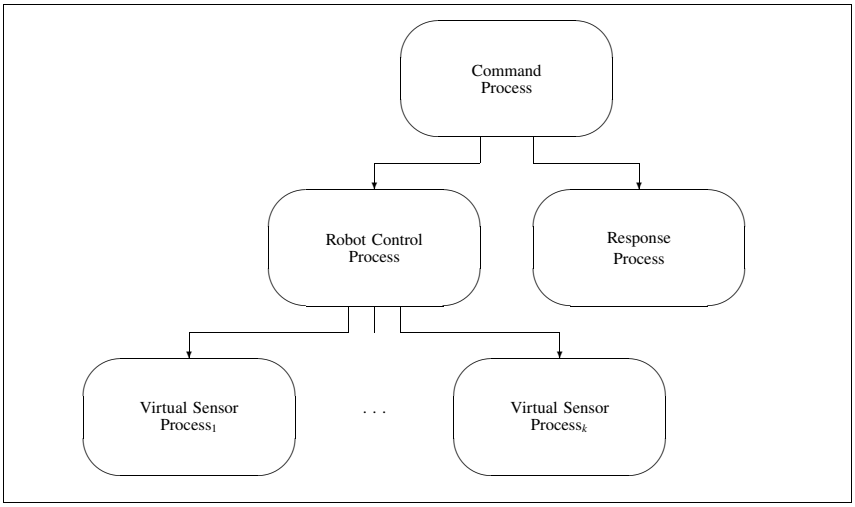

Figure 6.4. Hierarchical dependence of system processes

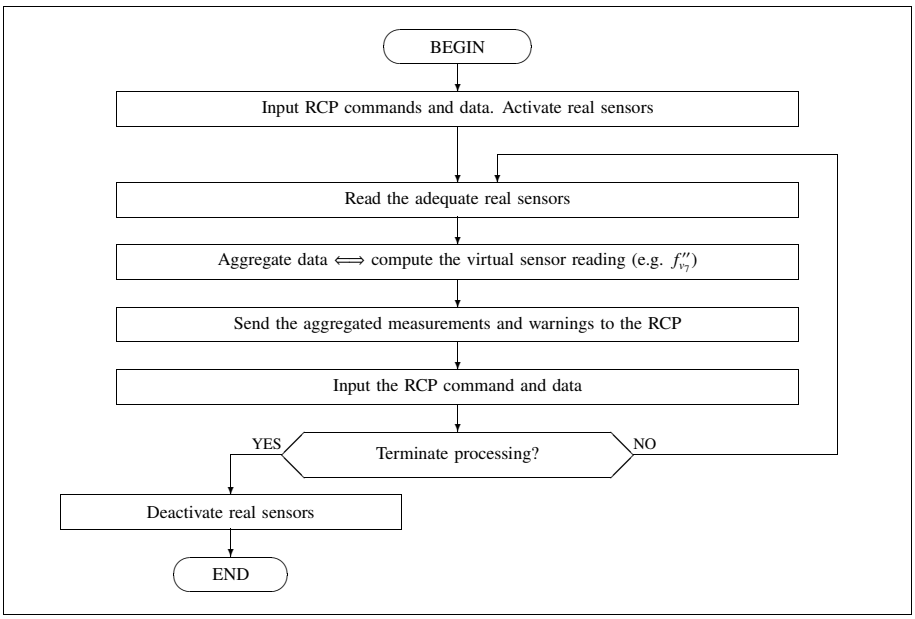

Figure 6.5. Generic structure of the Virtual Sensor Process VSP

the RCP will not interact properly with the other components of the system. One of the things that can vary at this level is the number of blocks denoted by $***$ in Fig. 6.9.

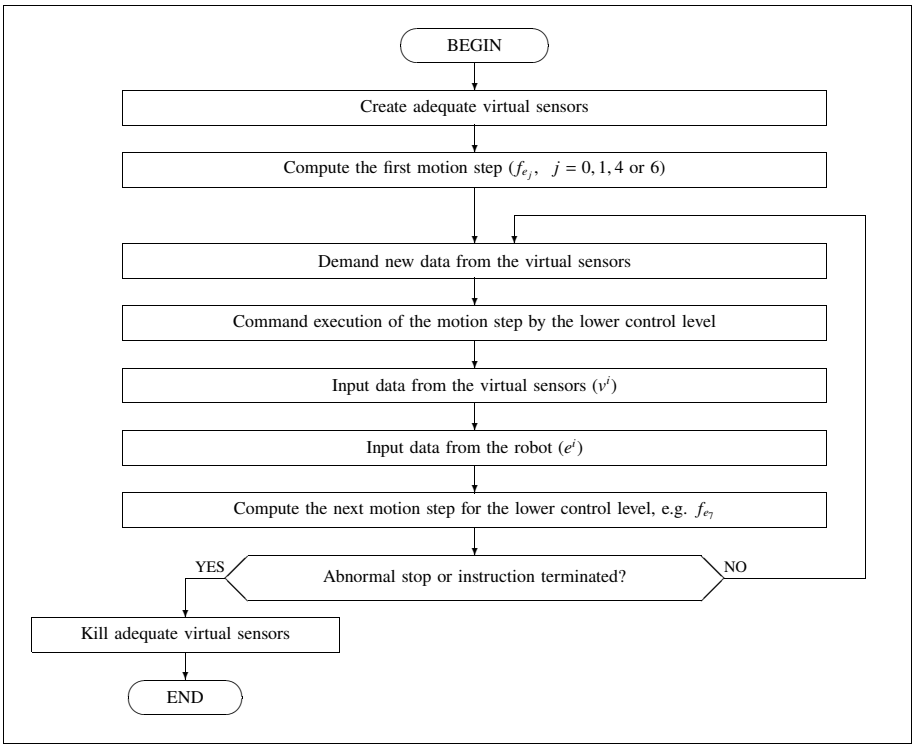

Figure 6.6. Generic structure of the MOVE procedure

However, full flexibility is obtained by varying the contents of these blocks. This is done by:

– using any C language instructions and functions not invoking directly or indirectly the hardware of the original robot control cabinet,
– calling MOVE and WAIT procedures executing motions in any of the three spaces: Cartesian-Euler, joint or motor-increment,
– creating any procedures invoking robot hardware, and later calling them inside such a block (in such a case these procedures have to conform to the calling conventions defined by the system specification).

6.2 Creation of a new controller

A new controller has to be assembled whenever the executed task changes. If the sensors remain the same, and the method of aggregating data remains unchanged, the VSPs need not be altered and only a new RCP is constructed. Usually, the set of existing library functions suffices, but new functions can be added if necessary – obviously, they have to conform to the RORC standard [178, 179]. The generic structure of the RCP is shown in Fig. 6.9. The kernel of the RCP is the user program – this varies with each task. It is surrounded by the shell that remains unaltered and is responsible for proper

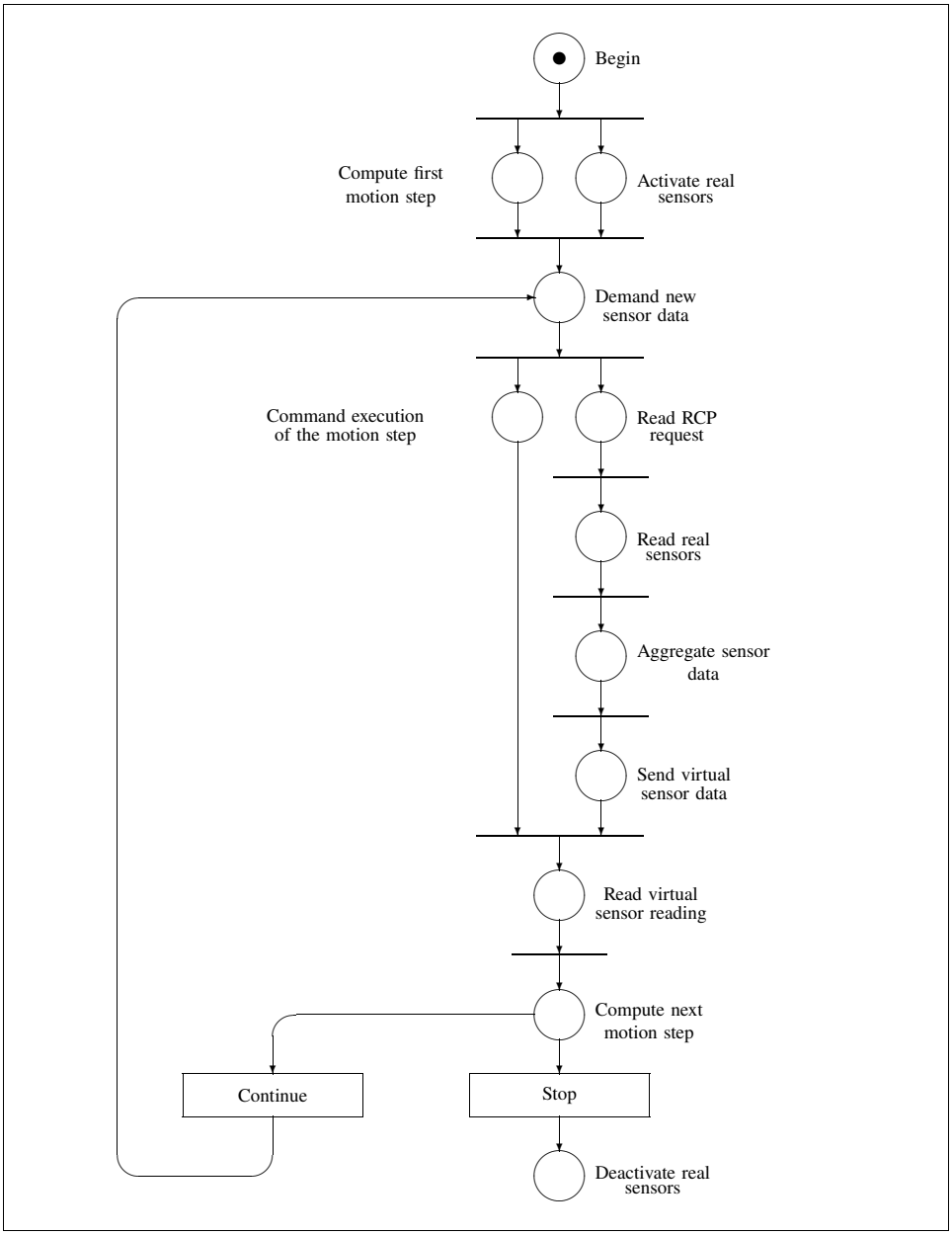

Figure 6.7. Synchronization of a MOVE procedure with a Virtual Sensor Process (Petri net with initial marking)

synchronisation with other processes. For both parts of the RCP to function in harmony the new procedures must conform to the RORC standard too [178, 179].

Each of the procedures interacting with the system hardware (arm, tool, cooperating devices or sensors) ends its execution in one of the five states: fatal error (system

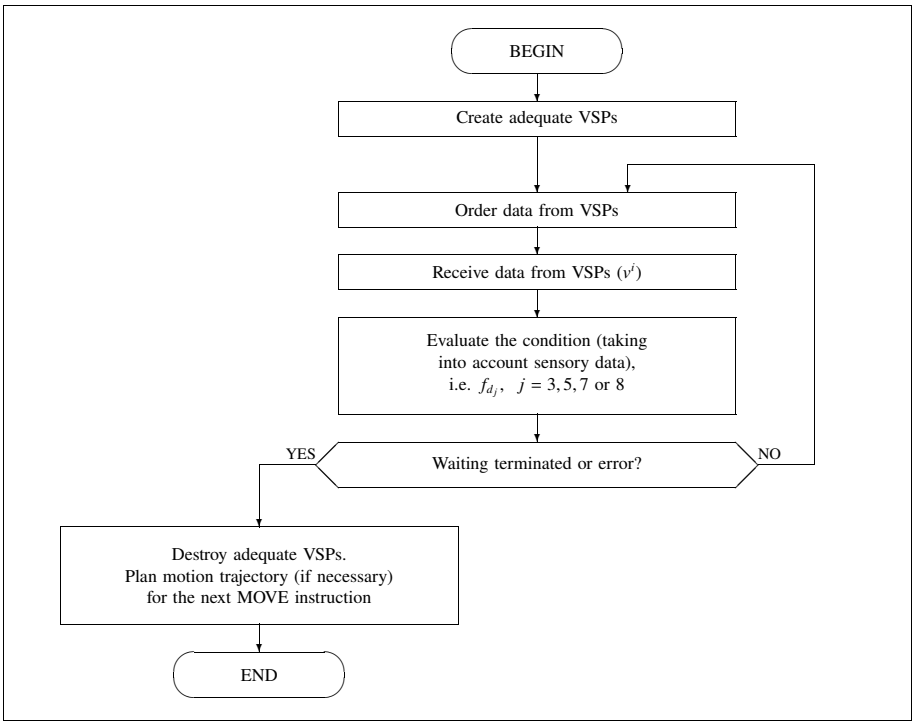

Figure 6.8. Generic structure of the WAIT procedure

hardware failed), non-fatal error (computational error), wait (operator issued a PAUSE command), stop (operator issued a STOP command) or uninterrupted termination. Besides this, the list of parameters of each of these procedures contains a pointer to a variable containing the specific error number or 0 (OK) if no error was detected during execution. For the proper functioning of the system those procedures cannot be invoked directly. They are called indirectly through a system interface procedure, which takes care of the proper synchronisation with other parts of the system (other processes). In this way the programmer creates only the kernel of the RCP and the synchronisation with the other parts of the system (i.e. shell) is handled by the interface procedure of the RCP. In consequence the format of motion instructions invocation is the following:

```
if ( non_terminal_instruction ( motion_proc(..., &r), &p, &r )
                                == BEGIN_PROGRAM)
    continue;
```

where the three dots symbolise any number and type of motion instruction (procedure) parameters, BEGIN_PROGRAM is a constant defined in the RCP process (if the non_terminal_instruction returns this value, the user program is repeated from its beginning), r holds the error number or OK, and p defines the user program status. If motion_proc is the last motion instruction of the user program, then it should be invoked through terminal_instruction rather than through non_terminal_instruction.

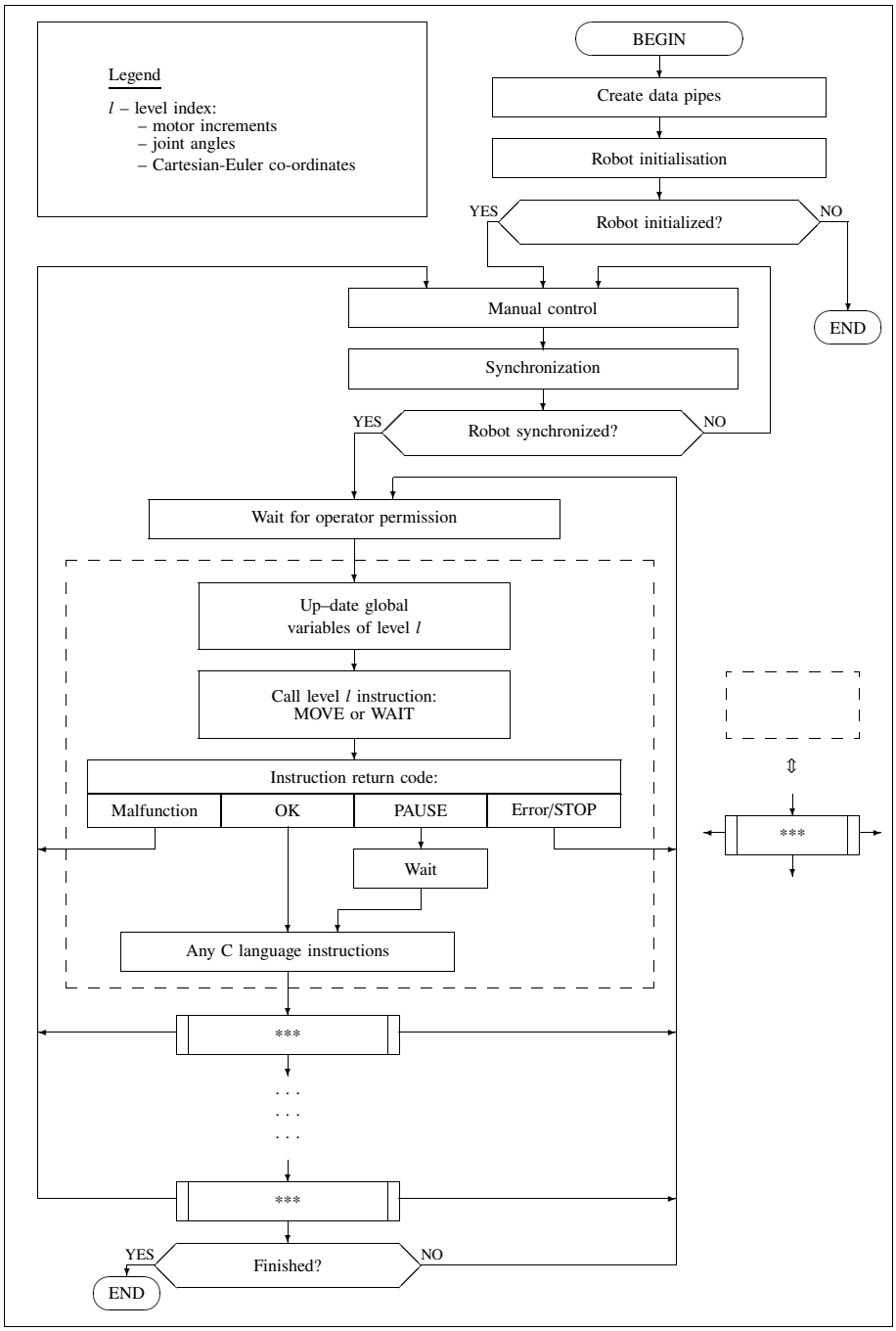

Figure 6.9. Method of creating the Robot Control Process

Obviously, the procedures controlling the hardware have to monitor both the software and hardware errors and the operator commands. As they are executed iteratively, in each step the hardware is checked, computations are verified (e.g. Is the inverse kinematics solution valid?) and the pipeline through which the operator commands are issued is tested. The operator has five commands at his disposal: start robot synchronisation – START (this is necessary only if a robot has incremental joint position measurement), execute or resume the task – PROGRAM, suspend the task execution – PAUSE, abort task execution – STOP, and halt the system – QUIT. Not all the commands can be issued at any instant. The system checks that and warns the operator, if the system state context of the command is wrong.

Table 6.1. Comparison of RORC and the two types of IRp–6 industrial controllers

FEATURE	INDUSTRIAL CONTROLLER WITHOUT SENSORS	INDUSTRIAL CONTROLLER WITH SENSORS	RESEARCH ORIENTED CONTROLLER
PROGRAMMING			
Programming method	on-line – teaching	on-line – teaching	off-line – C language
Program debugging	partial (single step execution, single instruction display)	partial (single step execution, single instruction display)	full (C language debugger, printout)
Program documentation	none (single instruction display)	none (single instruction display)	full (e.g. listing)
Programmer's qualifications	very low	low	high
controller extensibility	closed structure (manufacturer modifiable)	closed structure (manufacturer modifiable)	open structure (user modifiable)
Hardware configuration	non-modifiable (one processor)	non-modifiable (one processor)	modifiable (several processor)
MOTION SPECIFICATION			
Motion goal specification	teaching goal poses	taught goal pose can be modified by sensors	goal position modified or generated by sensors or calculated
Method of trajectory specification	instruction designates one of the few interpolation types	sensor readings modify the interpolated trajectory by a constant vector	any computed trajectory can be used
Influence over dynamics	none	none	servo set-value modification
Velocity control	percentage of maximum velocity and duration of motion	percentage of maximum velocity and duration of motion	any function
Servo sampling rate	rigid (32ms)	rigid (32ms)	quantified (8, 16, 32, 64 [ms])
MATHEMATICAL OPERATIONS			
Variables	none	none	all types
Expressions	none	none	all types
SENSOR INTERACTION			
Event detection	very limited (reads binary signals)	limited (several teach-pendant instructions)	unlimited
Data aggregation	none	none	unlimited
Incorporation of new sensors	not possible	only the few supplied by the manufacturer	unlimited

If new sensors are incorporated into the system or data aggregation methods for old sensors change, a new VSP has to be created. Apart from the shell that executes data communication with the RCP through data pipelines, the kernel of the VSP can assume any form – suitable to the hardware and the aggregation method.

Once the new RCP and VSPs are created the whole of the software has to be compiled and linked. To make the task easy for the user, a system of make-files was created [178, 179], so the job is straightforward. In this way an executable file is created and it is invoked under DOS[2].

[2] Disk Operating System running on IBM-PC class computers.

6.3 Experimental results

The controller software was tested on several tasks, requiring three to five processes:

- search for a path to a goal in a maze using only local information gathered by a 5–bit touch probe,
- search for a path to a goal in a maze using only local information gathered by a force/torque sensor,
- traversal of a shortest path to the goal in a maze using global information obtained by a CCD camera,
- exerting a constant force perpendicular to the traversed trajectory.

It was run on a 33 MHz IBM-PC/486 compatible computer. The processes were switched by the scheduler driven by a timer interrupt with a frequency of 18.4 Hz. Although each of the processes had 54 ms time slice for itself; the CP, RP and VSP used a negligible fraction of that for their needs, so whenever they finished their share of processing they informed the scheduler of this and the scheduler transferred control to the next process in the queue without further delay. The robot axis controllers demand new set values every: 8, 16, 32 or 64 ms (selected). It should be noted that if the VSP has to do more processing, which is the case if a more complex sensor is used (e.g. vision sensor), then the VSP uses up all of its 54 ms time slice. In the most complex cases even more time would be needed, so several time slices would be needed to compute a single trajectory modification. In some cases it would not be a problem, e.g. when the motion is slow or the reaction does not have to be very quick. In the case of fast and precise motions more time would have to be assigned to the VSP. That would result in the starvation of time for other processes, especially the RCP, which is also time-demanding. The only solution to this problem is to distribute the processes over several processors.

The Research-Oriented Robot Controller proved instrumental in the design of new robot control algorithms. By using traditional industrial robot controllers most of this work could not be done. The comparison between RORC and two IRp–6 industrial controllers is shown in Table 6.2. RORC excels in all categories but one: the programmer's qualifications must be much higher.

The next section shows how RORC can be used for investigating new control algorithms. The search for a path to a goal in a maze using only local information gathered by a force/torque sensor is an exemplary task. Sensor-based reactive robot control, developed by the author, is used as a control strategy in this example.

7 Sensor-based reactive robot control

The artificial intelligence approach to robot control strongly relies on world models to execute a task. Sensors, in this case, are mainly used to up-date the world model, which in turn is used in the generation of the plan of actions. On the other hand, the behavioural control concept does not need a world model to execute a task [143, 15, 16, 17, 94, 146, 22]. In this case the controller is built of several finite state automatons functioning in parallel, each achieving a single objective by a certain behaviour. The

controller is constructed incrementally by adding ever more complex layers of behaviours on top of the more elementary ones. Upper layers examine data from lower levels and can suppress or inhibit their behaviours. The lower layer continues to run unaware of the layer above it, which sometimes interferes with its data flow. Each of the layers relies on information obtained by sensors. Every behaviour is built to be active when the world is in an appropriate state. In such a system, instead of representing the environment by a world model explicitly, only a mapping from an aspect of the environment obtained by sensors is used.

It should be noted that the integration of a world model into behaviour-based mobile robots has also been the subject of research [94]. However, in this case the world model was distributed throughout a collection of concurrently active landmark behaviours, which matched landmarks detected by sensors to the particular behaviours they were encoding.

The approach, followed in this dissertation, to the utilization of sensor information in robot control was inspired by behavioural control, although it differs in many aspects. In the case of behavioural approach, as described above, the task is split into several task achieving behaviours which form a layered system with higher level layers subsumming the roles of lower level layers when they wish to take control.

A goal that is to be achieved is distinguished from a single layer of actions (that can also be called behaviours) that are executed when sensors detect appropriate conditions. Unlike in the pure behavioural approach, where the partitioning of the system is intuitive, a formal path was followed. First, the sensor reading space was partitioned into sub--spaces. With each of these sub-spaces an action (or rather a reaction) was associated. If during the realisation of the goal, the sensor readings "enter" a sub-space associated with a certain action, then the realisation of the global goal is interrupted and the action is executed. Usually, these are some kinds of defensive reactions to sudden changes in the environment. As the sensor reading space, robot reactions and the global goal can be described formally, a formal specification of the robot controller was produced. It was later used as the basis for coding the software of this controller. As a platform for the implementation of goal-achieving reaction-based controllers, the flexible controller described in [175, 180] was used.

The idea of subdivision of sensor reading space into sub-spaces can be found in [123] (although there it is called the information space). This space is divided into two sub-spaces: success sub-space and error sub-space. While the assembly task proceeds, the sensor readings are monitored. As long as those readings remain within the success sub-space, the task is continued. Once they enter the error sub-space, either the task is aborted or an error recovery routine is undertaken (e.g. unaltered action is repeated once more). In this case the task is treated as a single preprogrammed entity and sensors are passive, i.e. they only monitor (not control) the task execution. The task proceeds until completion or until an error is detected. First an error must occur and then a modifying action (error handling) can be undertaken. It would be much better if sensor readings were used to control motions in such a way so as to avoid errors rather than to detect them. The paper [123] does not mention how to specify the task or the error handling routines. Moreover, it does not deal with the problem of association of information sub-spaces with recovery routines.

The majority of robot systems is computer-controlled, so the execution of a task can be subdivided into steps. Let the initial state be labelled 0 and the consecutive interme-

diate states $i = 1, \ldots, i_T$, where i_T is the label of the terminal state. In the terminal state either the task is accomplished or an execution error is detected.

While the robot is executing the task, the sensors monitor the state of the system and the environment constantly. The task is executed according to some **general plan** – associated with the **main reaction**, but it can be hindered by certain external events. These events can be detected by sensors. Once this happens, the system has to react to them by a **reaction** appropriate to the event. The system can exhibit a number of pre-planned reactions $B_j, j = 0, \ldots, j_R$, where $j_R + 1$ is the number of reactions. Each of the reactions can be executed by realising an instance of this reaction. A **reaction instance** b_j of the reaction B_j is executed as a sequence of steps (characterised by effector and control subsystem state):

$$b_j = (e_j^0, c_j^0)\ (e_j^1, c_j^1)\ \ldots\ (e_j^T, c_j^T), \quad b_j \in B_j, \tag{7.1}$$

where (e_j^0, c_j^0) is the initial state and (e_j^T, c_j^T) is the terminal state of the reaction instance $b_j \in B_j$.

A reaction B_j is triggered by a virtual sensor reading belonging to a virtual sensor reading sub-space $V_j \subset V$. There exists a virtual reading sub-space $V_0 \subset V$, called the **neutral reading space**, which does not trigger any specific reaction, i.e. while the virtual sensors are supplying readings from this sub-space, the previously executed reaction is continued. In other words, if virtual sensors find out that the initial condition for the execution of the reaction B_j is satisfied, then this reaction is executed. Obviously, the previously executed reaction is aborted at this moment. The currently executed reaction terminates either because virtual sensors trigger another reaction or because the job of this reaction is done (the final state of this reaction is reached). It is assumed that there exists a reaction B_0, called the **main reaction**, which does not have to be triggered by any virtual sensor reading. The system initially executes the reaction B_0 and whenever the execution of other reactions terminates, and no other reaction is triggered by adequate virtual sensor readings, the execution of the reaction B_0 is resumed.

If the virtual sensor reading space is subdivided into sub-spaces $V_j, \quad j = 0, \ldots, j_R$, where $j_R + 1$ is the number of these sub-spaces, in such a way that:

$$V = V_0 \cup \bigcup_{j=1}^{j_R} V_j \quad \text{and} \quad \forall_{j \neq q} V_j \bigcap V_q = \emptyset, \quad q = 0, \ldots, j_R, \tag{7.2}$$

then each virtual sensor reading $v_j \in V_j, \quad j = 1, \ldots, j_R$, triggers a reaction instance $b_j \in B_j$. If the reaction B_j is triggered based only on information contained in the virtual sensor reading v_j ($v_j \Rightarrow B_j$), then B_j is called the **state independent reaction**, but if the reaction B_j is triggered based on both the information contained in virtual sensor reading v_j and the current effector state e_i or control state c_i ($(v_j, e_i, c_i) \Rightarrow B_j$), then B_j is called the **state dependent reaction**.

7.1 Simple maze running task

The task that the robot had to accomplish initially had been extremely simplified to make the example easy to follow. The task consisted in finding a path in a maze from

its entrance situated in the South-West (S-W) corner to the exit in the North-East (N-E) corner. The robot transferred a probe mechanically coupled with a force sensor. Each contact with a maze wall changed the force acting on the probe. The simplification consisted in the assumption of a maze with no cul-de-sacs. This assumption simplifies the path-finding algorithm (no backtracking in the N-S direction is necessary).

It was assumed that the N direction coincides with the $Y+$ axis of the maze co-ordinate frame, and the E direction with $X+$. The robot that was used had only 5 d.o.f., but as the probe is axially-symmetric, its location is specified by 5 co-ordinates (3 Cartesian and 2 Euler angles):

$$e = [e_x, e_y, e_z, e_\phi, e_\psi], \quad e \in E \tag{7.3}$$

The goal G_* was formulated as:

$$G_* \;:\; e^{i_T} \in E_*^{i_T} \qquad E_*^{i_T} = E_{*x}^{i_T} \times E_{*y}^{i_T} \times E_{*z}^{i_T} \times E_{*\phi}^{i_T} \times E_{*\psi}^{i_T} \tag{7.4}$$

where $E_*^{i_T}$ describes the exit from the maze and i_T is the number of executed steps (this number is *a priori* unknown).

The virtual sensor reading is expressed as:

$$v = [\, v_x,\ v_y,\ v_z \,] \tag{7.5}$$

where v_x, v_y, v_z are the components of the force acting on the probe, expressed in the maze frame. The aggregating function $f''_{v_2}(r^i)$ computes these from the real sensor readings.

The division of virtual sensor reading space V and the assignment of reactions B is:

$$\begin{cases} V_0 : & |v_x| \le threshold_x, & |v_y| \le threshold_y, & |v_z| \le threshold_z \\ & & & \Rightarrow B_j, \quad j = 0, \ldots, 4 \\ V_1 : & |v_x| \le threshold_x, & v_y < -threshold_y, & |v_z| \le threshold_z \\ & & & \Rightarrow B_1 = B_N \\ V_2 : & v_x > threshold_x, & |v_y| \le threshold_y, & |v_z| \le threshold_z \\ & & & \Rightarrow B_2 = B_W \\ V_3 : & v_x < -threshold_x, & |v_y| \le threshold_y, & |v_z| \le threshold_z \\ & & & \Rightarrow B_3 = B_E \\ V_4 : & \text{all other cases} & & \Rightarrow B_4 = B_{ERR} \end{cases} \tag{7.6}$$

If sensor readings enter neutral sub-space V_0 while the reaction B_j, $j = 1, \ldots, 4$, is being executed, then the reaction B_j is continued, if it has not been terminated at that instant due to all of its steps being completed. If termination of the reaction B_j occurs upon sensor readings entering neutral sub-space V_0 or if B_0 is being executed, then the reaction B_0 is continued.

The definition of a reaction is formulated by describing 'snapshots' of the state of the effectors and the control subsystem. This notation does not show how to obtain the next state, it only shows how the next state should look. The method of attaining that state is decided by the implementer of the system.

The main reaction B_0 is defined as follows:

$$b_0 \in B_0, \quad b_0 = (e_0^0, c_{vv_0}^0)\ (e_0^1, c_{vv_0}^1), \ \ldots$$

$$\begin{cases} e_0^0 & : \ e_x^0 = entry_x, \quad e_y^0 = entry_y, \quad e_z^0 = entry_z, \quad e_\phi^0 = entry_\phi, \\ & \ \ e_\psi^0 = entry_\psi \\ c_{vv_0}^0 & : \ \delta^0 = 1, \ \ i = 0 \\ e_0^i & : \ e_y^i = e_y^{i-1} + \Delta e_y, \quad \text{for } \ i = 1, \ldots, i_T \\ c_{vv_0}^i & : \ \delta^i = \delta^{i-1}, \quad \text{for } \ i = 1, \ldots, i_T \\ e_0^{i_T} & : \ e_x^{i_T} = exit_x, \quad e_y^{i_T} = exit_y, \quad e_z^{i_T} = exit_z, \quad e_\phi^{i_T} = exit_\phi, \\ & \ \ e_\psi^{i_T} = exit_\psi; \quad \text{where } \ exit_x \in E_{*x}^{i_T}, \quad exit_y \in E_{*y}^{i_T} \\ c_{vv_0}^{i_T} & : \ \delta^i = \delta^{i_T}, \ \ i = i_T \end{cases} \tag{7.7}$$

where:

i – the global task step number (associated with a counter),

i_T – the number of steps needed to accomplish the task – it is initially unknown, and

entry, *exit* – the entry and exit positions to and from the maze respectively ($exit_z = entry_z$, $exit_\phi = entry_\phi$, $exit_\psi = entry_\psi$).

The reaction B_0 starts its execution with the probe positioned at the entrance (*entry*) to the maze and the search-direction marker δ^0 (in step 0) set to 1 (when an obstacle is encountered, reaction B_N should start its search for a free passage to the right). If, while avoiding an obstacle, no thoroughfare is found in one direction, the marker will change sign, and the search will be continued in the opposite direction. The global step counter is 0. In each following step i the value of the marker (δ^i) does not change, the step counter is incremented and the probe is transferred by Δe_y in the north direction (equivalent to the maze Y direction). The reaction B_0 is executed until the probe reaches the exit area.

The reaction B_N is defined as follows:

$$b_N \in B_N, \quad b_N = (e_N^0, c_{vv_N}^0)\ (e_N^1, c_{vv_N}^1), \ \ (e_N^2, c_{vv_N}^2)$$

$$\begin{cases} e_N^0 & : \ \text{current effector state} \\ c_{vv_N}^0 & : \ i_N = 0, \ \ \delta_N = \delta^i \\ e_N^1 & : \ e_y^1 = e_y^0 - \Delta e_y' \\ c_{vv_N}^1 & : \ i_N = 1 \\ e_N^2 & : \ e_x^2 = e_x^1 + \delta_N * \Delta e_x \\ c_{vv_N}^2 & : \ i_N = 2, \ \ \delta^i = \delta_N \end{cases} \tag{7.8}$$

where i_N is the reaction B_N execution step number.

The 0–th state of the effectors in reaction B_N is equivalent to the last state of the effectors in the previously executed reaction. In the 0–th step the reaction B_N execution

step number i_N is 0, the global step number does not change and the search-direction marker δ_N for the reaction B_N assumes the same value as the value of the current search-direction marker δ^i. As the reaction B_N is activated by detecting $v_y < -threshold_y$, in the first step of the reaction B_N the probe slightly (by Δe_y) backs off to the south. In the next step the probe either moves east or west depending on the value of search-direction marker δ_N. With this motion the probe tries to avoid an obstacle (a wall).

The reaction B_E is defined as follows:

$$b_E \in B_E, \quad b_E = (e_E^0, c_{vv_E}^0)\ (e_E^1, c_{vv_E}^1))$$

$$\begin{cases} e_E^0 : & \text{current effector state} \\ c_{vv_E}^0 : & i_E = 0,\ \ \delta_E = \delta^i \\ e_E^1 : & e_x^1 = e_x^0 - \Delta e_x \\ c_{vv_E}^1 : & i_E = 1,\ \ \delta^i = -1 * \delta_E \end{cases} \tag{7.9}$$

where:

i_E – the reaction B_E execution step number,
δ_E – the reaction's local search-direction marker, and
Δe_x – the position increment in the west (i.e. negative X) direction.

The reaction B_E is activated if an obstacle is detected during the motion to the east. In this case, the probe moves slightly to the west and changes the sign of the search-direction marker.

The reaction B_W is defined as follows:

$$b_W \in B_W, \quad b_W = (e_W^0, c_{vv_W}^0)\ (e_W^1, c_{vv_W}^1))$$

$$\begin{cases} e_W^0 : & \text{current state of the effectors} \\ c_{vv_W}^0 : & i_W = 0,\ \ \delta_W = \delta^i \\ e_W^1 : & e_x^1 = e_x^0 + \Delta e_x \\ c_{vv_W}^1 : & i_W = 1,\ \ \delta^i = -1 * \delta_W \end{cases} \tag{7.10}$$

where:

i_W – the reaction B_W execution step number,
δ_W – the reaction's local search-direction marker, and
Δe_x – the position increment in the east (i.e. positive X) direction.

The reaction B_W is activated if an obstacle is detected during the motion to the west. In this case the probe returns east and changes the sign of the search-direction marker.

The reaction B_{ERR} is defined as follows:

$$b_{ERR} \in B_{ERR}, \quad b_{ERR} = (e_{ERR}^0, c_{vvERR}^0)\ (e_{ERR}^1, c_{vvERR}^1)$$

$$\begin{cases} e^0_{ERR} & : \quad \text{current state of the effectors} \\ c^0_{vv_{ERR}} & : \quad i_{ERR} = 0 \\ e^1_{ERR} & : \quad e^1_z = e^0_z + \Delta e_z \\ c^1_{vv_{ERR}} & : \quad i_{ERR} = 1, \;\; i = i_T \end{cases} \tag{7.11}$$

where:

i_{ERR} – the reaction B_{ERR} execution step number,

Δe_z – the position increment in the Z direction (i.e. rod lift increment).

The reaction B_{ERR} is executed when an unexpected event is detected. In this case the probe is lifted over the surface of the maze and the task is terminated (i.e. $i = i_T$). Under more complex conditions than those imposed by this task, freezing all the motions in the case of an error might be more appropriate.

Figure 7.2 presents the trajectory of the probe in an exemplary maze (Fig. 7.1).

The main advantage of the above specification of robot actions triggered by sensors is the ease of transformation of this specification into a robot control program. Figures 7.3 and 7.4 show the listings of procedures implementing reactions coded in pseudo-C. The names of variables have been retained as in the definitions of reactions, so that the

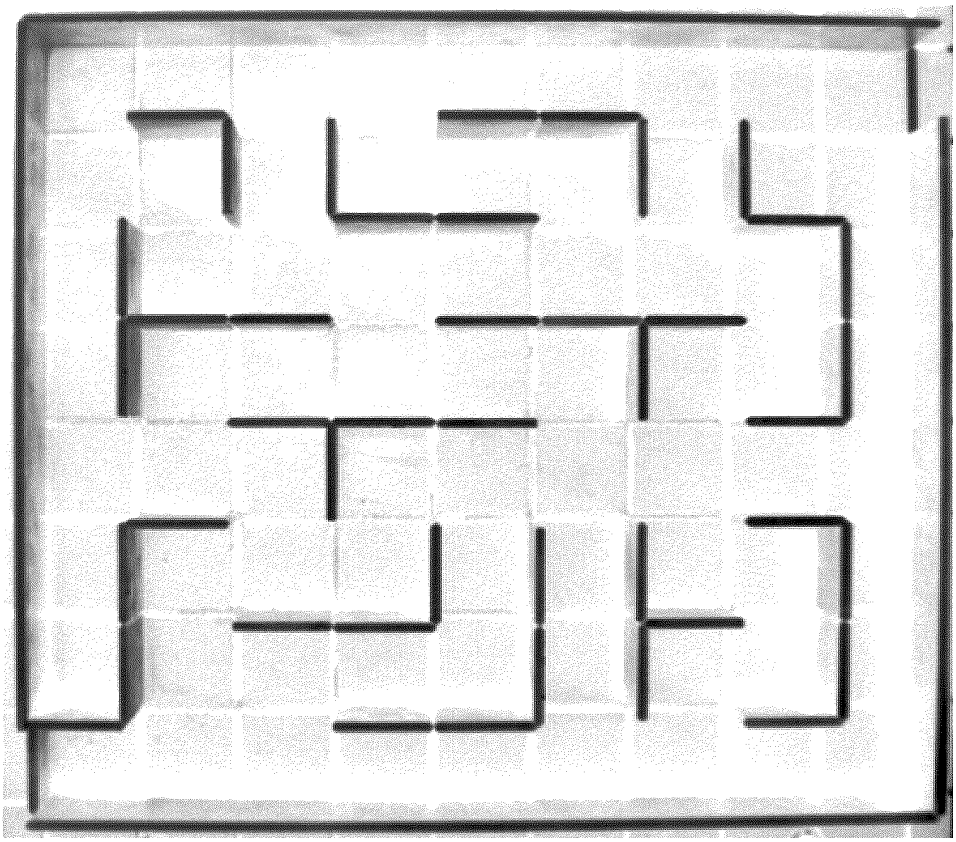

Figure 7.1. A camera view of the exemplary maze

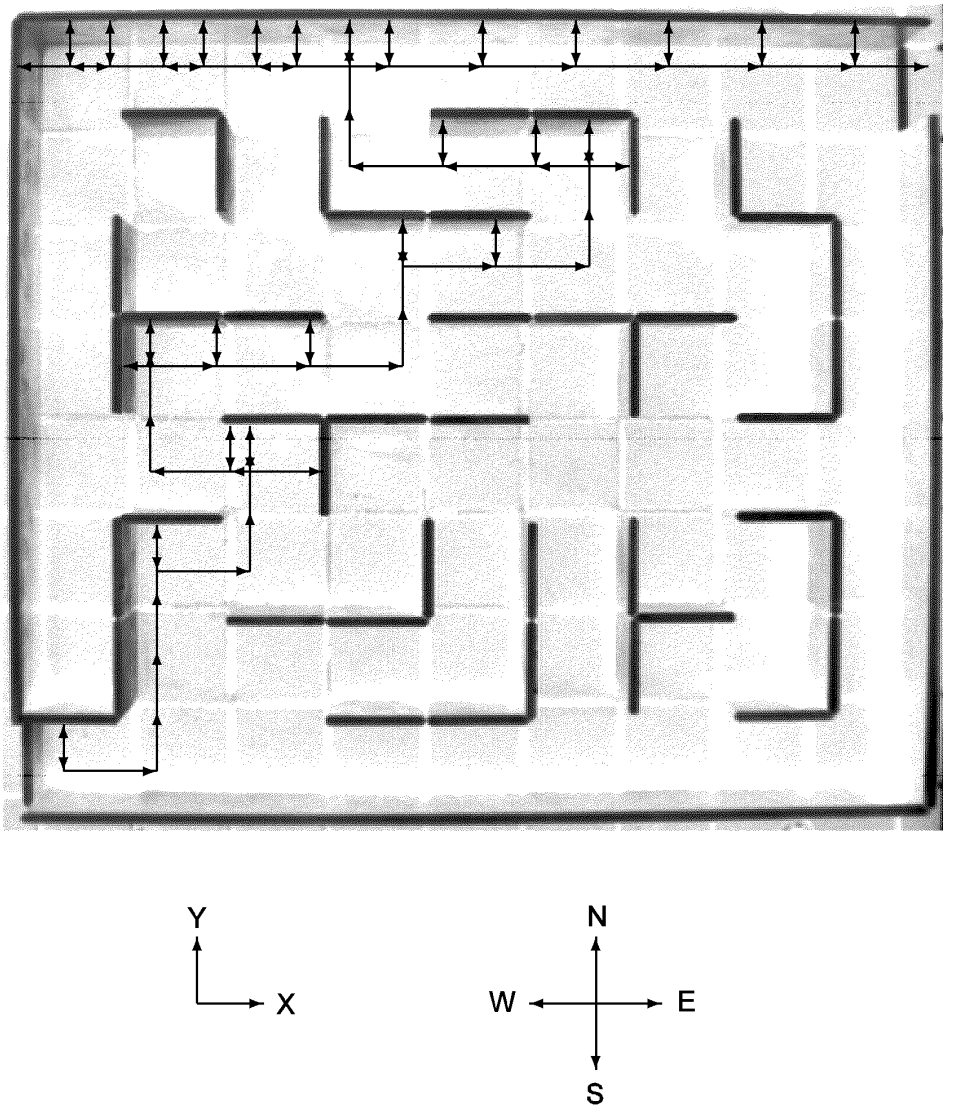

Figure 7.2. The probe trajectory in the exemplary maze (arrow heads indicate the end of each step)

transformation will be rendered obvious, hence pseudo-C and not C. The similarity between the definitions of reactions and the code of the procedures implementing these reactions is obvious. The other advantage of this method is the ability of incremental system design. First, a simple version of the system can be obtained by selecting only a few sub-spaces of the virtual sensor reading space and assigning only a few reactions to them. The remaining portion of the space can be assigned a single reaction (e.g. B_{ERR}). While the system is being developed, new sub-spaces can be extracted from this portion and new reactions can be assigned to them. The previously implemented reactions re-

```
.........
 int              δ^i ;    /* search direction marker */
 double           e^i ;    /* F/T sensor probe location */
 unsigned char    v^i ;    /* virtual sensor reading */
.........
δ^i = 1 ;
get_virtual_sensor_reading ( &v^i ) ;
/* e^i = entry – the F/T sensor probe is at the entrance to the maze */
while ( e^i ∉ E*^iT   /* goal not attained */ )
  {
   switch ( v^i )
        case V0 :   e^i_y += Δe_y ;                          /* continue B0 — */
                    execute_motion_to ( e^i, &v^i ); break;  /* pursue task goal */
        case V1 :   reaction_B_N( e^i, &δ^i, &v^i ); break;  /* avoid northern wall */
        case V2 :   reaction_B_W( e^i, &δ^i, &v^i ); break;  /* avoid western wall */
        case V3 :   reaction_B_E( e^i, &δ^i, &v^i ); break;  /* avoid eastern wall */
        default :   reaction_B_ERR( e^i, &v^i ) ;  exit ;    /* error detected */
    } ; /* end: switch */
  } ; /* end: while */
exit ;
.........



void execute_motion_to ( double e^l,  unsigned char *v )
 {  /* begin: execute_motion_to */
  do
    move_by_an_increment ( Δe ) ;
    get_virtual_sensor_reading ( v ) ;
  until ( *v ∉ V0  or  e = e^l ) ;
  return ;
 } ; /* end: execute_motion_to */
```

Figure 7.3. Pseudo-C code of the relevant sections of the sensorimotor controller executing the maze running task (**part 1**)

main unaltered. Indeed, if a reaction is inadequate, it can be modified or its sub-space further divided and so a single reaction will be divided into several more specific ones.

```
void reaction_B_N( double e, int *δ_N, unsigned char *v )
{ /* begin: reaction_B_N */
  e_y -= Δe'_y ;
  execute_motion_to ( e, v ) ;
  if ( *v ∉ V_0 )
    return ;
  e_x += *δ_N * Δe_x ;
  execute_motion_to ( e, v ) ;
  return ;
} ; /* end: reaction_B_N */


void reaction_B_E( double e, int *δ_E, unsigned char *v )
{ /* begin: reaction_B_E */
  e_x -= Δe_x ; *δ_E = - *δ_E ;
  execute_motion_to ( e, v ) ;
  return ;
} ; /* end: reaction_B_E */


void reaction_B_W( double e, int *δ_W, unsigned char *v )
{ /* begin: reaction_B_W */
  e_x += Δe_x ; *δ_W = - *δ_W ;
  execute_motion_to ( e, v ) ;
  return ;
} ; /* end: reaction_B_W */


void reaction_B_ERR( double e, unsigned char *v )
{ /* begin: reaction_B_ERR */
  e_z += Δe_z ;
  execute_motion_to ( e, v ) ;
  return ;
} ; /* end: reaction_B_ERR */
```

Figure 7.4. Pseudo-C code of the relevant sections of the sensorimotor controller executing the maze running task (**part 2**)

7.2 Complex tasks

The simple maze running task was used as a proof-of-the-concept benchmark for the reactive control concept. This task was implemented on the Research-Oriented Robot Controller RORC [175, 180], which can be tailored exactly to the needs of

the task at hand. The system consisted of a 5 d.o.f. robot, 6 d.o.f. F/T sensor, and an IBM/486/33MHz computer running a multi-process software of the controller. The software was coded in a concurrent version of C. The idea of reactive robot control was further tested on two other tasks [128]:

– complex maze running task, and

– contour-following task.

The complex maze running task is the basic model for any obstacle avoidance tasks in an unknown environment. This task is basically an extension of the simple maze running task onto mazes of any shape. In the simple case the decision where to go next had been made on the basis of the search direction parameter δ. In mazes with cul-de-sacs, during the exploration of the maze its partial map has to be created. That map is the basis for making decisions where to go next. The search direction marker δ is replaced in c_{vv} part of the control subsystem by a data structure (a graph) which is equivalent to the partial map of the maze. In this way the probe does not enter a second time areas which do not lead to the goal. Loops and cul-de-sacs are excluded. Basically, this is a trial-and-error method memorizing the already traversed path.

The contour following task differs from the maze running task in the frequency of invoking reactions. The maze running tasks are discrete event driven. The probe moves in large steps. Collision with an obstacle occurs after many elementary motions (increments) into which steps are divided. In the case of contour following, each step is equivalent to an elementary motion, so this task is, in a way, continuous in nature. The task goal is to reach a certain point on an object with approximately known shape. Once this object is detected, its contour is followed and a small force is exerted on its surface. Whenever the force exceeds a certain limit, it is relaxed by an adequate reaction. When the force falls below a certain limit, to avoid a loss of contact, a reaction increasing this force is invoked. It was found experimentally that the two above-mentioned limits should not be too near each other, as this results in excessive vibrations. Moreover, the step size has to be adequately chosen so as not to excite vibrations and on the other hand not make the overall motion too slow. As the step size has to be kept small, and the smallest duration of elementary motion in RORC is 8 ms, the speed of contour following was rather low. It should be noted that the the contour following task was subdivided into two distinct goals. The first one was to detect the object and the second – to follow its contour. Each of the subtasks had its own reactions associated with its execution.

RORC is not only a good tool for implementing and testing various control ideas, but also its structure is easily extensible into much larger systems than those consisting of a single robot equipped with diverse sensors. The next section will show how this set-up was extended into a multi-robot flexible manufacturing system.

8 Multi-robot systems

8.1 Problem formulation

Several research problems specific to **multi-robot systems** have been identified as important (e.g. [71]):

- trajectory planning ([105, 56]) and obstacle avoidance:
  - collision avoidance (especially with other moving objects, e.g. other robots),
  - detection and representation of obstacles by using aggregated data obtained from different sensors,
  - trajectory generation – including redundant and over-constrained[1] multiple manipulators [105, 56, 129, 130, 131],
- utilization of sensor data,
- dynamics modelling and control strategies:
  - modelling and simultaneous force/position control,
  - adaptive control,
  - coordination of motion of multiple arms [105],
  - parallel algorithms for control,
- software and artificial intelligence,
- control system structures and programming.

This section deals with systems containing: several manipulators, sensors and cooperating devices. The issues of programming such systems, their structure, and synchronisation between their subsystems will be at the focus of attention.

Multi-robot system is a special case of a complex mechatronic[2] system. Complex mechatronic systems (e.g. [137, 138, 70, 66, 114, 30, 85]) require great software implementation effort. It is very important that the software design methodology selected for implementation is easy to use and limits the possibility of introducing bugs into the control program. This implicates proper structuring of software.

Multi-robot systems, and mechatronic systems in general, have several effectors and various sensors, usually controlled by one or more processors. This section presents a generic structure of a mechatronic system taking into account all components of such systems. This structure implicates the structure of the software controlling the functioning of the system.

The idea of distributed software modules controlling complex mechatronic systems is several years old. In [102] the concept of Manipulation Module consisting of an actuator, electronics and control was introduced. Manipulation Modules can be linked with each other and sensors (e.g. position encoders) and so a distributed system comes into existence. Each Manipulation Module becomes a processing node exchanging information with some other nodes in the system. The proposed architecture is implemented on an Inmos transputer system using Occam. Parallel Occam processes communicate

[1] Systems in which the transferred object has less than 6 d.o.f. – usually due to one or more manipulators having less than 6 d.o.f.

[2] Mechatronic approach to designing consists in such distribution of functions throughout the mechanical, electronic and software components of the system that the overall design is the simplest in implementation and best suited to the needs of the user.

through data paths known as channels (in hardware: fast serial duplex links). There exists a Processor Node which is treated as a coordinator of Manipulation Modules. The problems of inter process synchronization are solved by Occam and sensors are not treated comprehensively.

Hierarchic master/slave system architectures (e.g. [7]) are usually considered to be best suited for control of complex mechatronic systems. Hierarchic (vertical) structure is sometimes combined with pipelined (horizontal) architecture to form a ziggurat structure [37]. The proposal of eliminating a Master Process in a master/slave distributed system was discussed in [7]. A system with nodes of equal status results, thus rendering the system uniformly modular, but task coordination becomes more complex, as it has to be distributed throughout the system. Moreover, usually higher communication demand on the system component interconnections results.

As it was mentioned in section 6, the KALI system [54, 55, 6, 105] can control several manipulators transferring a single rigid body. Motions are treated as processes. Synchronisation between the motions of different manipulators is obtained through the combined use of motion control flags and motion parameters such as velocity and time of arrival. Two kinds of processes exist: synchronous and asynchronous ones. Synchronous processes have higher priority than asynchronous. The task of the main synchronous process is to compute the nominal locations for the manipulators, taking into account the frame transformation graphs. Those graphs have a ring structure, with each node representing a transformation between the consecutive elements of the kinematic loop (e.g. manipulator, tool, goal, drive[3] transforms). Other synchronous processes execute the servo-control algorithm. One of the synchronous processes is responsible for gathering sensor information (from position encoders and force sensors). The main asynchronous process contains the robot task. Among others, it sets up the kinematic loops (transformation graphs) and issues the motion requests. The other asynchronous processes compute the dynamics parameters (e.g. gravity terms, inertia matrix, maximum force that the robot can produce, forces created by the velocity terms).

The presented proposal retains the hierarchic structure of master/slave systems. Two types of these structures are discussed. The problems of sensor incorporation and sensor data aggregation are dealt with comprehensively. Moreover, synchronization between processes controlling actuators and processes gathering sensor data is at the focus of this section. Unlike in KALI, processes are associated with effectors rather than with motions.

8.2 Multi-robot system structure

An open mechatronic system, e.g. a multi-robot one, with *a priori* unknown number of effectors is considered [189, 186]. Moreover, in any such system the number and kind of hardware sensors used cannot be determined, as the tasks to be executed are usually variable. The proposed methodology of constructing controllers takes care of two problems. One is: aggregation of data obtained from real sensors; the other – synchronization between sensor data processing and the effector motions.

[3] Relative transformation between the current tool and goal locations – when the tool reaches the goal, this transform becomes an identity.

The raw data has to be processed to obtain the reading of a virtual sensor as defined by (2.2) or in a more comprehensive version by (4.3):

$$v = f_{v_7}(r, e, c) \tag{8.1}$$

The system state s, as defined by (2.1), is decomposed by taking into account that now the system has several distinct effectors and that rather aggregated sensor readings v than real sensor readings r are used by the programmer:

$$s = < e_1, \ldots, e_k, v_1, \ldots, v_n, c > \tag{8.2}$$

where:

k – the number of distinct effectors in the system ($e = [e_1, \ldots, e_k]$),
n – the number of virtual sensors ($v = [v_1, \ldots, v_n]$).

Treating the system as a discrete time system, as in the case of a single manipulator system, the next state of each of the effectors can be computed using a transfer function $f_{e_{7j}}$:

$$e_j^{i+1} = f_{e_{7j}}(e_j^i, v_1^i, \ldots, v_n^i, c^i), \quad j = 1, \ldots, k \tag{8.3}$$

Hence, with each of the effectors $e_j, j = 1, \ldots, k$, an Effector Control Process is associated. Each process is responsible for computing its transfer function $f_{e_{7j}}$ and executing the related motion. The set of functions $f_{e_{7j}}$, $j = 1, \ldots, k$, must be such that the manipulators will move in a coordinated way. Three cases are distinguished.

1. Work spaces of manipulators and their tools do not intersect, and so no collision between them can result – this is the simplest case (control of several separate manipulators) in which functions $f_{e_{7j}}$ are hardly related to each other.
2. Work spaces of manipulators and their tools intersect, so collisions must be avoided and object transfers between the robots can occur – this is an intermediate case in which functions $f_{e_{7j}}$ are loosely related to each other.
3. Work spaces of manipulators and their tools intersect and the robots transfer together a rigid body – this is the most difficult case in which functions $f_{e_{7j}}$ influence each other strongly.

Each virtual sensor $v_l, l = 1, \ldots, n$ is implemented as a process running concurrently to other Virtual Sensor Processes and the Effector Control Processes. In consequence of (8.1)

$$v_l^i = f_{v_7}(r^i, e_m^i, c^i) \tag{8.4}$$

is obtained, where e_m is the state of the m–th effector. Each Effector Control Process creates or kills Virtual Sensor Processes according to the needs of control of motion. The Effector Control Processes in each step i obtain data from the Virtual Sensor Processes. Both kinds of processes can be treated as device dependent drivers. In this way, if only one component of the system is changed, the remaining components remain unaltered.

As all processes require coordination, either an additional **Master Process** can be created (Fig. 8.1) or one of the Effector Control Processes assumes the role of a coordinator and becomes the **Main Motion Process** (Fig. 8.2). In the former case a **hierarchical structure** results and in the latter – a **flat structure** is created. Whenever a mechatronic system has a single effector, the latter case is more suitable, and so this structure has been assumed in the Research-Oriented Robot Controller RORC (section 6). The influence

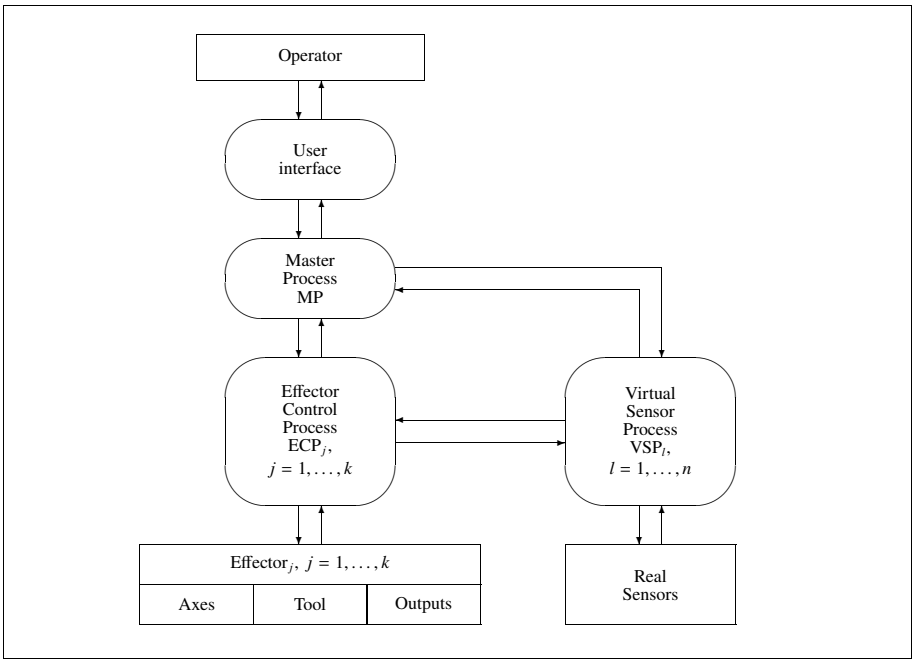

Figure 8.1. Software structure of the system with a Master Process – hierarchical structure

of the Master Process on the next effector state in formula (8.3) is exerted through c^i. The Master Process contains a trajectory generator, common to all robots. The trajectory generator supplies the nominal trajectory through c^i to all Effector Control Processes, so that they can evaluate appropriate functions $f_{e_{7j}}$.

Both the Master Process and the Main Motion Process change the state c of the control subsystem. Virtual Sensor Processes use the memory, and hence c, during sensor data aggregation and for data storage. Effector Control Processes use the memory during motion computations and to keep environment information databases.

The processes communicate through data pipelines or guarded global variables. The communication of each Effector Control Process with the Virtual Sensor Processes it uses can be of two kinds: interactive and non-interactive (Figs. 8.3 and 8.4). In the case of interactive communication the Effector Control Process sends a data request through a data pipeline to an adequate Virtual Sensor Process. The Virtual Sensor Process reads the real sensors, aggregates the obtained data and sends the result through another data pipeline to the Effector Control Process (Fig. 8.3). In the meantime, the Effector Control Process is free to control the effector. If the virtual sensor reading does not arrive before a *receive* operation is executed by the Effector Control Process, the Effector Control Process is suspended until the data arrives. Synchronization, between an Effector Control Process and a Virtual Sensor Process it uses, can be described by a Petri net [111, 52], and is presented in Fig. 6.7.

If several Effector Control Processes use the same Virtual Sensor Process, either several instances of the same Virtual Sensor Process can be created or a single Virtual Sensor Process starts processing sensor data when the first request is made, but it sends

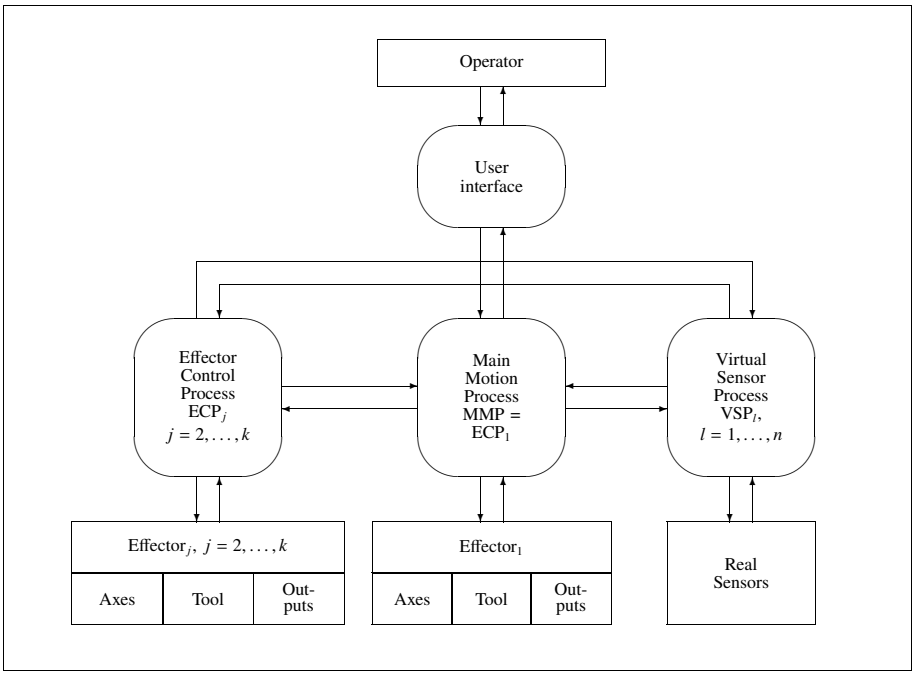

Figure 8.2. Software structure of the system with one of the effector control processes distinguished as a Main Motion Process – flat structure

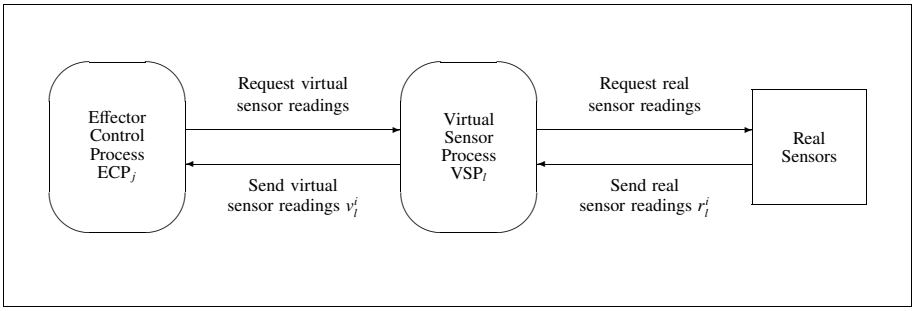

Figure 8.3. Interactive method of reading sensor data through data pipelines

the result to all the Effector Control Processes that have made the requests within a specified time from the initial request (the specified time should be longer than the sensor data acquisition and processing time). In the latter case, prior to sending the resulting reading, the Virtual Sensor Process checks all the data input pipelines for virtual sensor reading requests and sends the obtained result to all data output pipelines that correspond to those data input pipelines that had contained adequate requests.

In the case of non-interactive communication the Virtual Sensor Process is interrupt driven (Fig. 8.4). Whenever the timer issues an interrupt, the Virtual Sensor Process reads the real sensors, aggregates data and leaves the resulting reading in a buffer. In

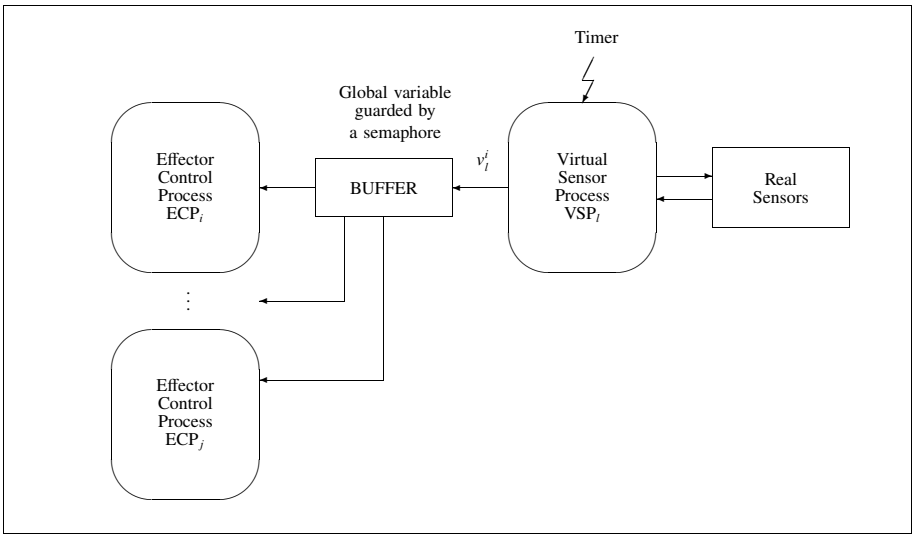

Figure 8.4. Non-interactive method of reading sensor data

this way any Effector Control Process can instantly get the latest reading from the buffer. However, proper synchronization of access to the buffer is necessary. This is done by semaphores on a single producer – several consumers basis. The Petri net picturing proper synchronization of access to the buffer is presented in Fig. 8.5.

A speed-up and a more elegant structure of the software component of the system can be obtained if the Effector Control Process is partitioned into ECP proper and the Effector Driver (Fig. 8.6), and run on two processors. This approach was followed in Multi-Robot Research-Oriented Robot Controller MRROC. The Effector Driver is responsible for:

– transformation of Cartesian-Euler co-ordinates into joint co-ordinates,
– transformation of joint co-ordinates into motor control increments,
– transmission of the set-values to the servo-drives,
– transmission of servo-drive status to upper levels of control structure.

The ECP proper, in this case, is responsible for trajectory generation when functions $f_{e_{7j}}$ are loosely or hardly related to each other. In the case of strong influence of functions $f_{e_{7j}}$ on each other, the ECP proper simply transmits the commands of the Master Process, which acts as a coordinator.

For further speed-up, each Effector Control Process can be partitioned into pipelined stages (not to be mistaken for data pipelines mentioned earlier), i.e. several concurrent processes performing, e.g.: future trajectory position (including orientation) generation taking into account virtual sensor readings, solving inverse kinematics problem (i.e. obtaining joint co-ordinates) and executing the joint control algorithms (i.e. reaching the generated position). In this case three stages suffice. If a three-stage pipelined architecture is used (Fig. 8.7), a delay of two steps is introduced between position generation and its execution, but the step duration can be considerably reduced owing to the parallel action of the three processes. Due to that (8.3) assumes the form:

$$e_j^{i+3} = f_{e_{7j}}(e_j^i, v_1^i, \ldots, v_n^i, c^i), \quad j = 1, \ldots, n \tag{8.5}$$

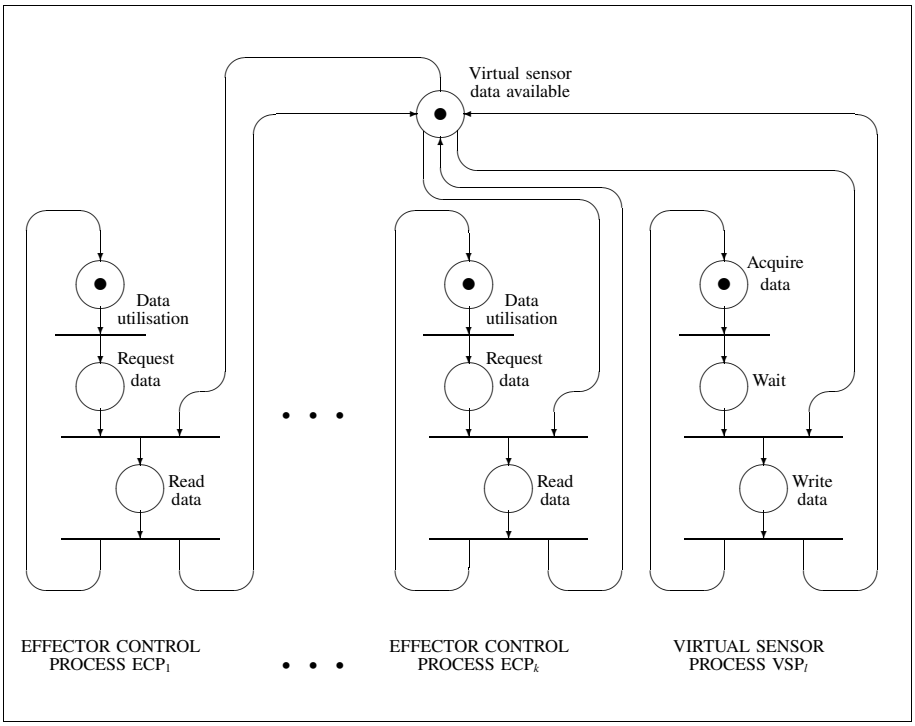

Figure 8.5. Synchronization of several Effector Control Processes with a Virtual Sensor Process (Petri net with initial marking)

In the case of high computational load associated with obtaining a virtual sensor reading (e.g. image processing and pattern recognition), the corresponding Virtual Sensor Process may also be partitioned into a pipelined structure. If the l–th Virtual Sensor Process is partitioned into h stages, the adequate reading v_l will be delayed by $h-1$ steps, and so in (8.5) v_l^i will have the following form:

$$v_l^i = f_{v_7}(r^{i-h+1}, e_m^{i-h+1}, c^{i-h+1}) \tag{8.6}$$

Obviously, only if each stage is processed in parallel to the others the speed-up results. If the processes (stages) time-share a single processor nothing would be gained.

8.3 Current status of the multi-robot system

A software design strategy for multi-robot systems evolved from a single robot system strategy. It consists in associating a concurrent process with each of the distinct effectors of the system (e.g. manipulator arm): the Effector Control Process proper and the Effector Driver (Fig. 8.6). The real sensors are grouped according to their function and the data obtained from them is aggregated by a single process (Virtual Sensor Process). Data obtained from several Virtual Sensor Processes can be transmitted through

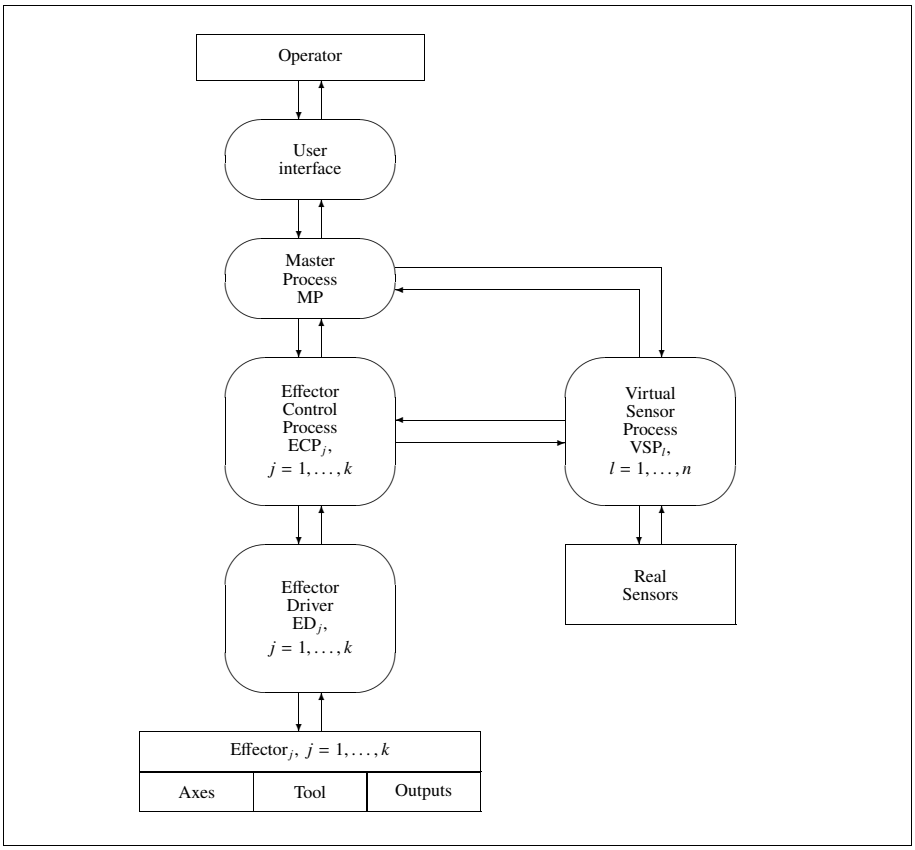

Figure 8.6. Hierarchical structure with an effector driver

data pipelines to adequate Effector Control Processes and the Master Process in an interactive way or stored in guarded buffers. In the latter case the Virtual Sensor Process is timer interrupt driven. This data is used for generating or modifying the trajectories of effector motions. A change in the task that the system has to execute brings about only a realignment of the ready software modules. In extreme cases a new module can be added to the system.

To design the Multi-Robot Research-Oriented Robot Controller MRROC, the single-robot system was enhanced by adding a second 5 d.o.f. robot and a conveyor. In this case the structure as in Fig. 8.6 is used. Each robot and the conveyor are controlled by their own Effector Control Processes and Effector Drivers. The Master Process is responsible both for the communication with the operator (through the user interface process) and coordination of Effector Control Processes. The QNX–4 [191, 52, 119] multi-computer real-time operating system is used for coordinating all processes.

The system was tested on two and tree computer configurations. In the latter case each robot was controlled by a single IBM-PC compatible computer running the associated ECP and ED and the Master Process used the third computer. The computers were connected by Ethernet links. Both synchronous and asynchronous motions of devices

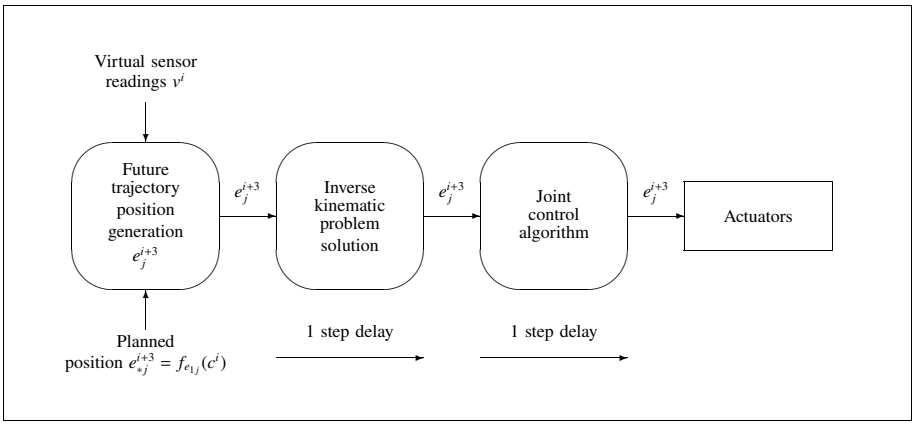

Figure 8.7. Partitioning of an Effector Control Process into a three-stage pipelined structure

were obtained. In the case of synchronous motions, both time and time-space synchronism was achieved. Addition of simple touch and proximity sensors did not require extra computational power. Such complex sensors as camera also do not need extra computational power if the image acquisition and motion decision making are separate. It is envisaged that an extra computer will be necessary if this condition is not true. As the QNX–4 operating system runs processes either in time sharing mode on a single computer or on several computers connected into a network, no change to the existing software will have to be made because of this enhancement.

9 Conclusions

This monograph gives a broad view of robot programming. On-line, off-line and hybrid programming methods are considered. Although on-line programming methods are more popular in industry, their drawbacks, especially the ones associated with sensor integration and long down-times during programming, caused intensive research into off-line methods. Unfortunately, the off-line methods, while on the one hand being a remedy to the on-line method drawbacks, on the other hand cause new problems that did not exist when on-line methods were utilised – especially the calibration problem. A partial solution to this problem was the creation of hybrid programming methods such as employed by VAL II. In the author's opinion both on-line and hybrid programming methods have reached the limits of their capabilities. Systems employing these programming techniques have problems with incorporating new sensors and new control algorithms, due to their closed structure. Only a certain – although broad – class of tasks can be executed by using them. The inability to execute tasks which the definition of the programming method (textual or any other, e.g. teach-pendant language) did not take into account *a priori* is caused mainly by hardware constraints (i.e. the system possesses instructions only for hardware that was available at the time the system was created). New

hardware usually needs new instructions and those cannot be appended by the user of the system. Similar problems, although not as acute, are encountered when using off-line programming methods employing specialised robot programming languages – especially if the RPL is not to be a superset of a CPL.

These problems led to the emergence of a new class of systems with an open structure and off-line methods of programming. Because in this case tasks are expressed in a CPL, for which robot specific instructions are coded as a library of procedures, the implementation effort is much smaller than in all the previous methods. It is much easier to write a library of robot specific procedures than to define a new language and implement its interpreter or a compiler and a run-time system. In this case, moreover, the incorporation of new sensors or new control algorithms causes only an addition of a few procedures. In the former case, the language itself usually had to be modified, and hence reimplemented.

It is a known fact that either all the objects that take part in the task execution have to be positioned very precisely, so the robot can do the job without sensing, or the locations of objects can be known approximately, but then considerable amount of sensing and intelligence is necessary for the accomplishment of the task. Employing diverse external sensors (receptors) can eliminate both prepositioning devices and problems associated with calibration. In the author's view the future is with systems integrating a multitude of diverse sensors and open in structure, so that ever more generic or specific control algorithms can be incorporated.

If robot programming systems are to be relatively simple, they have to possess an open structure, but the structure itself should be rigid. In such systems new procedures can be added, but they have to comply with the overall structure of the system, which has to be defined precisely (e.g. parameter passing, return codes, data transfers). The structures of such systems are an open research problem. This monograph presents the structure of a Research-Oriented Robot Controller RORC, which was implemented at WUT. Upper control layers should also be structured appropriately. One way of doing that is by utilising the reactive control concept.

This work, moreover, deals with the classification of robot motion instructions, their semantics and their implementation. The utilised formalism gives insight into the problem of creating an RPL instruction set and points out the problems of implementing such instructions. The problems associated with implementing joint level, manipulator level and object level RPLs were considered. Different implementation paradigms (structured, object-oriented) were investigated too. This formalism was used by the author to define the semantics of instructions and implement the run-time systems of TORBOL, ROPAS and ROOPL.

This work will be useful to all those who would like to implement an RPL. It points out the problems associated with different approaches to the definition and implementation of the language, especially the motion instructions (e.g. their semantics, level, method of implementation). The syntax of the RPLs has not been the subject of this dissertation, but it is no different to that of CPLs. Instructions utilising sensor information are at the focus of this monograph. The introduced formalism:

– states the instruction semantics without predetermining any particular implementation technique,
– takes into account all system components (i.e. effectors, receptors, and the control subsystem),

– points out what the constraints imposed on the system will be if a set of instructions with certain semantics is included into the currently defined RPL (e.g. what arguments of functions $f_e(\bullet)$, $f_v(\bullet)$, $f_d(\bullet)$ are taken into account, and hence what kind of control will be exerted on the system; will initial, terminal and error condition monitoring be possible),
– can be incorporated into flow-diagrams to render the implementation of the system software straightforward,
– enables the specification of upper-layers of control system structure (e.g. reactive robot control).

This formalism was used to define the software structure of the Research-Oriented Robot Controller which later evolved into a multi-robot control system. It points out what should be the formal parameter lists of procedures implementing different functions (e.g. sensor data aggregation, motion instructions). RORC, in turn, was used as a test bed for designing new control algorithms (e.g. sensor-based reactive robot control).

The future work will concentrate on transforming the multi-robot system into a full--blown flexible manufacturing cell. The introduced formalism is sufficient to describe the functioning of most mechatronic systems, so it will be the foundation of the cell description. The experience gained during the implementation of TORBOL, ROPAS and ROOPL was utilised in construction of RORC. RORC, in turn, was the predecessor of the multi-robot system. This evolutionary approach leads to the development of efficient ways of designing large distributed computer-controlled systems incorporating: robots, machining tools and diverse sensors.

As it was mentioned in subsection 1.2, the goal of this monograph has been to describe the programming of robots from the point of view of users and implementers of robot systems. This has been achieved, principally, through the description of the author's own research of the problem, but also by a detailed analysis of the work of other scientists, whose ideas were distributed throughout books, dissertations, conference proceedings and journal papers. This dissertation, besides presenting the author's views on the subject, brings together the knowledge which has been otherwise spread in the vast literature concerning robotics, mechatronics, automatic control and computer science. Both theoretical considerations and practical implications have been discussed. Moreover, the author's evolution of ideas has been presented. Starting with the analysis of the literature on the subject, theoretical development of useful formalism, through the definition and implementation of different robot programming languages (TORBOL, ROPAS, ROOPL), to open research-oriented robot control systems (RORC), which finally evolved into a full-blown multi-device, multi-robot and multi-sensor programming system.

Bibliography

[1] Ambler A. P., Popplestone R. J., Kempf K. G.: *An Experiment in the Off-line Programming of Robots*. Proceedings of the 12th International Symposium on Industrial Robots, Paris, 1982.

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

[3] Andersson R. L.: *Real-Time Gray-Scale Video Processing Using a Moment-Generating Chip*. IEEE Journal of Robotics and Automation, Vol.1, No.2, June 1985.

[4] Andersson R. L.: *A Robot Ping-Pong Player*. The MIT Press, Cambridge 1988.

[5] Asada H., Slotine J. J.: *Robot Analysis and Control*. John Wiley & Sons Inc., New York, 1986.

[6] Backes P., Hayati S., Hayward V., Tso K.: *The KALI Multi-Arm Robot Programming and Control Environment*. Proceedings of the NASA Conference on Space Telerobotics, 1989.

[7] Backman U. H., Lind H. A., Törngren E. M., Wikander J., Kuroki M.: *A Fully Distributed Real-Time Control System*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.179-188.

[8] Bartol W. M. et. al.: *Report on the LOGLAN 82 Programming Language*. Institute of Informatics, The University of Warsaw, Warsaw, June 1982.

[9] Bezi I., Tevesz G.: *PC Based Robot Controller for Advanced Control Algorithms*. **In:** *Mechatronics: the basis for new Industrial Development*. Eds: M. Acar, J. Makra, E. Penney, Computational Mechanics Publications. Southampton, Boston, 1994. Joint Hungarian--British Mechatronics Conference, 21-23 September 1994, Budapest, Hungary, pp.743-748.

[10] Blume C.: *A Structured Way of Implementing the High Level Programming Language AL on a Mini– and Microcomputer Configuration*. Proceedings of the 11th International Symposium on Industrial Robots, Tokyo, 7-9 October, 1981.

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

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

[13] Bonner S., Shin K. G.: *A Comparative Study of Robot Languages*. Computer (IEEE), Vol. 15, No. 12, December 1982.

[14] Braganca C. A., Saunders D. A.: *VAL II Based Software for Communicating to Sensors, PLCs, and Networks, Using Standard Defined Interfaces*. Proceedings of the 16th International Symposium on Industrial Robots, Brussels, 1986.

[15] Brooks R. A.: *Intelligence Without Representation*. Artificial Intelligence, No.47, 1991. pp.139-159.

[16] Brooks R. A.: *The Whole Iguana*. Report of Workshop on Coordinated Multiple Robot Manipulators: Planning, Control, and Applications. Eds: G. Koivo, G. A. Bekey, 7-9 January 1987.

[17] Brooks R. A.: *A Robust Layered Control System For a Mobile Robot*. IEEE Journal of Robotics and Automation, Vol. RA-2, No.1, March 1986. pp.14-23.

[18] Burdea G. C., Wolfson H. J.: *Solving Jigsaw Puzzles by a Robot*. IEEE Transactions on Robotics and Automation, Vol.5, No.6, December 1989.

[19] Caldwell D. G., Gray J. O.: *'Dynamic' Multi-Functional Tactile Sensing*. Proceedings of the 9th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'92, 1-4 September 1992, Udine, Italy, Lecture Notes in Control and Information Sciences 187, Springer-Verlag, Berlin 1993.

[20] Chang P. R., Lee C. S.: *Residue Arithmetic VLSI Array Architectures for Manipulator Pseudo-Inverse Jacobian Computation*. IEEE Transactions on Robotics and Automation, Vol.5, No.5, October 1989.

[21] Cheung E., Lumelsky V. J.: *Proximity Sensing in Robot Manipulator Motion Planning: System and Implementation Issues*. IEEE Transactions on Robotics and Automation, Vol.5, No.6, December 1989. pp.740-751.

[22] Connel J. H.: *A Behavior-Based Arm Controller*. IEEE Transactions on Robotics and Automation, Vol.5, No.6, December 1989. pp.784-791.

[23] Corke P., Kirkham R.: *The ARCL Robot Programming System*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.484-493.

[24] Craig J. J.: *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Publishing Company, Reading 1986.

[25] Critchlow A. J.: *Introduction to robotics*. Macmillan Publishing Company, New York, 1985.

[26] Dario P., Buttazzo G.: *An Anthropomorphic Robot Finger for Investigating Artificial Tactile Perception*. International Journal of Robotics Research, Vol.6, No.3, Fall 1987. pp.25-48.

[27] Dembiński P., Małuszyński J.: *Matematyczne metody definiowania języków programowania*. Wydawnictwa Naukowo-Techniczne, Warszawa1981.

[28] Denavit J., Hartenberg R. S.: *A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices*. Journal of Applied Mechanics, June 1955. pp.215-221.

[29] Deveza R., Russell A., Thiel D.: *Robot Navigation by Smell: Problems and Some Solutions*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.458-466.

[30] Dinsdale J. D., Hutcheon A. J.: *Self-service Banking and Mechatronics: The First Twenty Years*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.149-158.

[31] Durrant-Whyte H. F.: *Consistent Integration and Propagation of Disparate Sensor Observations*. International Journal of Robotics Research, Vol.6, No.3, Fall 1987.

[32] Dutkiewicz P., Kozłowski K. R., Wróblewski W. S.: *System programowania robota do celów badawczych*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.335--386.

[33] Dutkiewicz P., Kozłowski K. R., Wróblewski W. S.: *Programming system for the IRp-6 robot*. Archives of Control Sciences, Vol.3 (XXXIX) no.1-2, 1994, pp.51-68.

[34] Dutkiewicz P., Królikowski A., Kozłowski K. R., Wróblewski W. S.: *System programowania robota IRp-6, jego model oraz sprzężenie z czujnikami do pomiaru sił i momentów*. **In:** PIAP Bulletin: *Projekty badawcze – granty w dziedzinie robotyki*. Proceedings of a robotics seminar funded by Polish Scientific Research Committee, 7-8 December 1993, PIAP, Warsaw 1994, pp.51-63.

[35] Engelberger J. F.: *Robotics for the Service Sector*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.3-19.

[36] Finlay P. A.: *Robotic Systems for Health and Retirement Care*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.104-111.

[37] Foster K., Fenney L.: *A Control Structure for Flexible High-Speed Machinery*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.189--194.

[38] Fu K. S., Gonzalez R. C., Lee C. S. G.: *Robotics: Control, Sensing, Vision, and Intelligence*. McGraw-Hill Inc., 1987.

[39] Funda J., Taylor R. H., Paul R. P.: *On Homogeneous Transforms, Quaternions, and Computational Efficiency*. IEEE Transactions on Robotics And Automation, Vol.6, No.3, June 1990. pp.382-388.

[40] Galicki M.: *Wykorzystanie rozwiązania zadania odwrotnego kinematyki do planowania bezkolizyjnych trajektorii manipulatorów*. Archiwum Automatyki i Telemechaniki, Vol.34, No.3-4, 1989.

[41] Gini G., Gini M.: *POINTY. A Philosophy in Robot Programming*. Information Control Problems In Manufacturing Technology, Proceedings of the 2nd IFAC/IFIP Symposium, 1979.

[42] Gini G., Gini M., Somalvico M.: *Program Abstraction and Error Correction in Intelligent Robots*. Proceedings of the 10th International Symposium On Industrial Robots, Milan, 1980.

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

[44] Gosiewski A., Hajdukiewicz M.: *Układ sterowania czaso-optymalnego dla robota URp-6*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.117-125.

[45] Graham J. H.: *Special Computer Architectures for Robotics: Tutorial and Survey*. IEEE Transactions on Robotics and Automation, Vol.5, No.5, October 1989.

[46] Grocholewski S.: *Możliwości wejścia głosowego w sterowaniu robotem*. Proceedings of the 1st National Conference on Robotics, Wrocław, Vol.3, 18-20 September 1985, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1985. pp.33-38.

[47] Grodecki A., Gosiewski A.: *Hybrydowy sterownik siły/położenia dla robota IRp-6. Cz.I: opis kinematyczny*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.126--133.

[48] Grodecki A.: *Hybrydowy sterownik siły/położenia dla robota IRp-6. Cz.II: struktura układu i wyniki eksperymentalne*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.134-141.

[49] Grodecki A., Niezgoda J.: *Implementacja sprzężeń "feedforward" od prędkości i przyspieszenia w robocie IRp-6*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.142-149.

[50] Grodecki A.: *Analiza porównawcza układów sterowania siłowego robotów przemysłowych*. Internal memo, Institute of Automatic Control, Warsaw University of Technology, May 1993.

[51] Gruver W. A., Soroka B. I., Craig J. J., Turner T. L.: *Evaluation of Commercially Available Robot Programming Languages*. Proceedings of the 13th International Symposium on Industrial Robots, Chicago, 1983.

[52] Halang W. A., Sacha K. M.: *Real-Time Systems: Implementation of Industrial Computerised Process Automation*. World Scientific Publishing Co., Singapore 1992.

[53] Hayward V., Paul R. P.: *Robot Manipulator Control Under Unix RCCL: A Robot Control C Library*. International Journal of Robotics Research, Vol.5, No.4, Winter 1986. pp.94-111.

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

[55] Hayward V., Daneshmend L., Hayati S.: *An Overview of KALI: A System to Program and Control Cooperative Manipulators*. **In:** *Advanced Robotics*. Ed.: K. Waldron, Springer Verlag, Berlin 1989.

[56] Hayward V.: *Aspects of the Control of Complex Mechanical Systems with Time-Varying Topologies*. Proceedings of the 8th World Congress on the Theory of Machines and Mechanisms, Prague, Czechoslovakia, August 26-31, 1991, pp.639-642.

[57] Hayward V.: *Trajectory generation and Control for Automatic Manipulation*. Robotica, 1992.

[58] Hirose S.: *Wall Climbing Vehicle Using Internally Balanced Magnetic Unit*. Preprints of the proceedings of the 6th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'86, 9-12 September 1986, Cracow, Poland, SOW ZSP Alma--Press. pp.363-370.

[59] Hirzinger G.: *Issues in Low-Dimensional Sensing and Feedback*. IEEE Transactions on Systems, Man, and Cybernetics, Vol.19, No.4, July/August 1989. pp.832-839.

[60] Hollerbach J. M.: *A Survey of Kinematic Calibration*. **In:** *The Robotics Review 1*. Eds: O. Khatib, J. J. Craig, T. Lozano-Perez, The MIT Press, Cambridge 1989.

[61] Innocenti C., Parent-Castelli V.: *Analytical Form Solution of the Direct Kinematics of a 4-4 Fully-Parallel Actuated Six Degree-of-Freedom Mechanism*. Proceedings of the 9th CISM--IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'92, 1-4 September 1992, Udine, Italy, Lecture Notes in Control and Information Sciences 187, Springer-Verlag, Berlin 1993.

[62] Jacak W., Rogaliński P., Łysakowska B.: *Planowanie optymalnych trajektorii ruchu robota*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.43-50.

[63] Jacak W., Łysakowska B.: *Wielopoziomowy system planowania działań i ruchów robota*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.2, pp.184-189.

[64] Jacak W.: *Roboty inteligentne – metody planowania działań i ruchów*. Prace Naukowe ICT, Vol.85, Wydawnictwo Politechniki Wrocławskiej, Wrocław, 1991.

[65] Jacak W., Rogaliński P.: *ICARS – system do syntezy inteligentnego sterowania robotami w gnieździe produkcyjnym*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.2, pp.317-323.

[66] Jacak W., Rozenbilt J., Sierocki J.: *Simulation Based Intelligent Control Design in Manufacturing Automation*. Proceedings: International Workshop on Intelligent Robotic Systems. Zakopane, July 20-24, 1993. Eds: J. L. Crowley, A. Dubrawski, (Warsaw-Grenoble), pp.81--90.

[67] Kasahara H., Narita S.: *Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System*. IEEE Journal of Robotics and Automation, Vol.1, No.2, June 1985

[68] Kasiński A., Drapikowski P.: *Komputerowe planowanie zadań robota z wykorzystaniem interfejsu graficznego*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.2, pp.203-209.

[69] Kernighan B. W., Ritchie D. M.: *The C Programming Language*. Prentice-Hall, Englewood Cliffs 1980.

[70] Khodabandehloo K., Brett P.: *Intelligent Robot Systems for Automation in the Food Industry*. Proceedings of the Institution of Mechanical Engineers. International Conference. Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.247-254.

[71] Koivo A. J., Bekey G. A.: *Report of Workshop on Coordinated Multiple Robot Manipulators: Planning, Control, and Applications*. IEEE Journal of Robotics and Automation, Vol.4, No.1, February 1988, pp.91-93.

[72] Koivo A. J.: *Fundamentals for Control of Robotic Manipulators*. John Wiley & Sons Inc., New York, 1989.

[73] Kozłowski K. R.: *POLROB – a manipulator level programming language based on Pascal*. Journal of Microcomputer Applications, no.14, 1991, pp.49-60.

[74] Kozłowski K. R.: *Rozwiązanie zagadnień dynamiki odwrotnej oraz prostej w oparciu o klasyczny aparat teorii sterowania, cz.I*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.2, pp.223-228.

[75] Kozłowski K. R.: *Rozwiązanie zagadnień dynamiki odwrotnej oraz prostej w oparciu o klasyczny aparat teorii sterowania, cz.II*. Proceedings of the 3rd National Conference on

Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.2, pp.229-235.

[76] Kręglewska U.: *Interfejs bezpośredniego dostępu IBM AT do zasobów sterownika robota IRp-6*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.90-95.

[77] Kuzan P., Pilat Z., Malinowski K.: *Analiza własności manipulatora robota RIMP-1000 i adaptacyjne algorytmy sterowania jego ruchem*. Archiwum Automatyki i Telemechaniki, Vol.32, no.1-2, 1987.

[78] Latombe J. C., Mazer E.: *LM: A High-Level Programming Language for Controlling Assembly Robots*. Proceedings of the 11th International Symposium On Industrial Robots, Tokyo, 1981.

[79] Latombe J. C.: *Survey of Advanced General-Purpose Software for Robot Manipulators*. Proceedings of the 1982 CERN School of Computing.

[80] Lee C. S. G.: *Robot Arm Kinematics and Dynamics*. **In:** *Advances in Automation and Robotics*, Vol.1, JAI Press Inc., 1985.

[81] Leniowski R.: *Krępy – robot laboratoryjny o sterowaniu cyfrowym, charakterystyka urządzenia*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.2, pp.250-255.

[82] Leniowski R., Irzeński W., Trybus L.: *Wieloprocesorowy 32-bitowy układ sterowania robota laboratoryjnego*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.2, pp.34-42.

[83] Lieberman L. I., Wesley M. A.: *AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly*. IBM Journal of Research and Development, Vol.21, No.4, July 1977.

[84] Lloyd J., Parker M., McClain R.: *Extending the RCCL Programming Environment to Multiple Robots and Processors*. Proceedings of the IEEE International Conference on Robotics and Automation, 1988. pp.465-469.

[85] Loose D. C., Colson J. C.: *PPA – A Precise, Data Driven Component Tool*. IEEE Robotics and Automation Magazine, Vol.1, No.1, March 1994, pp.6-12.

[86] Lozano-Perez T.: *Robot Programming*. Proceedings of the IEEE, Vol. 71, No. 7., July 1983.

[87] Lumia R., Albus J. S.: *Teleoperation and Autonomy for Space Robotics*. Robotics and Autonomous Systems, Vol.4, No.1, March 1988. pp.27-33.

[88] Luh J.: *An Anatomy of Industrial Robots and Their Controls*. IEEE Transactions On Automatic Control, Vol.AC-28, No. 2, February 1983.

[89] Lumelsky V. J.: *Complexity of Sensor-Based Robot Motion Planning*. Proceedins of the International Meeting "Advances in Robot Kinematics", Ljubljana, Yugoslavia, September 19-21, 1988. pp.74-81.

[90] Luo R. C., M-H. Lin, Scherp R. S.: *Dynamic Multi-Sensor Data Fusion System for Intelligent Robots*. IEEE Journal of Robotics and Automation, Vol.4, No.4, August 1988.

[91] Luo R. C., Kay M. G.: *Multisensor Integration and Fusion in Intelligent Systems*. IEEE Transactions on Systems, Man, and Cybernetics, Vol.19, No.4, July/August 1989.

[92] Malcolm C., Smithers T.: *Programming Assembly Robots in Terms of Task Achieving Behavioural Modules: First Experimental Results*. Department of Artificial Intelligence, Research Paper No.410, University of Edinburgh, 1988.

[93] Malec J., Wnuk M.: *Sztuczna inteligencja w robotyce*. **In:** *Robotyka. Podstawowe Problemy Współczesnej Techniki*, T.25. Ed.: A. Morecki, Państwowe Wydawnictwo Naukowe, Warszawa1987. pp.55-96.

[94] Mataric M. J.: *Integration of Representation Into Goal-Driven Behavior-Based Robots*. IEEE Transaction on Robotics and Automation, Vol.8, No.3, June 1992. pp.304-312.

[95] Miyatake Y., Fujimori T., Ueno T.: *Automation and Robotics in Construction Industry*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.130-141.

[96] Morecki A., Ekiel J., Fidelus K.: *Cybernetyczne systemy ruchu kończyn zwierząt i robotów*. Państwowe Wydawnictwo Naukowe, Warszawa1979.

[97] Morecki A., Marszalec E., Marszalec J.: *Theoretical and Experimental Investigations of Optical Fibre Reflective Sensors for Robotics*. Preprints of the proceedings of the 6th CISM--IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'86, 9-12 September 1986, Cracow, Poland, SOW ZSP Alma-Press. pp.219-226.

[98] Morecki A.: *Manipulatory elastyczne*. **In:** *Podstawy robotyki – teoria i elementy manipulatorów i robotów*. Eds: A. Morecki, J. Knapczyk, Wydawnictwa Naukowo-Techniczne, Warszawa1993. pp.512-550.

[99] Morecki A., Knapczyk J., Galicki M., Marszalcowie E. i J., Wiliński A., Zielińska T.: *Podstawy Robotyki – teoria i elementy manipulatorów i robotów*. Eds: A. Morecki, J. Knapczyk, Wydawnictwa Naukowo-Techniczne, Warszawa, 1994.

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

[101] Mujtaba M. S.: *Current Status of the AL Manipulator Programming System*. Proceedings of the 10th International Symposium on Industrial Robots, Milan, Italy, 5-7 March, 1980.

[102] Naghdy F., Strickland P.: *A Distributed Architecture for Parallel Control of Advanced Production Machines*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.65-70.

[103] Nanua P., Waldron K. J., Murthy V.: *Direct Kinematic Solution of a Stewart Platform*. IEEE Transactions on Robotics and Automation, Vol.6, No.4, August 1990.

[104] Narasimhan S., Siegel D. M., Hollerbach J. M.: *CONDOR: An Architecture for Controlling the Utah-MIT Dexterous Hand*. IEEE Transactions on Robotics and Automation, Vol.5, No.5, October 1989.

[105] Nilakantan A., Hayward V.: *The Synchronisation of Multiple Manipulators in Kali*. Robotics and Autonomous Systems, No.5, Elsevier Science Publishers, 1989. pp.345-358.

[106] Orin D. E., Sadayappan P., Ling Y-L. C., Olson K. W.: *Robotics Vector Processor Architecture for Real-Time Control*. **In:** *Sensor-Based Robots: Algorithms and Architectures*. Ed.: C. S. G. Lee, NATO ASI Series, Vol.F66, Springer-Verlag, Berlin 1991.

[107] Orin D. E.: *Parallel Algorithms for Computation of Manipulator Jacobian*. Proceedings of the International Meeting "Advances in Robot Kinematics", Ljubljana, Yugoslavia, September 19-21, 1988. pp.95-102.

[108] Pachuta M., Jabłoński P., Wawerek Z.: *Cyfrowe sterowniki dla napędów robotów przemysłowych*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.2, pp.51-64.

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

[110] Paul R.: *Robot Manipulators: Mathematics, Programming, and Control*. The MIT Press, Cambridge, 1982.

[111] Peterson J. L.: *Petri Net Theory and Modeling of Systems*. Prentice-Hall Inc., Englewood Cliffs 1981.

[112] Popplestone R. J., Ambler A. P., Bellos I.: *RAPT: A Language for Describing Assemblies*. The Industrial Robot, September, 1978. pp.131-137.

[113] Popplestone R. J., Ambler A. P., Bellos I.: *Formation of Body Models and Their Use in Robotics*. Proceedings of the 8th International Symposium on Industrial Robots, Vol. 1, Stuttgart, 1978.

[114] Purves W. K., Wright D. A.: *Microprocessors on Co-ordinate Measuring Machines*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers), pp.37-46

[115] Raibert M. H., Tanner J. E.: *Design and Implementation of a VLSI Tactile Sensing Computer*. International Journal of Robotics Research, Vol.1, No.3, 1982. pp.3-18.

[116] Reisig W.: *Sieci Petriego*. Wydawnictwa Naukowo-Techniczne, Warszawa 1989.

[117] Rogaliński P.: *An Approach to Automatic Robots Programming in the Flexible Manufacturing Cell*. Robotica, Vol.12, part 3, 1994, pp.263-279.

[118] Russell R. A.: *Giving Robots a Sense of Touch: An Overview*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.443-449.

[119] Sacha K. M.: *Systemy czasy rzeczywistego*. Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1993.

[120] Sadayappan P., Ling Y-L. C., Olson K. W., Orin D. E.: *A Restructurable VLSI Robotics Vector Processor Architecture for Real-Time Control*. IEEE Transactions on Robotics and Automation, Vol.5, No.5, October 1989.

[121] Saito F., Fukuda T., Arai F.: *Swing and Locomotion Control of a Two-Link Brachiation Robot*. IEEE Control Systems, Vol.14, No.1, February 1994, pp.5-12.

[122] Schraft R. D., Kaun R.: *New Robot Applications in Manufacturing*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.142-162.

[123] Selke K.: *A Framework for Intelligent Robotic Assembly*. Mechatronic Systems Engineering, Vol.1, No.1, 1990. pp.41-51.

[124] Sevila F.: *Robotics in Agriculture Through the Fruits and Vegetables Harvesting Developments*. Proceedings of the International Conference on Robots for Competitive Industries, Brisbane, Queensland, Australia, 14-16 July 1993. pp.83-103.

[125] Soroka B. I.: *What Can't Robot Languages Do?* Proceedings of the 13th International Symposium on Industrial Robots, Chicago, 1983.

[126] Szacka K.: *Zastosowanie odkształcania trajektorii zadanej w układach sterowania robotów przemysłowych*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.290--297.

[127] Szkodny T.: *Planowanie sterowań i trajektorii minimalnoczasowych manipulatorów robotów przemysłowych*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.298--316.

[128] Szymczak M.: *Algorytmy sterowania reaktywnego robotem wyposażonym w czujnik sił*. M.Sc. Thesis, Supervised by C. Zieliński, Institute of Control and Computation Engineering, Department of Electronics and Information Technology, Warsaw University of Technology, Warsaw 1995.

[129] Szynkiewicz W., Gosiewski A.: *Constraints Satisfaction Approach to Admissible Path Determination for Two Cooperating Robot Arms*. Archives of Control Sciences, No.1, 1993, pp.119-137.

[130] Szynkiewicz W.: *Admissible Path Planning for Two Cooperated Robot Arms*. MELECON'94 (IEEE Mediterranean Electrotechnical Conference), Antalaya, Turkey, April 1994, pp.699--702.

[131] Szynkiewicz W., Gosiewski A.: *Coordinated Trajectories Planning for Two Cooperating Robots*. 1st IFAC Workshop on New Trend in Design of Control Systems, Smolenice, Slovak Republic, September 1994.

[132] Śluzek A.: *Komputerowa analiza obrazów*. Wydawnictwa Politechniki Warszawskiej, Warszawa 1991.

[133] Tadeusiewicz R.: *Systemy wizyjne robotów przemysłowych*. Wydawnictwa Naukowo-Techniczne, Warszawa1992.

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

[135] Todd D. J.: *Walking Machines: An Introduction to Legged Robots*. Kogan Page, London 1985.

[136] Topper A.: *A Computing Architecture for a Multiple Robot Controller*. M.Sc. Thesis, Department of Electrical Engineering, McGill University, Montreal, Canada, June 1991.

[137] Trabasso L. G., Hewit J. R., Slade A. P.: *Robotics Applied in the Decoration of Scale Models*. Proceedings of the 8th CISM IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'90, 2-6 July 1990, Cracow, Poland.

[138] Trabasso L. G., Hewit J. R.: *Artificial Intelligence Techniques Applied to Robotic Decoration of Scale Models*. International Conference – Mechatronics: Designing Intelligent Machines. Cambridge, United Kingdom, 12-13 September, 1990, (Proceedings of the Institution of Mechanical Engineers) pp.165-170.

[139] Trevelyan J. P.: *Sensing and Control for Sheep-Shearing Robots*. IEEE Transactions on Robotics and Automation, Vol.5, No.6, December 1989.

[140] Trybus L., Leniowski R., Irzeński W.: *Wieloprocesorowy 32-bitowy układ sterowania robota laboratoryjnego*. **In:** PIAP Bulletin: *Projekty badawcze – granty w dziedzinie robotyki*. Proceedings of a robotics seminar funded by Polish Scientific Research Committee, 7-8 December 1993, PIAP, Warsaw 1994, pp.41-50.

[141] Van Brussel, Staszulonek A.: *Multiprocessor Controller for Advanced Robotic Applications – Hardware Structure*. Proceedings of the 2nd National Conference on Robotics, Wrocław, 21-23 September 1988, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1988. Vol.1, pp.83-99.

[142] Vukobratowić M., Kirćanski N., Lee S. G.: *An Approach to Parallel Processing of Dynamic Robot Models*. International Journal of Robotics Research, Vol.7, No.2, April 1988.

[143] Verschure P. F. M. J., Krose B. J. A., Pfeifer R.: *Distributed Adaptive Control: The Self-Organization of Structured Behavior*. Robot&Autonomous System, No.9, 1992. pp.181-196.

[144] Wang Y., Butner S.: *RIPS: A Platform for Experimental Real-Time Sensory-Based Robot Control*. IEEE Transactions on Systems, Man, and Cybernetics, Vol.19, No.4, July/August 1989. pp.853-860.

[145] Warczyński J., Romanowski K.: *Paralelizacja obliczeń kinematyki i dynamiki manipulatora*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.202-209.

[146] Watanabe T., Shimoyama I., Miura H.: *Study on Group-Behavior Control of Microrobots*. Journal of Robotics and Mechatronics, Vol.4, No.3, 1992. pp.223-229.

[147] Whitney D. E.: *Historical Perspective and State of the Art in Robot Force Control*. International Journal of Robotics Research, Vol.6, No.1, Spring 1987, pp.3-14.

[148] Witkowski C. M., Bond A. H., Burton M.: *The Design of Sensors for a Mobile Teleoperator Robot*. **In:** *Computing Techniques for Robots*. Ed.: I. Alexander, Kogan Page, London 1985. pp.58-84.

[149] Wood B. O., Fugelso M. A.: *MCL, The Manufacturing Control Language*. Proceedings of the 13th International Symposium On Industrial Robots, Chicago, 1983.

[150] Wu J., Chan C.: *Isolated Word Recognition by Neural Network Models with Cross-Correlation Coefficients for Speech Dynamics*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol.15, No.11, November 1993. pp.1174-1185.

[151] Woźniak A., et. al.: *Autonomiczne roboty mobilne*. Ed.: A. Woźniak, Wydawnictwo Politechniki Poznańskiej, Poznań 1994.

[152] Yamada Y., Shin K., Tsuchida N., Komai M.: *A Tactile Sensor System for Universal Joint Sections of Manipulators*. IEEE Transactions on Robotics and Automation, Vol.9, No.4, August 1993. pp.512-517.

[153] Yin B.: *SensoryRAPT User's Manual*. Department of Computing Science, University of Aberdeen, 1989.

[154] Yoshikawa T.: *Foundations of Robotics: Analysis and Control*. The MIT Press, Cambridge 1990.

[155] Zheng Y. F.: *Integration of Multiple Sensors into a Robotic System and its Performance Evaluation*. IEEE Transactions on Robotics and Automation, Vol.5, No.5, October 1989.

[156] Zielińska T.: *Maszyny kroczące*. **In:** *Podstawy robotyki: teoria i elementy manipulatorów i robotów*. Eds: A. Morecki, J. Knapczyk, Wydawnictwa Naukowo-Techniczne, Warszawa1993. pp.551-596.

[157] Zieliński C.: *Roboty – Proste języki programowania*. Informatyka nr 7, 1984.

[158] Zieliński C.: *Roboty – Złożone języki programowania*. Informatyka nr 8, 1984.

[159] Zieliński C.: *Semantics of the low level robot language instructions*. Proceedings of the 15th International Symposium on Industrial Robots, Tokyo 1985, pp.985-993.

[160] Zieliński C., Śluzek A.: *New data types for 4/5 degree of freedom robot manipulators*. Proceedings of the 15th International Symposium on Industrial Robots, Tokyo 1985, pp.1067--1073.

[161] Zieliński C.: *Hierarchia języków programowania robotów wraz z opisem wybranych rozkazów*. Archiwum Automatyki i Telemechaniki, Vol.30, No.3-4, 1985. pp.387-405.

[162] Zieliński C.: *TORBOL – opis języka i systemu*. Ver. 1.0, 2.0, 3.0, internal memo, Institute of Automatic Control, Warsaw University of Technology, 1987-1989.

[163] Zieliński C.: *Języki programowania robotów*. **In:** *Podstawowe Problemy Współczesnej Techniki: Robotyka*, Vol.25, Państwowe Wydawnictwo Naukowe, Warszawa1987. pp.129-139.

[164] Zieliński C.: *Klasyfikacja i metody definiowania języków programowania robotów: zastosowanie do sformułowania języka zorientowanego na przemieszczanie obiektów*. Ph.D. dissertation, Warsaw University of Technology, Department of Electronics, Warsaw 1988.

[165] Zieliński C.: *TORBOL – język programowania robotów zorientowany na przemieszczanie obiektów*. Proceedings of the 10th National Conference on Automatic Control, 21-24 June 1988, Lublin, Wydawnictwa Politechniki Lubelskiej, Lublin 1988, Vol.3, pp.61-62.

[166] Zieliński C.: *Środowisko programowe języka programowania robotów zorientowanego na przemieszczanie obiektów*. Proceedings of the 2nd National Conference on Robotics, Wrocław, 21-23 September 1988, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1988. Vol.3, pp.249-255.

[167] Zieliński C.: *Przegląd cech istniejących języków programowania robotów*. Archiwum Automatyki i Telemechaniki, No.3-4, 1989. pp.339-365.

[168] Zieliński C.: *TORBOL – język programowania robotów przeznaczonych do wykonywania zadań transportowo-montażowych*. Archiwum Automatyki i Telemechaniki, No.3-4, 1989. pp.367-379.

[169] Zieliński C., Grodecki A., Kręglewska U., Śluzek A., Zielińska T.: *Koncepcja sterownika robotów przeznaczonego do celów badawczych*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.336-341.

[170] Zieliński C.: *Opis semantyki rozkazów języków programowania robotów*. Archiwum Automatyki i Telemechaniki, No.1-2, 1990. pp.15-45.

[171] Zieliński C.: *Zastosowanie sieci Petriego do opisu działania robota wyposażonego w czujniki*. Proceedings of the 3rd National Conference on Robotics, 19-21 September 1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.330-335.

[172] Zieliński C.: *Incorporation of Sensors in Object-Level Robot Programming Language*. Proceedings of the 8th CISM IFToMM Symposium on Theory and Practice of Robots and

Manipulators, Ro.Man.Sy'90, 2-6 July 1990, Cracow, Poland, Warsaw University of Technology Publications. pp.418-425.

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

[174] Zieliński C.: *Description of Semantics of Robot Programming Languages*. Mechatronics, Vol.2, No.2, Pergamon Press, 1992. pp. 171-198.

[175] Zieliński C.: *Flexible Controller for Robots Equipped with Sensors*. Proceedings of the 9th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'92, 1-4 September 1992, Udine, Italy, Lecture Notes in Control and Information Sciences 187, Springer-Verlag, Berlin 1993. pp.205-214.

[176] Zieliński C.: *Object Level Robot Programming Languages*. **In:** *Robotics Research and Applications*. Eds: A. Morecki, W. Muszyński, K. Tchoń, Warsaw 1992. pp.221-235.

[177] Zieliński C.: *Programming Languages for Flexible Automation*. SERC Grant Report (GR/G64091), Loughborough University of Technology, September 1992.

[178] Zieliński C., Grodecki A., Kręglewska U., Sobczyk J., Śluzek A., Zielińska T.: *Sterownik robotów przeznaczony do celów badawczych*. KBN memo, Institute of Automatic Control, Warsaw University of Technology, December 1992.

[179] Zieliński C., Grodecki A., Kręglewska U., Sobczyk J., Śluzek A., Zielińska T.: *Specyfikacja sterownika robotów przeznaczonego do celów badawczych*. KBN memo, Institute of Automatic Control, Warsaw University of Technology, December 1992.

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

[181] Zieliński C.: *Controller Structure for Robots with Sensors*. Mechatronics, Vol.3, No.5, Pergamon Press, 1993. pp.671-686.

[182] Zieliński C.: *Sterownik robotów przeznaczony do celów badawczych*. Proceedings of the 4th National Conference on Robotics, 22-24 September 1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.73-80.

[183] Zieliński C.: *Sensory Robot Motions*. Archives of Control Sciences, Vol.3 (XXXVIII) no.1-2, 1994, pp.5-20.

[184] Zieliński C., Grodecki A.: *Reaktywne sterowanie robotem wyposażonym w czujnik sił*. Institute of Automatic Control memo no.503/031/003/1, Warsaw University of Technology, January 1994.

[185] Zieliński C., Zielińska T.: *Sensor-Based Reactive Robot Control*. Proceedings of the 10th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'94, 12-15 September 1994, Gdańsk, Poland.

[186] Zieliński C., Zielińska T.: *Software for Mechatronic Devices*. **In:** *Mechatronics: the basis for new Industrial Development*. Eds: M. Acar, J. Makra, E. Penney, Computational Mechanics Publications. Southampton, Boston, 1994. Joint Hungarian-British Mechatronics Conference, 21-23 September 1994, Budapest, Hungary, pp.755-760.

[187] Zieliński C.: *Sterownik robotów wyposażonych w różnorodne czujniki*. **In:** PIAP Bulletin: *Projekty badawcze – granty w dziedzinie robotyki*. Proceedings of a robotics seminar funded by Polish Scientific Research Committee, 7-8 December 1993, PIAP, Warsaw 1994, pp.31-40.

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

[189] Zieliński C.: *Distributed Software for Mechatronic Systems*. International Journal of Intelligent Mechatronic: Design and Production, Vol.1, No.1, 1994. pp.11-24.

[190] *Turbo Pascal: Programmer's Guide*. Ver.6.0, Borland International Incorporated, 1990.

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

[192] *Turbo C++: Programmer's Guide*. Ver.2.0, Borland International Incorporated, 1990.

[193] *User's Guide to VAL*. Unimation Company, 1980.

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

[195] *Webster's Unabridged Dictionary of the English Language*. Portland House, New York, 1989.

Metody programowania robotów

Streszczenie

Celem pracy jest pokazanie problemu programowania robotów z punktu widzenia użytkownika oraz osoby projektującej i wdrażającej system sterujący robotami, a ponadto przedstawienie opracowanych przez autora metod programowania robotów. We wstępie do rozprawy tak ogólnie określony temat umiejscowiono w relacji do całości oprogramowania tworzonego dla robotów. Robotyka jest interdyscyplinarną dziedziną wiedzy, więc zakres tworzonego oprogramowania jest bardzo duży. W związku z tym skoncentrowano się na oprogramowaniu związanym bezpośrednio z funkcjonowaniem systemu robotycznego, a więc na oprogramowaniu:

– sterującym efektorami (jednym lub wieloma ramionami, nogami, narzędziami etc.),
– obsługującym czujniki,
– wnioskującym i specyficznym dla konkretnie realizowanego zadania oraz
– współdziałającym z operatorem systemu.

Następnie przeprowadzono szczegółową analizę literatury dotyczącej języków programowania robotów oraz sterowników robotów przeznaczonych do celów badawczych. Ponadto, we wstępie przedstawiono wkład własny autora do omawianej dziedziny wiedzy.

W rozdziale 2. zdekomponowano system robotyczny na trzy części: efektory, receptory oraz podsystem sterujący. Wprowadzono pojęcie czujnika wirtualnego stanowiącego agregat danych uzyskanych z czujników rzeczywistych (sprzętowych). Zdefiniowano stan wszystkich wyżej wymienionych części systemu. Celem tej formalizacji było określenie semantyki instrukcji języków programowania robotów.

Rozdział 3. przedstawia trzy metody programowania robotów: *on-line*, *off-line* oraz *hybrydową*. Podano zarówno zalety, jak i wady tych metod. Następnie skoncentrowano się na metodzie *off--line*. Jako przykłady przedstawiono języki programowania robotów: WAVE, VAL II, AL, RAPT oraz język zdefiniowany i zaimplementowany przez autora – TORBOL. Języki programowania robotów zostały sklasyfikowane w zależności od abstrakcyjnych pojęć, do których odwołują się instrukcje ruchowe języka; są to: fragmenty łańcucha kinematycznego, końcówka manipulatora, obiekty znajdujące się w środowisku lub zadanie, które ma być wykonane.

Kolejny rozdział przedstawia wpływ wykonania instrukcji na stan części systemu robotycznego zdefiniowanych w rozdziale 2. Celem tej formalizacji jest ułatwienie doboru instrukcji nowo tworzonych języków (przede wszystkim semantyki, gdyż problemy składni nie różnią się od tych spotykanych w informatyce) oraz uproszczenie implementacji języków (poprzez precyzyjne określenie semantyki). Ponadto, rozważono sposoby wykorzystania czujników w systemach robotycznych. Wskazano, że czujniki mogą być używane biernie: do śledzenia (monitorowania) warunku początkowego i końcowego, wykrywania sytuacji awaryjnych oraz aktywnie: do sterowania, tzn. generowania lub modyfikowania zaplanowanej trajektorii ruchu efektora. W zakończeniu rozdziału pokazano sposób użycia wprowadzonego formalizmu do definiowania semantyki instrukcji ruchowych przykładowych języków.

Rozdział 5. zajmuje się sprawami implementacji języków programowania robotów. Wyróżniono trzy metody implementacji:

– jako język specjalizowany,
– jako rozszerzenie istniejącego języka uniwersalnego oraz
– jako biblioteki procedur napisanej w języku uniwersalnym.

Podkreślono wady i zalety każdej z metod oraz wskazano szczególną przydatność ostatniej z wymienionych metod do celów badawczych. Metodę tę autor zastosował do implementacji języków (bibliotek) ROPAS oraz ROOPL. Oba języki zaimplementowane są w Pascalu, z tym że pierwszy

z nich wykorzystuje podejście strukturalne, natomiast drugi – podejście obiektowe do programowania. Ponadto, w rozdziale tym opisano dwa sposoby implementacji języka specjalizowanego, jakim jest TORBOL.

W rozdziale 6. zajęto się rozwinięciem koncepcji bibliotek procedur napisanych w językach uniwersalnych do tworzenia sterowników "przykrojonych" do potrzeb zadania, które ma być zrealizowane przez robota. We wstępie przeanalizowano literaturę dotyczącą specjalizowanych, badawczych sterowników robotów, a następnie opisano strukturę sterownika badawczego RORC. Struktura ta wynika z rozważań teoretycznych zamieszczonych w poprzednich rozdziałach. Przyjęto, że sterownik będzie podzielony na procesy wykonywane współbieżnie. Z rozważań teoretycznych wynikało, że wygodnie jest wyróżnić jeden proces sterujący efektorem oraz tyle procesów, ile jest czujników wirtualnych. Nadto dodano proces komunikacji z operatorem oraz proces zarządzający ekranem monitora. Przedstawiono również sposób konstrukcji sterowników do realizacji określonych zadań. Sterownik badawczy składa się z procedur i procesów bibliotecznych, ewentualnie uzupełnionych o dodatkowe specjalizowane procedury zapisywane w języku C. Zapewnia to łatwość tworzenia takich sterowników oraz dużą otwartość powstałej struktury. Zmiana zadania albo dołączenie dodatkowych czujników lub narzędzi wiąże się jedynie z dodaniem lub modyfikacją niektórych procedur. Całość zaimplementowano na jednym komputerze obsługującym procesy w podziale czasu.

Rozdział 7. przedstawia wykorzystanie RORC do badań nad sterowaniem reaktywnym (reakcyjnym). Metoda sterowania rektywnego (opracowana przez autora) wywodzi się z dość szerokiej i różnorodnej grupy metod zwanych zbiorczo metodami behawioralnymi. Większość z tych metod bazuje na heurystyce i nie posiada podstaw formalnych. W metodzie reaktywnej dzieli się wielowymiarową przestrzeń odczytów czujników wirtualnych na podprzestrzenie. Z każdą z tych podprzestrzeni kojarzy się reakcję, którą robot wykona, gdy odczyty czujników wejdą do tej podprzestrzeni. Wyróżnia się również podprzestrzeń neutralną. Tak długo jak odczyty pozostają w tej podprzestrzeni, wykonywane jest zaplanowane zadanie, czyli reakcja główna. Jeżeli cokolwiek przeszkodzi w wykonaniu planu, odzwierciedlone to będzie we wskazaniach czujników wirtualnych, a to spowoduje wywołanie reakcji korygujących. Rozważono zarówno reakcje zależne, jak i niezależne od stanu efektorów i podsystemu sterującego. Okazało się, że formalny zapis reakcji bardzo łatwo jest przełożyć na program (w języku C), który będzie stanowił sterownik RORC wykonujący postawione zadanie. Metodę zilustrowano na przykładzie znajdowania drogi w prostym labiryncie (zadanie dyskretne). Następnie, metoda została sprawdzona dla labiryntów dowolnych oraz zastosowana dla zadań ciągłych – śledzenie konturów obiektów.

W rozdziale 8. uogólniono koncepcję sterownika badawczego RORC na systemy wielorobotowe. Rozważono zarówno strukturę hierarchiczną, jak i płaską tego typu systemów. Do realizacji wybrano jednak strukturę hierarchiczną. Zaimplementowano zarówno metodę interakcyjną, jak i nieinterakcyjną komunikacji między procesami agregującymi dane z czujników oraz procesami bezpośrednio sterującymi efektorami. Ponadto, przeanalizowano problemy synchronizacji między procesami, stosując sieci Petriego. W odróżnieniu od RORC wykorzystano do realizacji układu sterującego systemem wielorobotowym system operacyjny czasu rzeczywistego rozproszony na wielu komputerach połączonych siecią.

W zakończeniu pracy omówiono dorobek autora. Podkreślono przydatność wprowadzonego formalizmu do definiowania i implementacji nowych języków programowania robotów oraz do realizacji sterowników sytemów, zarówno jednorobotowych, jak i wielorobotowych. Podkreślono, że systemy te mogą być wyposażone w wiele czujników o różnorodnej złożoności i są łatwe do modyfikacji, dzięki swej otwartej strukturze. Całość monografi przedstawia pogląd autora na problemy programowania robotów.

Acknowledgements

The work on the manuscript of this dissertation was supported by the Polish Committee of Scientific Research under Grant no. 3 P403 016 05. This source of funding is gratefully acknowledged.

Moreover, I express my thanks for the comments on the draft version of this monograph that I received from Professor Krzysztof Malinowski and Professor Anatol Gosiewski from the Institute of Control and Computation Engineering, Warsaw University of Technology.

I am also grateful to the Robotics Team working with me at the Institute of Control and Computation Engineering, Warsaw University of Technology, on the implementation of Research-Oriented Robot Controller RORC and its subsequent enhancement that evolved into a multi-robot system MRROC.

I am very grateful to Mr Włodzimierz Macewicz, the computer wizard, who helped me to typeset the monograph by fiddling with the intricacy of LaTeX and TeX.

Last but not least, thanks are due to my British friends: Dr Peter Reid and Dr Helen Reid, who polished the English that had initially been "Polished" by me in the early version of this monograph.

Index

Contents