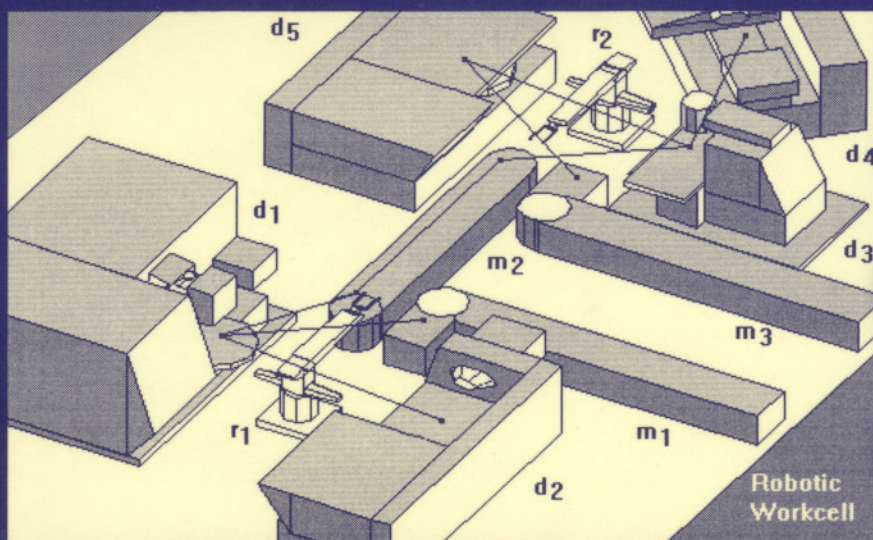


INTELLIGENT ROBOTIC SYSTEMS

Design, Planning, and Control

WITOLD JACAK



IFSR International Series on
Systems Science and Engineering
Volume 14

INTELLIGENT ROBOTIC SYSTEMS

DESIGN, PLANNING, AND CONTROL

International Federation for Systems Research

International Series on Systems Science and Engineering

Series Editor: George J. Klir
State University of New York at Binghamton

Editorial Board

Gerrit Broekstra
*Erasmus University, Rotterdam,
The Netherlands*

John L. Casti
Santa Fe Institute, New Mexico

Brian Gaines
University of Calgary, Canada

Ivan M. Havel
*Charles University, Prague,
Czech Republic*

Manfred Peschel
Academy of Sciences, Berlin, Germany

Franz Pichler
University of Linz, Austria

- | | |
|-----------|---|
| Volume 8 | <i>THE ALTERNATIVE MATHEMATICAL MODEL OF
LINGUISTIC SEMANTICS AND PRAGMATICS</i>
Vilém Novák |
| Volume 9 | <i>CHAOTIC LOGIC: Language, Thought, and Reality from the
Perspective of Complex Systems Science</i>
Ben Goertzel |
| Volume 10 | <i>THE FOUNDATIONS OF FUZZY CONTROL</i>
Harold W. Lewis, III |
| Volume 11 | <i>FROM COMPLEXITY TO CREATIVITY: Explorations in
Evolutionary, Autopoietic, and Cognitive Dynamics</i>
Ben Goertzel |
| Volume 12 | <i>GENERAL SYSTEMS THEORY: A Mathematical Approach</i>
Yi Lin |
| Volume 13 | <i>PRINCIPLES OF QUANTITATIVE LIVING SYSTEMS
SCIENCE</i>
James R. Simms |
| Volume 14 | <i>INTELLIGENT ROBOTIC SYSTEMS: Design, Planning,
and Control</i>
Witold Jacak |

IFSR was established "to stimulate all activities associated with the scientific study of systems and to coordinate such activities at international level." The aim of this series is to stimulate publication of high-quality monographs and textbooks on various topics of systems science and engineering. This series complements the Federation's other publications.

A Continuation Order Plan is available for this series. A continuation order will bring delivery of each new volume immediately upon publication. Volumes are billed only upon actual shipment. For further information please contact the publisher.

Volumes 1-6 were published by Pergamon Press.

INTELLIGENT ROBOTIC SYSTEMS

DESIGN, PLANNING, AND CONTROL

WITOLD JACAK

*Johannes Kepler University
Linz, Austria and
Polytechnic University of Upper Austria
Hagenberg, Austria*

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-46967-7
Print ISBN: 0-306-46062-9

©2002 Kluwer Academic Publishers
New York, Boston, Dordrecht, London, Moscow

Print ©1999 Kluwer Academic / Plenum Publishers
New York

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Kluwer Online at: <http://kluweronline.com>
and Kluwer's eBookstore at: <http://ebooks.kluweronline.com>

Preface

Robotic systems are effective tools for the automation necessary for industrial modernization, improved international competitiveness, and economic integration. Increases in productivity and flexibility and the continuous assurance of high quality are closely related to the level of intelligence and autonomy required of robots and robotic systems.

At the present time, industry is already planning the application of intelligent systems to various production processes. However, these systems are semi-autonomous and need some human supervision. New intelligent, flexible, and robust autonomous systems are key components of the factory of the future, as well as in the service industries, medicine, biology, and mechanical engineering.

A robotic system that recognizes the environment and executes the tasks it is commanded to perform can achieve more dexterous tasks in more complicated environments. Integration of sensory data and the building up of an internal model of the environment, action planning based on this model and learning-based control of action are topics of current interest in this context. System integration is one of the most difficult tasks whereby sensors, vision systems, controllers, machine elements, and software for planning, supervision, and learning are tied together to give a functional entity. Moreover, robot intelligence needs to interact with a dynamic world. Cognition, perception, action, and learning are all essential components of such systems, and their integration into real systems of different levels of complexity should help to clarify the nature of robotic intelligence.

In a complex robotic agent system, knowledge about the surrounding environment determines the structure and methodologies used to control and coordinate the system, which leads to an increase in the intelligence of the individual system components.

Full or partial knowledge of an agent's environment, as in industry, leads to an intelligent robotic workcell. Because of the rather high level of this knowledge, all the planning activities can be performed off-line, and only task execution needs to be done on-line.

A different approach is needed when little or no information about the environment is available. In this situation, a robotic multiagent system that shows no clear

grouping of components is better suited to develop plans and to react to changes in a dynamic environment. All the calculations have to be done on-line. This requires more processing power and faster algorithms than the organized structure, where only the operations in the execution phase have to be computed in real time.

This book only treats the intelligent robotic cell and its components; the fully autonomous robotic multiagent system is not covered here. However, the on-line components, methods, and algorithms of the intelligent robotic cell can be used in multiagent systems as well.

The book deals with the basic research issues associated with each subsystem of an intelligent robotic cell and discusses how tools and methods from different discrete system theory, artificial intelligence, fuzzy set theory, and neural network analysis can address these issues. Each unit of design and synthesis for workcell control needs different mathematical and system engineering tools such as graph searching, optimization, neural computing, fuzzy decision making, simulation of discrete dynamic systems, and event-based system methods.

The material in the book is divided into two parts. The first part gives detailed formal descriptions and solutions of problems in technological process planning and robot motion planning. The methods presented here can be used in the off-line phase of design and synthesis of the intelligent robotic system. The chapters present the methods and algorithms which are used to obtain the executable plan of robot motions and manipulations and device operations based only on the general description of the technological task.

The second part treats real-time events based on multilevel coordination and control of robotic cells using neural network computing. The components of such control systems use discrete-event, neural-network, and fuzzy logic-based coordinators and controllers. Different on-line planning, coordination, and control methods are described depending on the knowledge about the surrounding environment of robotic agent. These methods call on different degrees of autonomy of the robotic agent. Possible solutions to obtain the required intelligent behavior of robotic system are presented.

In writing this book, a formal approach has been adopted. The usage of mathematics is limited to the level required to maintain the clarity of the presentation. The book should contribute to the better understanding, advancement, and development of new applications of intelligent robotic systems.

Acknowledgments

This book would not have been possible without the help of numerous friends, colleagues, and students. On the professional side, I am most grateful to my colleagues at the University of Linz for the level of support they showed through all these years. In particular, I would like to thank Prof. Franz Pichler, Prof. Gerhard Chroust, and Prof. Bruno Buchberger for providing me with an academic home in Austria.

Much of the work included here was taught in lectures at the University of Linz and at the Technical University of Wroclaw, and several improvements can be attributed through feedback from my students there. Other parts of the theory were developed in cooperation with my Ph.D. students and colleagues, in particular with Dr. Ireneusz Sierocki, Dr. Stephan Dreiseitl, Dr. Gerhard Jahn, **Dr. Paweł Rogaliński**, Dr. Robert **Muszyński**, Dr. Ignacy **Dulęba**, and Dr. Tomasz Kubik, who should also be mentioned for providing valuable input on several topics.

Finally let me thank my family for their continuous support during weekends and late nights when this text was written.

This page intentionally left blank

Contents

1. Introduction	1
1.1. The Modern Industrial World: The Intelligent Robotic Workcell	2
1.2. How to Read this Book	7
2. Intelligent Robotic Systems	9
2.1. The Intelligent Robotic Workcell	9
2.2. Hierarchical Control of the Intelligent Robotic Cell	12
2.3. Centralization versus Autonomy of the Robotic Cell Agent	15
2.4. Structure and Behavior of the Intelligent Robotic System	17

I. Off-Line Planning, Programming, and Simulation of Intelligent Robotic Systems

3. Virtual Robotic Cells	23
3.1. Logical Model of the Robotic Cell	24
3.2. Geometrical Model of the Robotic Cell	24
3.3. Basic Methods of Computational Geometry	26
4. Planning of Robotic Cell Actions	33
4.1. Task Specification	33
4.2. Methods for Planning Robotic Cell Actions	38
4.3. Production Routes — Fundamental Plans of Action	43
5. Off-Line Planning of Robot Motion	55
5.1. Collision-Free Path Planning of Robot Manipulator	55
5.2. Time-Trajectory Planner	99
5.3. Planning for Fine Motion and Grasping	126

6.	CAP/CAM Systems for Robotic Cell Design	141
6.1.	Structure of the CAP/CAM System ICARS	141
6.2.	Intelligent Robotic Cell Design with ICARS	143
6.3.	Structure of the HyRob System and Robot Design Process	148

II. Event-Based Real-Time Control of Intelligent Robotic Systems Using Neural Networks and Fuzzy Logic

7.	The Execution Level of Robotic Agent Action	155
7.1.	Event-Based Modeling and Control of Workstation	157
7.2.	Discrete Event-Based Model of Production Store	164
7.3.	Event-Based Model and Control of a Robotic Agent	165
7.4.	Neural and Fuzzy Computation-Based Intelligent Robotic Agents	169
8.	The Coordination Level of a Multiagent Robotic System	211
8.1.	Acceptor: Workcell State Recognizer	211
8.2.	Centralized Robotic System Coordinator	213
8.3.	Distributed Robotic System Coordinator	219
8.4.	Lifelong-Learning-Based Coordinator of Real-World Robotic Systems	221
9.	The Organization Level of a Robotic System	241
9.1.	The Task of the Robotic System Organizer	241
9.2.	Fuzzy Reasoning System at the Organization Level	242
9.3.	The Rule Base and Decision Making	246
10.	Real-Time Monitoring	255
10.1.	Tracing the Active State of Robotic Systems	255
10.2.	Monitoring and Prediagnosis	256
11.	Object-Oriented Discrete-Event Simulator of Intelligent Robotic Cells	261
11.1.	Object-Oriented Specification of Robotic Cell Simulator	262
11.2.	Object Classes of Robotic Cell Simulator	269
11.3.	Object-Oriented Implementation of Fuzzy Organizer	285
	References	295
	Index	303

CHAPTER 1

Introduction

In a complex system using robotic agents, knowledge about the surrounding environment determines the structure and methodologies used to control and coordinate the system, which leads to an increase in the intelligence of the individual system components.

Full or partial knowledge of the agents' environment, as is found in industry, leads to *an intelligent robotic workcell*. Because of the rather high level of this knowledge, all the planning activities can be performed off-line, and only task-execution needs to be done on-line.

A different approach is needed when little or no information about the environment is available. In this situation, a robotic multiagent system that shows no clear grouping of components is better suited to develop plans and to react to changes in a dynamic environment. All the calculations have to be done on-line. This requires more processing power and faster algorithms than the organized structure, where only the operations in the execution phase have to be computed in real time.

The distinction between these two paradigms is shown in Figure 1.1. This book will treat only *the intelligent robotic cell* and its components (shown on the left side of Figure 1.1). Fully autonomous robotic multiagent systems are not covered here. However, the on-line components and algorithms for an intelligent robotic cell can be used in multiagent systems as well.

The knowledge it will have about the environment determines the requirements of robotic agent intelligence. Depending on the uncertainty in the work space of a robotic agent in a workcell (existence of dynamic objects), the agent can be classified as belonging to one of the following three classes:

- *Nonautonomous agents* require a central processing module to perform off-line and on-line calculations for them.
- *Partially autonomous agents* (reactive agents) can react independently to dynamic changes in the environment by calculating new path and trajectory segments on line.
- *Autonomous agents* require the least amount of supervision by a coordinator and that can change or adopt a given plan of action based on experience learned during their whole life cycle.

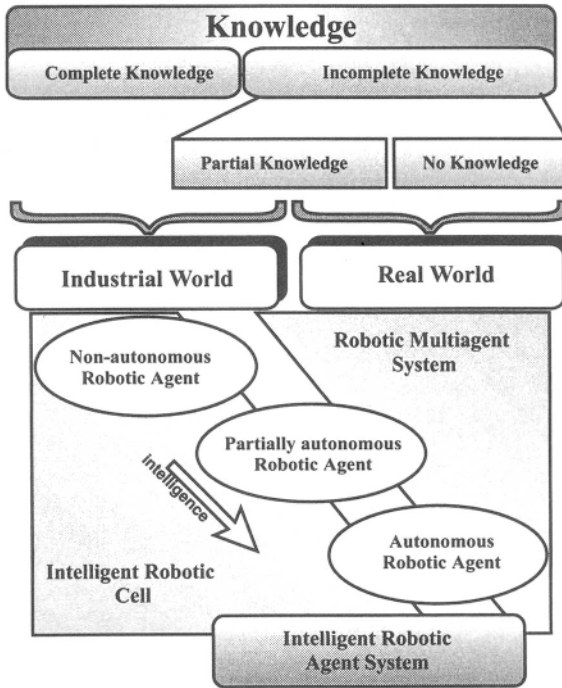


Figure 1.1. Degree of autonomy of a robotic system as a function of the amount of knowledge it has about its environment.

1.1. The Modern Industrial World: The Intelligent Robotic Workcell

Modern manufacturing is characterized by low-volume, high-variety production and close-tolerance, high-quality products. In response to the ever-increasing competition in the global market, major efforts have been devoted to the research and development of various technologies to improve productivity and quality. The economic pressure for increases in quality, productivity, and efficiency of manufacturing processes has motivated the development of more complex and intelligent flexible manufacturing systems (FMS) (Buzacott, 1985; Kusiak, 1990; Lenz, 1989; Meystel, 1988).

The flexible and economic production of goods requires a new level of automation. *Intelligent robotic workcells*, integrating manufacturing stations (workstations) and robots, form the basis of a flexible manufacturing process. Intelligent robotic workcells and computer integrated manufacturing are effective tools to increase manufacturing competitiveness.

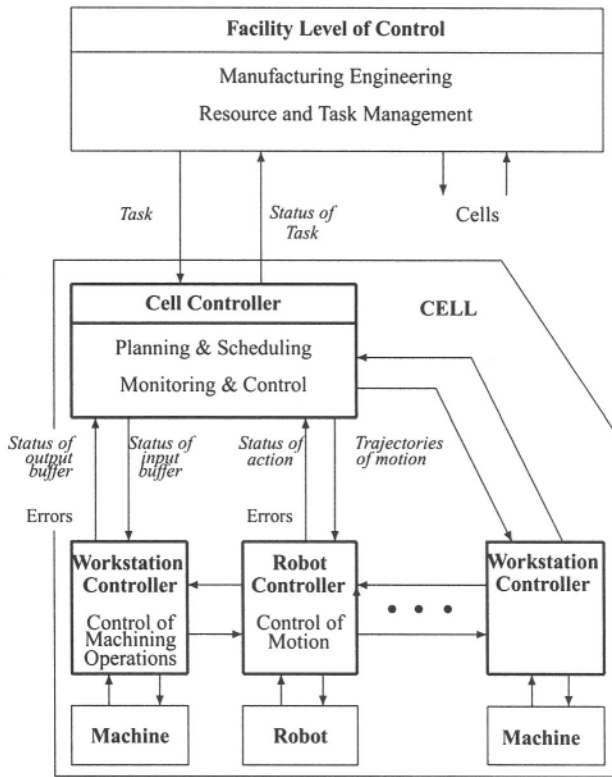


Figure 1.2. General structure of FMS.

In a manufacturing environment, FMS are generally constructed based on a hierarchical architecture (Buzacott, 1985; Jones and McLean, 1986). The FMS hierarchy consists of the following levels: *facility*, *cell*, and *workstation and equipment*. The levels in the hierarchical architecture have the following functions:

- The *facility* level implements the manufacturing engineering, resource, and task management functions.
- The control functions at the *cell* level are job sequencing, scheduling, material handling, supervision, and coordination of the physical activities of workstations and robots.
- Machining operations are performed at the *workstation* level.

The structure of the FMS control system is shown in Figure 1.2. In the above architecture, the control mechanisms are established in such a way that the

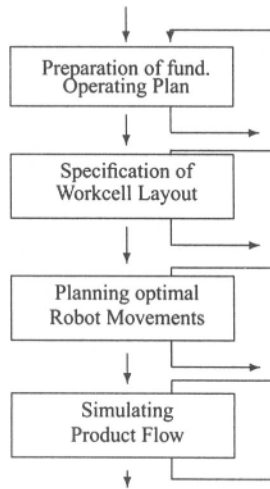


Figure 1.3. Basic definition of the manufacturing process.

upper-level components issue commands to lower-level ones and receive feedback upon the completion of command execution by these lower-level components. The physical components at each level are computer systems and control devices, connected by a communication network such as a local area network (LAN) with a manufacturing automation protocol (MAP) (Buzacott, 1985; Jones and McLean, 1986). Control software is a key component in achieving a high degree of FMS flexibility.

The design of robotic cell control software involves the application and implementation of concepts and methods from different scientific disciplines. For a robotic workcell one has to define the *process* according to which the goods are to be manufactured. This process should be defined, designed, and then loaded into the components of the manufacturing cell and executed.

The synthesis of the *manufacturing process* and its enactment have to be performed off-line and thus executed in a radically different environments, in contrast to software engineering, which has largely the luxury to be able to *design, quality assure, and execute* the programs in roughly the same environment (Chroust, 1992; Pichler, 1989).

With respect to the above hierarchy of manufacturing activities, we list the major subtasks to be performed and provide a process model for it (Saridis, 1983; Black, 1988; Jacak and Rozenblit, in press; Jacak and Rozenblit, 1994). On the highest level of abstraction we have (Figure 1.3):

- **Preparation of the Basic Operating Plan:**

In this step the sequence of processing steps (as defined by the processing

Off-line phase (Part I)
Off line planning and programming

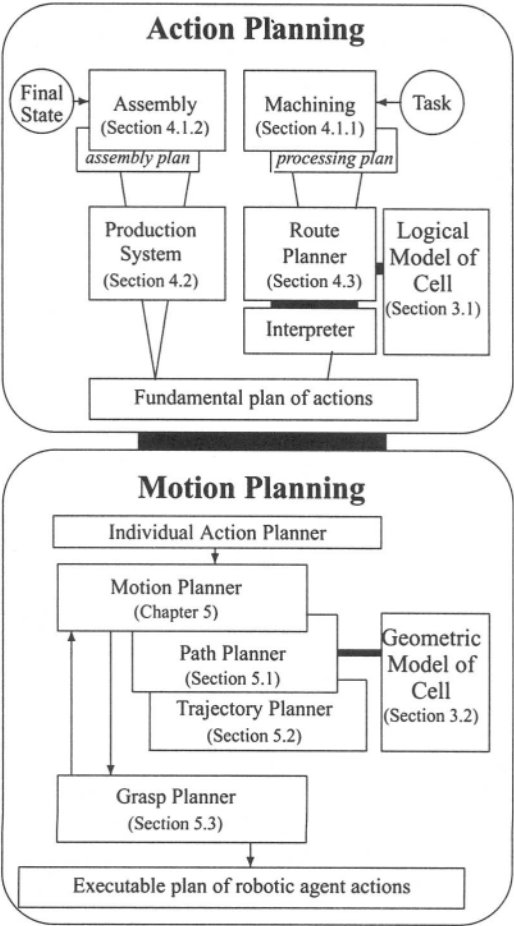


Figure 1.4. Organization of Part I of the book.

On-line phase (Part II)

On-line control and coordination

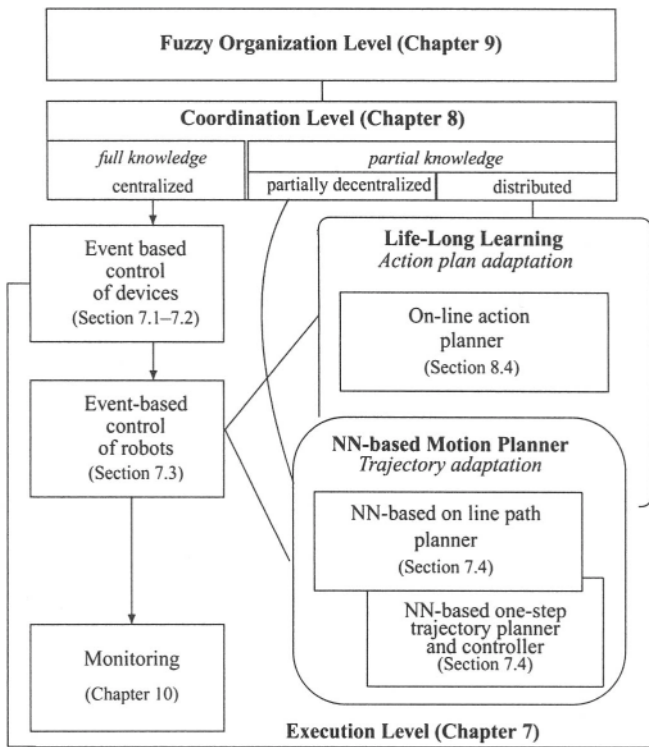


Figure 1.5. Organization of Part II of the book. NN, neural network.

requirements of the product and the applied technology) are defined and the individual processing steps assigned to machines (or machine classes). The subtasks of this process step are: (1) *material selection*, (2) *technological operation selection*, (3) *machine and tool selection*, (4) *machining parameter selection*, and (5) *machining process sequencing* (Black, 1988; Wang and Li, 1991).

- **Modeling of the Processing Workcell:**

It is necessary for there to be an easy way to describe the physical layout of the cell and specify its components, and easy ways to change it and to provide a large repository of standardized models in a library. The result is a so-called *virtual cell*, a complete description of the real cell and its components.

- **Task Planning and Programming of Cell Equipment:**

The automatic programming and task planning is based on logical and geometric models of the cell and robots, mathematical algorithms, and to a certain extent experiments. The generation of a robot action sequence is only one phase in the hierarchy of steps required to plan the robot's behavior in programmable robotic cells. To make the generation of the robot plan applicable to practical problems, more systematic approaches to the design and planning of actions are needed to enhance their performance and enable their cost-effective implementation. At the implementation level, the system for generating the action plan should be capable of reasoning about the geometry and times of actions. Special attention must be focused on questions of directional approach ("what is the best orientation under which a partial product is to be moved toward the machine?"), on collision-freeness, and on optimization of the desired attributes (be it time, energy consumption, speed, etc.) (Prasad, 1989; Bedworth et al., 1991; Maimon, 1987; Lozano-Perez, 1989; Latombe, 1991; Shin and McKay, 1986; Shin and McKay, 1985).

- **Materials Flow — Event-Based Emulation:**

Only for very simple producer/consumer models can the actual behavior of the product flow be computed in a closed analytical form. In practically all interesting cases only simulation can provide a solution (Ranky and Ho, 1985; Wloka, 1991; Rozenblit and Zeigler, 1988; Jacak and Rozenblit, 1993).

The basic manufacturing process specification is shown in Figure 1.3. For most of the presented steps no closed solution or construction method exists, and thus we are forced to verify and validate the results of our engineering efforts heuristically.

1.2. How to Read this Book

In this book we introduce basic research issues associated with each subsystem of the intelligent robotic cell and discuss how different discrete system theory, artificial intelligence, fuzzy set theory, and neural network tools and methods can address these issues. Each block of a workcell control synthesis system need different mathematical and system engineering tools such as graph searching, optimization, neural computing, fuzzy decision making, simulation of the discrete dynamic system, and event based system methods.

The book is organized as follows:

Part I gives detailed descriptions and solutions of problems relating to planning the technological process and robots motions. The methods presented here are used in off-line synthesis of the intelligent robotic cell (Chapters 2–6). Methods and algorithms are given to obtain executable plans of robot motions and manipulations based only on general descriptions of the technological task or on the final state of the assembly process. Examples of software systems are given for the design of intelligent control of robotic systems. The plan of this part of book is shown in Figure 1.4.

Part II treats the real-time, event-based multilevel coordination and control of robotic system (Chapters 7–11). The components of such control systems use discrete event, neural network, and fuzzy-logic based controllers. Different coordination methods are described depending on the state of knowledge about the surrounding environment of the robotic agent. These methods need different degrees of autonomy for the robotic agent. Possible solutions for obtaining the required intelligent behavior of robotic systems are presented.

Chapter 10 describes the synchronized simulation of the manufacturing process performed in a virtual cell parallel to the real technological process, which allows rapid monitoring and diagnosis. The object-oriented specification of an intelligent organizer, coordinator, and executor of cell actions is described in Chapter 11. The plan of this part of the book is shown in Figure 1.5.

CHAPTER 2

Intelligent Robotic Systems

A robotic system and its control are termed intelligent if the system can self-determine its decision choices based upon the simulation of needed solutions or upon experience stored in the form of rules in its knowledge base. The required level of intelligence depends on how the complete its knowledge is about its environment. The different classes of intelligent robotic systems are shown in Figure 2.1. One such system is *the intelligent robotic workcell*. Intelligent robotic cells are effective tools to increase productivity and quality in modern industry.

2.1. The Intelligent Robotic Workcell

In recent years, the use of flexible manufacturing systems has enabled partial or complete automation of machining and assembly of products. The flexible manufacturing system (FMS) is an efficient production system which can be directly integrated with production functions (Prasad, 1989; Bedworth et al., 1991; Black, 1988).

The basic building block of the system is *the robotic manufacturing cell*, called *the robotic workcell*. The parts processed in the system are selected and grouped into families based on the similarity of operations (Prasad, 1989; Bedworth et al., 1991). The machines related to these families are grouped and allocated to the cells. This provides benefits such as reduced setup and flow times and lower in-process inventory levels through simplified work flows. They consist of three main components:

- *a production system* (technological devices)
- *a material handling system* (robots)
- *a hierarchical computer-assisted control system*

Robotic cellular manufacturing systems are data-intensive systems. The robotic workcell integrates all aspects of manufacturing. The intelligent robotic

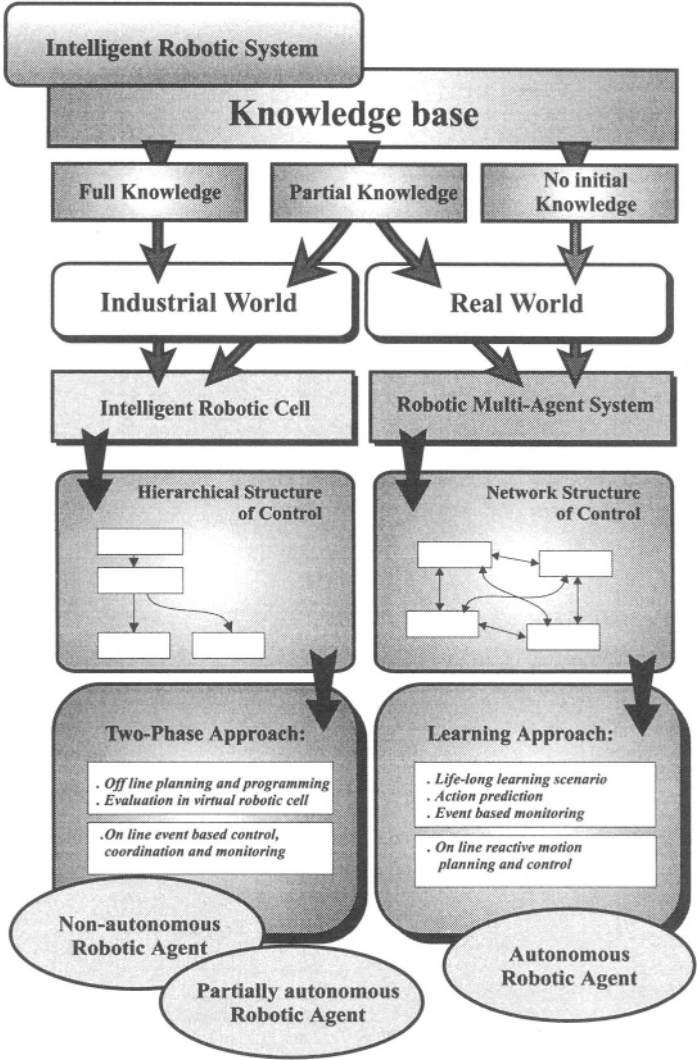


Figure 2.1. Intelligent robotic systems: Classes, structures, and methods.

workcell, and consequently intelligent cellular manufacturing systems, represent the direction of the development of modern manufacturing (Kusiak, 1990; Prasad, 1989; Saridis, 1983; Meystel, 1988).

Definition 2.1.1 (Intelligent Robotic Cell). The robotic cell and its control are termed *intelligent* if it can *self-determine its decisions choices* based upon the simulation of needed solutions in virtual world or upon experience gained in the past both from failures and successful solutions which are stored in the form of rules in the system knowledge base (Kusiak, 1990; Sacerdot, 1981; McDermott, 1982; Saridis, 1989; Yoshikawa and Holden, 1990). An intelligent robotic system in the industrial world is a *computer-integrated cellular system* consisting of partially or fully intelligent robotic workcells.

The planning and control within a cell is done off-line and on-line by a hierarchical controller which itself is regarded as an integral part of the cell. Such a structured robotic manufacturing cell will be called a *computer-assisted robotic cell (CARC)*.

The main purpose of the CARC is to *synthesize* and *execute* a sequence of actions so that the overall system objectives are achieved even under circumstances which may require replanning.

The control system should tie all the data available to the solutions required to run the manufacturing system effectively. Some of the problems to be solved in such an environment are *grouping, machine choice* and *process and motion planning*.

Definition 2.1.2 (Control Task of CARC). The intelligent computer-assisted robotic cell should be able to self-determine for given technological task the control of workcell actions such that:

- the task is realized
 - deadlocks are avoided
 - maximal flow time is minimal
 - work-in-process factor is minimal
 - geometric constraints are satisfied
 - collisions between robotic agents are avoided
-

Design and control of intelligent robotic manufacturing systems involves the application and implementation of concepts, methods, and tools from different disciplines of science, mathematics, and engineering. To synthesize a completely autonomous or semiautonomous computer-assisted robotic cell operating in dynamic environment we use concepts, ideas, and tools from artificial intelligence,

computational intelligence, and general system theory, such as hierarchical decomposition of control problems, the hierarchy of specification models, and discrete and continuous simulation from system theory, and action planning methods, graph-searching of the model's state, neural computation, learning, and fuzzy decision making from artificial and computational intelligence.

2.2. Hierarchical Control of the Intelligent Robotic Cell

The control problem of a computer-assisted robotic cell is a complicated one. Due to the large number of possible solutions (which differ depending on the sequence of technological operations, sequence of sensor-dependent robot actions, geometric forms of manipulator paths, and dynamics of movements along the paths), it is necessary to apply a *stratified* methodology. This is possible since robot actions can be modeled in terms of different conceptual frameworks, namely, operational, geometrical, kinematic, and dynamic.

Thus, to reduce the complexity of the control problem, we propose to apply a hierarchical decomposition process to break down the original problem into a set of subproblems. In this way, the solution of the control synthesis problem is formulated in terms of successive levels of a model of a flexible production system behavior.

The control laws which govern the operation of a CARC are structured hierarchically. We distinguish three basic levels of control:

- the *execution* (workstation) level
- the *coordination* (cell) level
- the *organization* level

This follows the classification of intelligent control systems often cited in the literature (Kusiak, 1990; Lenz, 1989; Saridis, 1983; Meystel, 1988; Maimon, 1987).

The organization level accepts and interprets related feedback from the lower levels, defines the strategy of task sequencing to be executed in real-time and processes large amounts of information with little or no precision. Its functions are defined to be reasoning, decision making, learning feedback, and long-term memory exchange.

The coordination level defines the routing of the part in logical and geometric terms and coordinates the activities of workstations and robots, which in turn coordinate the activities of the equipment in the workstation. It is concerned with the formulation of the actual control task to be executed by the lowest level.

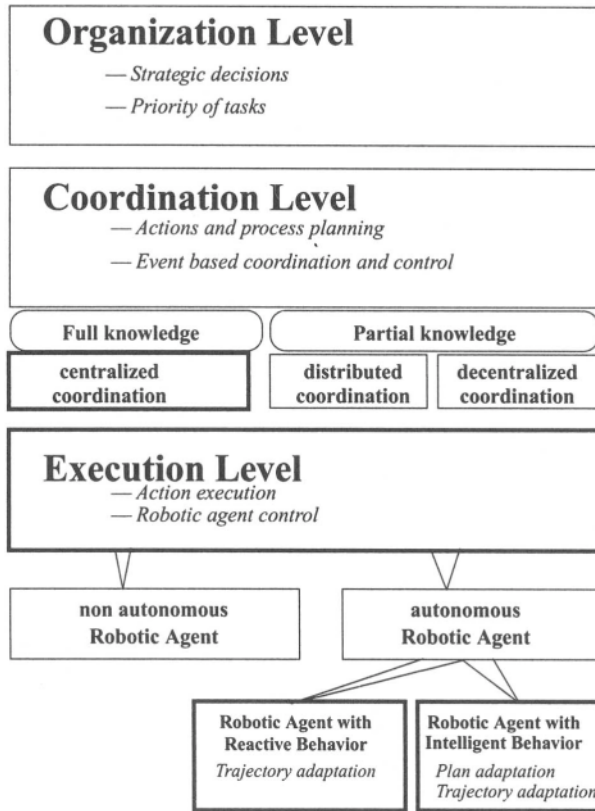


Figure 2.2. Functional structure of an intelligent robotic system.

The execution level is composed of device controllers, and executes the action programs issued by the coordinator.

An intelligent CARC (with the hierarchical structure shown in Figure 2.2) composed of the three interactive levels of organization, coordination, and execution, is modeled with the aid the theory of intelligent systems (Sacerdot, 1981; Saridis, 1989). Figure 2.3 presents the knowledge base and the different classes of formal models which are needed for the planning and control of cell action. All planning and decision making actions are performed within the higher levels. In general, the performance of such systems is improved through self-planning with different planning methods and through self-modification with learning algorithms and schemes interpreted as interactive procedures for the determination of the best possible cell action. There are two major problems in the planning and synthesis of such complex control laws. The first depends on *coordination and integration*

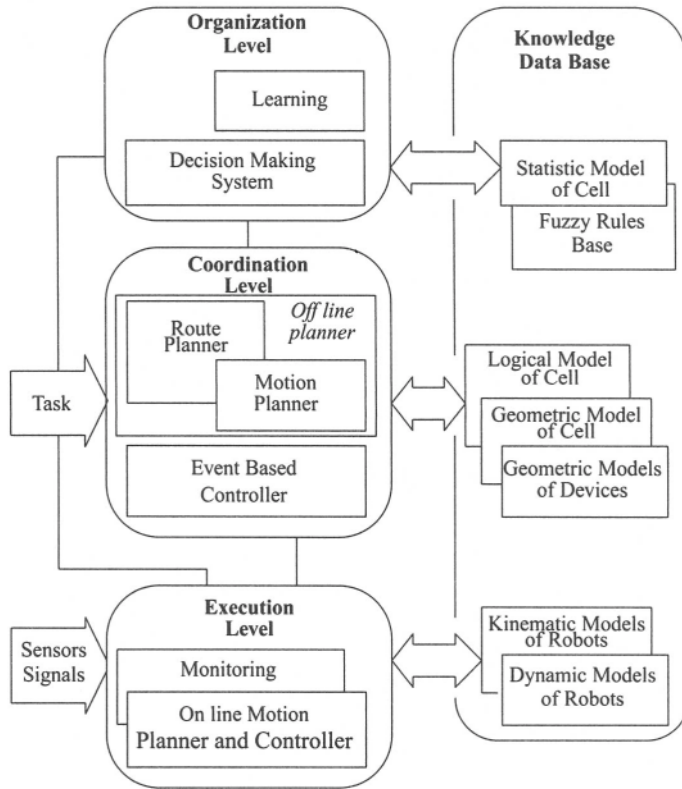


Figure 2.3. Structure of a knowledge base for an intelligent robotic cell.

at all levels in the system, from that of the cell, where a number of machine must cooperate, to that of the whole manufacturing workshop, where all cells must be coordinated. The second problem is that of *automatic action planning and programming* of the elements of the system.

Thus, the control problem of a robotic cell can be considered as having two main elements.

- The first, which we shall call *logical control* or *operational control*, relates to the coordination of *events*, for example, the loading of a part into a machine and the starting of the machine program cycle. Logical control acts to satisfy ordering constraints on event sequences.
- The second, termed *geometric and dynamic control*, relates to the determination of the geometric and dynamic parameters of motions for the elements of the system. Geometric control ensures that the position, path,

and time of movement of all elements of the system and its environment satisfy the geometric and dynamic constraints at all times.

2.3. Centralization versus Autonomy of the Robotic Cell Agent

For a multirobot system forming a robotic cell, there are two extreme possibilities a centralized or a distributed system.

In a centralized robotic system, each robot is only a collection of sensors, actuators and some local feedback loops. Almost all tasks are processed in a coordinator (central controller). The communication between the coordinator and the robots only involves sending data from sensors to the coordinator and receiving detailed commands from the coordinator.

Conversely, in a distributed robotic system, each robot plans and solves a problem (task) “independently” and communicates its information, which is processed in each robot.

2.3.1. Centralized Control of the Intelligent Robotic Cell

A multirobot system which aims at cooperative work always has some tasks which are common to the whole system rather than to individual robots, e.g., a task for planning the manipulation, or a task for global cooperation, etc. These tasks are suited for processing in a coordinator rather than in each robot. However, in a centralized system, defects such as the limitation of processing ability or lack of fault tolerance might become more significant as the system becomes larger, because the processing of all of tasks is performed by the coordinator. Moreover, since the robots are distributed physically, it is more suitable to process some tasks separately rather than concentrating them at the coordinator. Centralized systems, due to the limits on available computational power and the existence of overhead in transferring data between the robots and the central system (coordinator), are not appropriate for other than small groups of robots. Thus, centralized processing is not quite suitable for cooperative work via multiple robots.

2.3.2. Distributed Control of the Intelligent Robotic Cell

The trend in studies of distributed autonomous robotic systems seems to indicate that this approach is superior to centralized system from the view points of flexibility, robustness and fault-tolerance ability. In a pure distributed system, the processing of a cooperative manipulation task which is common to the whole

system is also done separately. In this situation, cooperation among distributed agents becomes necessary to perform the task. This kind of cooperation requires excessive robot intelligence, excessive communication, and tautological processing. These are unnecessary and unnatural for constructing a multirobot system and can be thought of as a type of loss to the whole system (Ahmadabi and Nakano, 1996; Lambrinos and Scheier, 1996; Wang et al., 1996; Kosuge and Oosumi, 1996).

A mixed form, rather than aiming at constructing a pure distributed robotic system, is a multirobot system which maintains a coordinator as its leader, and incorporates homogeneous behavior-based robots which have limited abilities for manipulation. In contrast to the coordinator in a centralized system, the coordinator here acts as a leader and an organizer which coordinates robot behavior, generates goals for the robots, and offers some global position information to each robot to modify its own data. It does not perform any calculation of target object dynamics or force distribution for dynamic cooperation, and does not do any path planning for each robot.

In contrast to collision avoidance among moving robots, in this system, the robots which are working on a manipulation interfere with each other dynamically through the target object. For cooperation, some information about the target object and other robots can be obtained from sensors on each robot. Then, with the information obtained from the coordinator by communication and from sensors, cooperation with other robots for manipulation work can be realized, and the necessity of communication among robots vanishes.

A system without communication among robots has the advantage of avoiding a rapid increase of communication quantity when the number of robots in the system increases. Such a system is not just a simple mixture of the two types of systems, centralized and distributed, in order to obtain an average of the advantages of each system. The coordinator, the leader of the system, not only compensates for the incompleteness of each robot's ability, but also serves to organize robots whose behavioral attributes include limited manipulating ability and cooperative ability. As an illustration, consider that even among human beings, a better quality of cooperation often appears in a group with a leader or a supervisor when a difficult task is being performed.

2.3.3. Cooperation in the Robot Group

Various researchers on multirobot systems have different targets in mind, and thus there are various definitions of cooperation. The most basic and essential point of the cooperation between multiple robots is that they perform a task together without conflict.

For performing different tasks, the information and factors involved in achieving cooperation will be different. This gives different characteristics to the different

cooperation strategies. Cooperation in a group of robots can be defined such that both the information obtained or exchanged by each robot and the elements of the task which the cooperation is working to achieve, include dynamic factors (such as accelerations of forces).

In general, dynamic cooperation is only necessary when performing a task with a dynamic system, whether in a model-based or in a behavior-based approach. Of course, behavior-based cooperation strategy is different from the strategy applied in a model-based cooperating robotic system.

In the *model-based* approach, cooperation in a group of robots is achieved in two steps:

- generating a set of the desired physical parameters (e.g., the desired path and torque of each robot) by using a model-based planner (off-line phase)
- controlling the robot mechanisms to realize the desired parameters.

Therefore, a dynamic cooperation strategy in the model-based approach is required to consider dynamic factors of the system in the planning stage and to guarantee that its controller is able to cope with the dynamic characteristic of the system.

On the contrary, the *behavior-based* approach is based on the idea that robot control can be realized by constructing a robot's behavioral attributes from a certain quantity of behavioral elements. Each behavioral element constructs a behavior control mechanism to act on the world in some situation. Cooperation among robots emerges from robot behavioral attributes and their interaction through the object and the environment. Thus a dynamic cooperation strategy must be such that a robot's behavior acts on the object and on the environment dynamically. Also, each robot's behavioral attributes must be able to cope with the dynamic interaction (Wang et al., 1996; Kosuge and Oosumi, 1996; Haddadi, 1995; Wooldridge et al., 1996).

2.4. Structure and Behavior of the Intelligent Robotic System

The intelligent control of a computer-assisted robotic cell is synthesized and executed in two phases, namely:

- *Planning and off-line simulation*
- *On-line simulation based monitoring and intelligent control*

In the first phase a hierarchical simulation model of a robotic workcell termed a *virtual cell* is created. Because the computer-assisted robotic cell has to make too

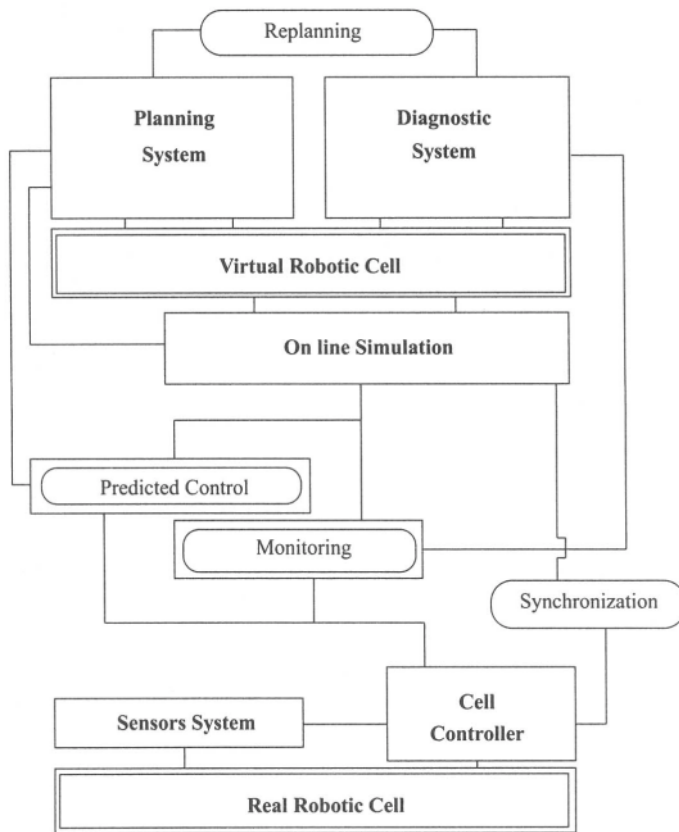


Figure 2.4. Structure of computer-assisted robotic cell (CARC).

many subjective decisions on the basis of deterministic programming methods, the knowledge database and knowledge-based decision support system need to act as an adviser. Such a knowledge database is represented by the virtual workcell. There are two basic types of simulation employed for modeling manufacturing systems: *discrete* and *continuous* simulation. Discrete simulation is event oriented and is based on the concept of a complex discrete events system (DEVS) (Zeigler, 1984; Rozenblit and Zeigler, 1988). Workcell components such as NC-machine tools, robots, conveyors, etc., are modeled as elementary DEVS systems. Discrete changes of state of these systems are of interest. This type of simulation is used for verification and testing different variant of workcell task realizations, called *processes*, obtained from the *Process Planner*. Process planning is based on the description of task operations and their precedence relations. The resulting fundamental planes of cell action describe different ways to decompose the cell

task into ordered sequences of technological operations. To simulate the variants of a process the system know how individual robot actions are carried out. For the detailed modeling of cell components the continuous simulation approach and motion planning methods are used.

The geometrical interpretations of cell actions obtained from the *Motion Planner* and tested in a geometric cell simulator allow us to select the optimal task realizations which establish the logical control unit of the control system. The motion trajectories of robots obtained by continuous simulation create the geometric control unit of the control system.

In the second phase the real-time discrete event simulator of a CARC generated in the first phase is used to generate a sequence of future events of a virtual cell in a given time window. These events are compared with the current states of a real cell and are used to predict motion commands for the robots and to monitor the process flow. The simulation model is modified at any time when the states of the real cell change, and current real states are introduced into the model.

The structure of a computer-assisted robotic cell is shown in Figure 2.4.

This page intentionally left blank

PART I

**Off-Line Planning, Programming, and
Simulation of Intelligent Robotic
Systems**

This page intentionally left blank

CHAPTER 3

Virtual Robotic Cells

Robotic cells are formed by group technology clustering techniques such as the rank order clustering (ROC) method (Black, 1988; King, 1980; Wang and Li, 1991); improved methods also exist (Black, 1988). The problem of grouping parts and machines has been extensively studied. The available approaches can be classified as follows:

- evaluative methods
- clustering algorithms
- similarity coefficient-based methods
- bond energy algorithms
- cost-based heuristics
- within-cell utilization-based heuristics
- neural network-based approaches

A *real cell* is a fixed physical group of machines and a *virtual cell* is a formal representation (computer model) of a machine group in the central computer of a control system. Each cell is designated for the production of a small family of parts. The production related to one type of part is called a *technological task*.

More specifically, an intelligent robotic cell is a set of NC (or CNC)-programmable machines (technological devices) D called *workstations* and production stores M , connected by a flexible material handling facility R (such as robots or automated guided vehicles), and controlled by a computer net (LAN) connected with a sensory system.

Each workstation has its own control and programming system. A workstation (machine) $d \in D$ can have a buffer. Parts are automatically loaded into the machine from the buffer. Then they are machined and subsequently can be stored in the buffer. Depending on the type of technological operation to be carried out on a part, various tooling programs can be used to control the workstation's machining process.

The virtual robotic cell has a hierarchical structure of models. The robotic cell modeling process proceeds in two phases, namely (1) modeling of the logical structure of the cell and (2) modeling of the geometry of the cell.

3.1. Logical Model of the Robotic Cell

In the first phase, the workcell entity structure needs to be designed, i.e., a logical structure of the cell must be created. The group of machines is divided into subgroups, called *machining centers*, which are serviced by separate robots. Parts are transferred between machines by the robots from set R , which service the cell.

A robot $r \in R$ can service only those machines within its service space $Serv_Sp(r) \subset E_0$ (E_0 is the Cartesian base frame). The set of devices which lie in the r th robot service space is denoted by $Group(r) \subset D \cup M$. More specifically, the device belongs to a group serviced by robot r [i.e., $d \in Group(r)$] if all positions of its buffer lie in the service space of robot r .

Consequently, the logical model of a workcell is represented by

$$Cell_{Logic} = (D \cup M, R, \{Group(r) | r \in R\}) \quad (3.1)$$

Example 3.1.1. Consider **Example_Cell**. To perform the technological tasks the following machines are grouped: NC-millers WHD 25 (d_{01}) and FYD 30 (d_{03}), NC-lathes TNS 26e (d_{02}) and Weiler 16 (d_{04}), a quality inspection center (d_{05}), a feeder conveyor (m_{01}), conveyor (m_{02}), and an output conveyor (m_{03}).

The workcell is serviced by two IRb ASEA robots $\{r_{01}, r_{02}\}$. The machining center serviced by robot r_{01} consists of the following devices:

$$Group(r_{01}) = \{d_{01}, d_{02}, m_{01}, m_{02}\}$$

The machining center of robot r_{02} has the following equipment:

$$Group(r_{02}) = \{d_{03}, d_{04}, d_{05}, m_{02}, m_{03}\}$$

The logical structure of the manufacturing workcell is shown in Figure 3.1.

3.2. Geometrical Model of the Robotic Cell

In the second phase of the modeling process the geometry of the virtual cell must be created.

Formally the geometry of the cell is defined as follows (Brady, 1986; Yoshikawa and Holden, 1990; Jacak, in press; Ranky and Ho, 1985; Wloka, 1991):

$$Cell_{Geometry} = (G, H) \quad (3.2)$$

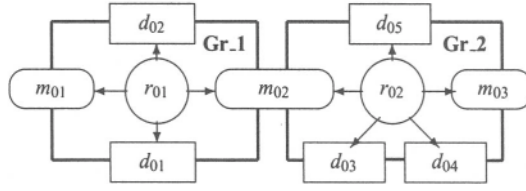


Figure 3.1. Logical structure of a virtual cell.

The first component of the cell geometry description

$$\mathbf{G} = \{\mathcal{G}_d = (E_d, V_d) | d \in D \cup M\} \quad (3.3)$$

represents the set of geometrical models of the cell's objects. E_d is the coordinate frame of the object (device) d , and V_i is the polyhedral approximation of geometry d th object in E_i (Ranky and Ho, 1985; Wloka, 1991).

The second component of the geometrical model

$$\mathbf{H} = \{\mathcal{H}_d : E_d \rightarrow E_0 | d \in D \cup M\} \quad (3.4)$$

represents the cell layout as a set of transformations between an object's coordinate frame E_d and the base coordinate frame E_0 (Brady, 1986).

Consequently, a geometry modeling process has two phases: (1) workcell *object* modeling and (2) workcell *layout* modeling.

3.2.1. Workcell Object Modeling

The geometric model of each object is created through solid modeling (Bedworth et al., 1991; Yoshikawa and Holden, 1990), Which incorporates the design and analysis of virtual objects created from primitives of solids stored in a geometric database. Constructive solid geometry handles primitives of solids, which are bounded intersections of closed half-spaces defined by planes or shapes. More complex objects (such as technological devices, auxiliary devices or static obstacles) can be built by composition using set operations, such as the union, intersection, and difference of solid bodies. As an example of virtual object synthesis, the modeling of the NC-lathe WH64 VF is shown in Figure 3.2.

3.2.2. Workcell Layout Modeling

Workcell objects V_d can be placed in a robot's workscene in any position and orientation. The transformation $\mathcal{H}_d : E_d \rightarrow E_0$ between the object frame E_d and the base frame E_0 is used to calculate the location of virtual objects (devices, stores,

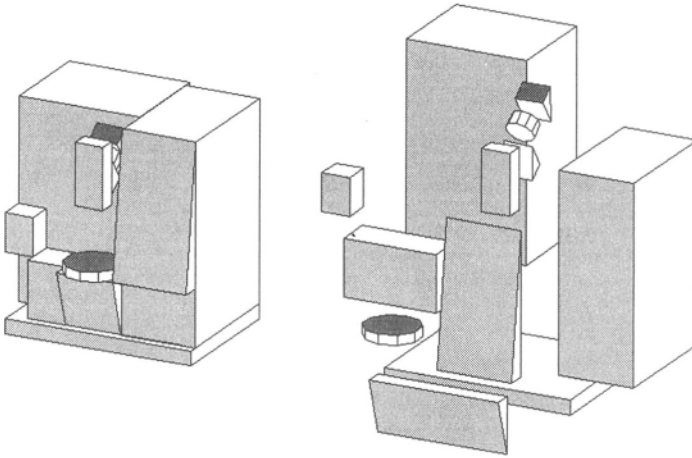


Figure 3.2. The geometric model of the NC-lathe WH64 VF.

robots). The virtual objects (devices, stores) are loaded from a catalogue into the Cartesian base frame and can be located anywhere in the cell using translation and rotation operations in the base coordinate frame (Jacak, in press).

The layout of the manufacturing workcell considered in the example **Example_Cell** is presented in Figure 3.3.

Major drawbacks of such graphics modeling include the following:

- Unless already stored in the catalogue, the graphics images of the robots, devices, and other components of the cell must be designed by the user. This is often time-consuming.
- Using three-dimensional polyhedral approximation of objects, collisions can be detected by complex time-consuming computer geometry algorithms.

3.3. Basic Methods of Computational Geometry

We focus on defining tools for the efficient computation of distances between bodies of objects in three-dimensional space.

3.3.1. Distance Computing Problem

The most natural measure of the proximity is the Euclidean distance between two objects, i.e., the length of the shortest line segment joining the two objects.

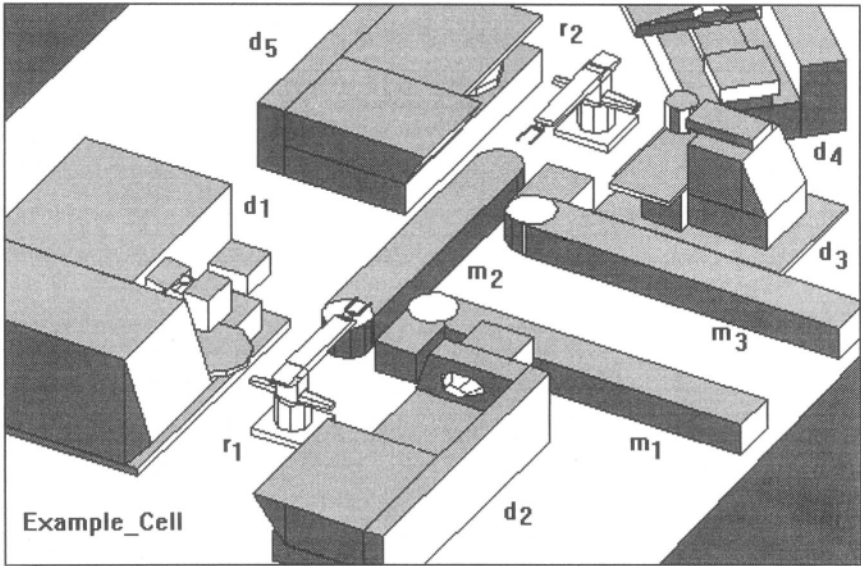


Figure 3.3. The geometric model of the robotic workcell **Example_Cell**.

There is an extensive computational geometry literature concerning the distance calculation problem (Gilbert et al., 1988). Many algorithms have been specifically designed to achieve bounds on the form of the asymptotic time. For two-dimensional problems, Schwartz (1981) gives an $O(\log^2 M)$ algorithm, and more recently, $O(\log M)$ has been used (Schwartz, 1981) (M is the number of vertices). The three-dimensional problem has been considered (Red, 1983) with a time of $O(M \log M)$. Because of their complexity and special emphasis on asymptotic performance, it is not clear that the algorithms are efficient for practical problems. Other schemes have also been described: Red (1983) presents a program which uses a projection approach for polyhedra with facial representation, Gilbert (1988) considers negative distances for polytopes, Mayer (1986) considers boxes and their distances, and Lumelsky (1985) considers line segments.

Let O_1 and O_2 denote two convex solids, x and z two points belonging respectively to O_1 and O_2 , and \mathbf{n} a unit vector. The notation $(\mathbf{n}|x)$ refers to the inner product of vectors \mathbf{n} and x .

The Euclidean distance between O_1 and O_2 , equal to

$$\rho(O_1, O_2) = \min_{x, z} \{ \|x - z\| \mid x \in O_1, z \in O_2 \} \quad (3.5)$$

can be computed by alternatively projecting a point of O_1 onto O_2 , the point of O_2 that we obtain onto O_1 , etc., until the distance between the points converges. For

nonoverlapping objects the Euclidean distance can be defined as

$$\rho(O_1, O_2) = \max_{\|\mathbf{n}\|=1} \rho(\mathbf{n}) \quad (3.6)$$

where

$$\rho(\mathbf{n}) = \min_{x,z} \{(\mathbf{n}[\mathbf{x} - \mathbf{z}]) | \mathbf{x} \in O_1, \mathbf{z} \in O_2\} \quad (3.7)$$

If they do overlap, such a distance becomes negative and measures how far objects interpenetrate.

The interesting point in this definition is that $\rho(\mathbf{n})$ is always lower than $\rho(O_1, O_2)$. Let us define the influence distance τ as the threshold on interactions between objects. Then, if $\rho(\mathbf{n})$ is greater than τ for some value \mathbf{n} , the same stands for the exact distance and we can declare these objects to be noninteracting.

Note that the computation of $\rho(\mathbf{n})$ can be decomposed into

$$\rho(\mathbf{n}) = \min_z \{(\mathbf{n}[\mathbf{o} - \mathbf{z}]) | \mathbf{z} \in O_2\} + \min_x \{(-\mathbf{n}[\mathbf{o} - \mathbf{x}]) | \mathbf{x} \in O_1\} \quad (3.8)$$

for an arbitrary point \mathbf{o} . Based on such a definition of $\rho(\mathbf{n})$ we can use the following procedure for distance estimation:

1. Select an arbitrary point x in O_1 and project it on O_2 .
2. With $\mathbf{n} = [\mathbf{x} - \mathbf{z}] / \|\mathbf{x} - \mathbf{z}\|$ compute the first point x' of O_1 in direction $-\mathbf{n}$.
3. Then $\rho(\mathbf{n}) = (\mathbf{n}[\mathbf{x}' - \mathbf{z}])$ and we have $\rho(\mathbf{n}) \leq \rho(O_1, O_2) \leq \|\mathbf{x} - \mathbf{z}\|$.

It can be shown that $\rho(\mathbf{n})$ converges toward ρ if the procedure is repeatedly applied (Faverjon, 1986).

It is also possible to convert the distance problem into a quadratic programming problem and apply a feedback neural network to solve it (Lee and Bekey, 1991; Jacak, 1994b). Obstacles are modeled as unions of convex polyhedra in 3D Cartesian space. A polyhedral obstacle can be represented by a set of linear inequalities $\mathbf{a}_j^T \mathbf{x} + b_j < 0$ where $\mathbf{x} \in E_0$. The polyhedral object is modeled by a connectionist network with three units in the bottom layer which represent the x, y, z coordinates of the point. Each unit in the second layer corresponds to one inequality constraint of the object: The connections between the bottom and the second layer have their weights equal to the coefficients a_j, b_j of the corresponding face. Then the j th face of the polyhedral object is represented by a neuron with a sigmoid function

$$\text{Face}_j : \varphi(x) = \frac{1}{1 + e^{-\alpha(\mathbf{a}_j^T \mathbf{x} + b_j)}} = \varphi \left(\sum_1^3 w_i x_i + w_0 \right) \quad (3.9)$$

When a point is given to the bottom-layer units, each of the second-layer neurons decides whether the given point satisfies its constraints. To reduce the training time we can apply hybrid techniques which automatically create the full network topology and values of neural weights based on symbolic computation of the polyhedral object's faces.

Let M, N denote the number of neurons representing the objects O_1 and O_2 , respectively. The distance between objects O_1 and O_2 , i.e., $\rho(O_1, O_2)$, is the solution of the following optimization problem:

$$\rho(O_1, O_2) = \min \left(v(x, z) = \frac{1}{2} [x - z]^T [x - z] \right) \quad (3.10)$$

with constraints

$$\begin{aligned} O_1 : \quad & a_i^T x + b_i < 0 \text{ for } i = 1, \dots, M \\ O_2 : \quad & a_j^T z + b_j < 0 \text{ for } j = 1, \dots, N \end{aligned}$$

The above problem can be transformed into a problem without constraints by introducing the penalty function $r(x, z)$ defined as

$$r(x, z) = \sum_1^M \frac{\varphi_i(x)^2}{1 - \varphi_i(x)} + \sum_1^N \frac{\varphi_j(z)^2}{1 - \varphi_j(z)} \quad (3.11)$$

where φ is the neuron activation function given by Equation (3.9). Then the modified criterion function is

$$w(x, z) = v(x, z) + r(x, z) \quad (3.12)$$

The solution of the above problem can be obtained by attaching a feedback around a feedforward network to form a recurrent loop. This coupled neural network is the neural implementation of the gradient method of distance calculation (Lee and Bekey, 1991; Park and Lee, 1990; Han and Sayeh, 1989; Jacak, 1994b).

Let $d = (x_x, x_y, x_z, z_x, z_y, z_z)^T$; then

$$d(k+1) = d(k) - c(d(k)) \cdot \text{grad } w(d(k)) \quad (3.13)$$

where

$$\begin{aligned} \text{grad } w(d) = & \left[\left([x - z] + \alpha \sum_1^M \varphi_i(x)^2 \frac{2 - \varphi_i(x)}{1 - \varphi_i(x)} \cdot a_i \right)^T \right. \\ & \left. \left(-[x - z] + \alpha \sum_1^N \varphi_j(z)^2 \frac{2 - \varphi_j(z)}{1 - \varphi_j(z)} \cdot a_j \right)^T \right]^T \end{aligned} \quad (3.14)$$

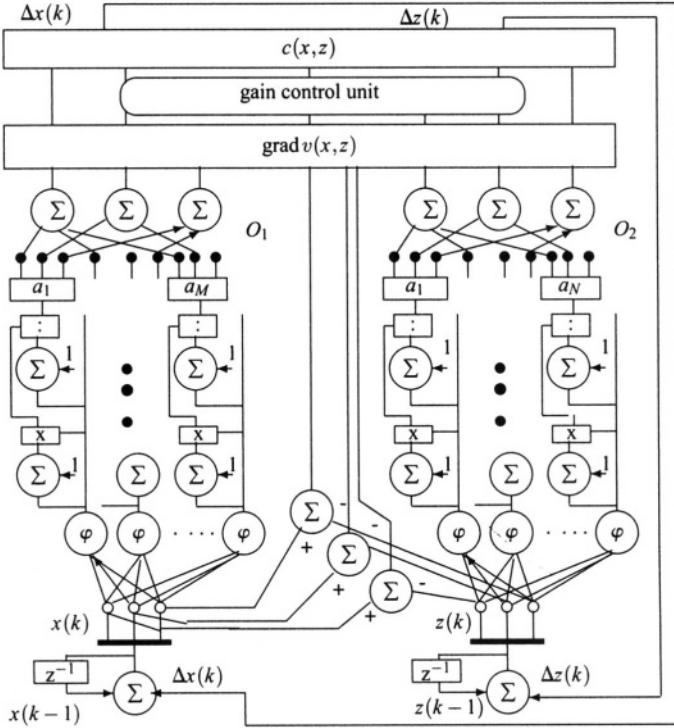


Figure 3.4. Neural computation of the distance between objects O_1 and O_2 .

and $c(d)$ is called the update ratio and can be chosen as

$$c(d) = \frac{\text{grad } w^T \text{grad } w}{\text{grad } w^T H(d) \text{grad } w} \quad (3.15)$$

where $H(d) = [\partial^2 w / \partial d^2]$ is the Hessian of w .

It should be noted that the Hessian of w is easy to compute based on the output of the neurons ϕ . The structure of a neural network realizing the distance calculation is shown in Figure 3.4. The above method can be easily extended for nonpolyhedral objects (Jacak, 1994b).

3.3.2. Intersection Detection Problem

In many cases of collision detection it is possible to avoid calculating the minimal distance between two objects by testing the intersection only. The most important problem is the detection of an intersection between a line segment and a convex solid.

To obtain fast and fully computerized methods for intersection detection we can use the additional geometrical representation of each cell's object. Each volume representation decomposes objects into primitive volumes such as cylinders, boxes, or cubes. We introduce the ellipsoidal representation of 3D objects, which uses ellipsoids for filling the volume. Instead of planes that are in polyhedral approximation, the surface of the object is modeled by parts of ellipsoids. The accuracy of representation depends on the number of ellipsoids and their distribution in the object. The algorithm for packing primitive volumes with touching ellipsoids or spheres has the following major stages: (1) distance calculation, (2) search for the local centers of the ellipsoids, and (3) selection of ellipsoids.

The steps are repeated performed until sufficient accuracy is reached. The ellipsoid packing algorithm transforms each primitive solid into a union of ellipsoids, and consequently the geometry of the virtual object is represented by

$$V_d = \bigcup_i \mathcal{E}_i^d \quad \text{where} \quad \mathcal{E}_i^d : (D_i^T(x - r_i))^T A_i (D_i^T(x - r_i)) - 1 \leq 0 \quad (3.16)$$

and r_i represents the centers of the ellipsoids, D_i is the matrix of the major axes, and A_i is the matrix of the axis lengths.

As an example, a parallelepiped is packed by a bigger central ellipsoid and eight smaller spheres in the corners. The packed ellipsoids are a hierarchical representation in the sense that for gross representation only the biggest ellipsoids are used.

3.3.2.1. Intersection of an ellipsoid with a line segment. The ellipsoidal representation of a virtual object can be used to test the feasibility of a robot configuration, represented by the robot's skeleton. Checking for the collision-free nature of a robot configuration can be reduced to the "broken line–ellipsoid" intersection detection problem, which in this case has the following analytical solution.

Let a point from the given line segment $[P, Q]$ be given by

$$p = P + s[Q - P] \quad \text{where} \quad s \in [0, 1] \quad (3.17)$$

Plugging the point p into the equation of the ellipsoid [Equation (3.16)] gives the following condition of nonintersection between the line segment and ellipsoid

$$as^2 + bs + c \geq 0 \quad \text{for every} \quad s \in [0, 1] \quad (3.18)$$

where

$$a = [D^T(Q - P)]^T A [D^T(Q - P)], \quad b = 2[D^T(Q - P)]^T A [D^T(P - r)] \quad (3.19)$$

$$c = [D^T P]^T A D^T P - 2[D^T P]^T A D^T r + [D^T r]^T A D^T r - 1 \quad (3.20)$$

It is easy to show that:

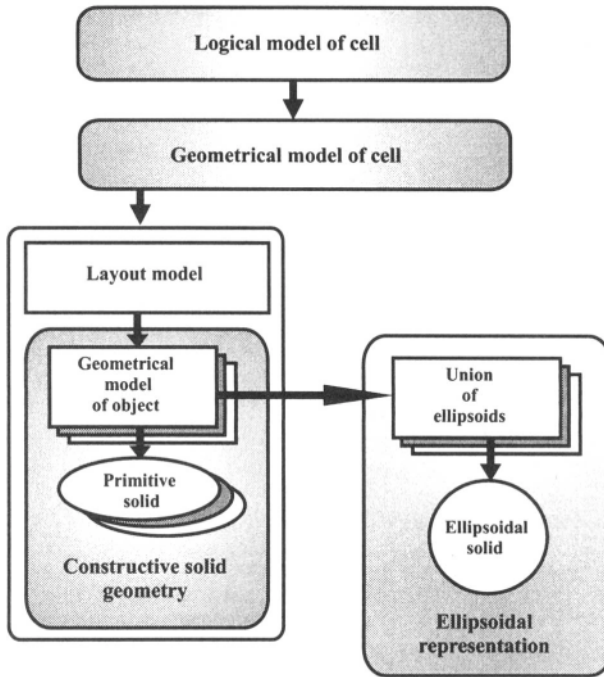


Figure 3.5. Hierarchy of virtual cell models.

Fact 3.3.1. *The line segment $[P, Q]$ does not intersect with the ellipsoid if*

$$c \geq 0, \quad a + b + c \geq 0, \quad \text{and} \quad \sqrt{c} + \sqrt{a + b + c} - \sqrt{a} \geq 0$$

□

□

The virtual robotic workcell is represented by a multilayer system of models which can be used as a knowledge base for planning systems. The hierarchy of workcell models is shown in Figure 3.5.

CHAPTER 4

Planning of Robotic Cell Actions

Small-batch production requires complex control systems with reasonably high flexibility; not only with regard to manufacturing equipment, but also in connection with planning, scheduling, handling, and management decision-making procedures.

The computer integrated manufacturing (CIM) system integrates all aspects of manufacturing. The key to a successful CIM implementation is “*integration*”. Every component in a manufacturing system has to be an integral part of the system. The various components of a system such as CAD (computer-aided design), CAM (computer-aided manufacturing), FMS (flexible manufacturing system), CAT (computer-aided testing), and so on must be integrated (Prasad, 1989; Bedworth et al., 1991; Yoshikawa and Holden, 1990; Black, 1988; Wang and Li, 1991). The communication between CAD and CAM is a key link which to a great extent determines the success of CIM. Computer-aided process planning (CAPP) serves as the bridge between CAD and CAM. CAPP determines how a design will be made in a manufacturing system.

The manufacturing process includes not only processes acting directly upon the manufactured objects, but also preparatory processes (such as production planning, process planning, production scheduling, etc.) and auxiliary processes (such as equipment maintenance, workpiece handling, quality inspection, etc.).

4.1. Task Specification

To begin the synthesis of a computer-assisted robotic cell (CARC) control, one specifies the family of technological tasks to be realized in the cell.

Machining Task: All activities that consist of material-removal operations constitute machining. Examples are turning, drilling, and face milling.

Assembly Task: Assembly involves assembling the manufactured parts to form the required products. In order to perform this task, the parts have to be machined according to the required specifications and tolerances.

Depending on how the parts arrive at the machines, there are two modes of processing (machining or assembly), flow shop and job shop. In the flow shop, all parts flow in one direction, whereas in the job shop, the parts may flow in different directions; also, a machine can be visited more than once by the same part. In both cases, a part does not have to visit all machines.

4.1.1. Machining Task Specification

In this section we focus on issues involved in the machining process. The *machining process* of part may include various machining operations, such as turning, milling, drilling, grinding, broaching, gear-cutting, etc., depending on the required shape, dimensions, tolerance, and surface quality of the part.

The basic components of the machining process are *operations*. An operation is a complete portion of the machining process for cutting, grinding, or otherwise treating a workpiece at a single workplace. The operation is characterized by unchanged equipment, unchanged workpiece, and continuity. Each operation is to be processed by a most one machine at a time.

Process planning for CARCs is critically dependent on the task representation. Several task representation schemes have been proposed (Sacerdot, 1981; Lifschitz, 1987; Homem De Mello and Sanderson, 1990; Sanderson et al., 1990; Maimon, 1987), and include lists of operations, triangle tables, and AND/OR graphs.

Here, we use a general description of a workcell machining task, formally specified as follows.

Definition 4.1.1 (The Machining Task). The machining task realized by a robotic cell is represented by a three-tuple:

$$Task = (O, \prec, \alpha) \quad (4.1)$$

where O is a finite set of *technological operations* (machine, test, etc.) required to process the parts, $\prec \subset O \times O$ is the partial order *precedence relation* on the set O , and $\alpha \subset O \times (D \cup M)$ is the relation of device or store assignment.

The partial order represents an operational precedence, i.e., $q \prec o$ means that the operation q is to be completed before the operation o can begin. $(o, d) \in \alpha$ means that the operation o can be performed on the workstation d , and if $(o, m) \in \alpha$, then m is the production store from the set M where the parts can be stored after the operation o has been completed. The parts are transferred between devices by the robots.

Based on the previously defined logical model of the cell and the description of the set $Group(r)$, we can define the relation β which describes the transfer of

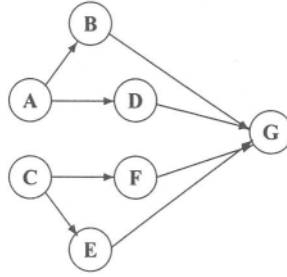


Figure 4.1. The machining task.

parts after each technological operation of the task

$$\beta \subset (O \times O) \times R \quad (4.2)$$

where

$$((o_i, o_j) \beta r) \Leftrightarrow (\{d_i, d_j\} \subset \text{Group}(r) \vee \{m_i, m_j\} \subset \text{Group}(r))$$

and

$$\alpha(o_i) = \{d_i, m_i\} \quad \text{and} \quad \alpha(o_j) = \{d_j, m_j\}$$

We assume that for each operation o_i there exists a robot r_i which can transfer parts between the workstation d_i and stores m_i from the set $\alpha(o_i)$, i.e., $\{d_i, m_i\} \subset \text{Group}(r_i)$. In a special case β is a partial function, i.e., $\beta : O \times O \rightarrow R$.

Example 4.1.1. In the example **Example_Cell**, the manufacturing cell should perform a task consisting of the following operations:

- $\{A, D, E\}$ mill operations
- $\{B, C, F\}$ turn operations
- $\{G\}$ quality test operation

The precedence relation \prec is shown in Figure 4.1. To perform this task the following machines are grouped: the NC-millers WHD 25 (d_{01}) and FYD 30 (d_{03}), the NC-lathes TNS 26e (d_{02}) and Weiler 16 (d_{04}), and quality inspection center (d_{05}), a feeder conveyor (m_{01}), a conveyor (m_{02}) and an output conveyor (m_{03}).

The symbols in brackets represent the logical names of the cell devices. The allocation relation α is illustrated in Figure 4.2.

Such a task is realized in *robotic cells* based on the group technology concept, which groups parts together based on their similarities. Such a cellular manufacturing approach organizes machines and robots into groups responsible for producing a family of parts.

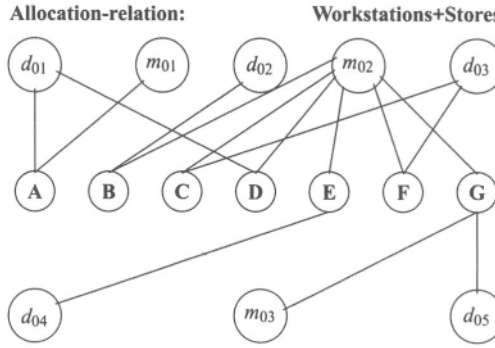


Figure 4.2. The allocation relation.

4.1.2. Assembly Task Specification

The second type of technological task is the assembly task. All operations which lead from parts to the required final product constitute assembly.

Definition 4.1.2 (The Assembly Task). The assembly task is the partial ordered set of the assembly operations over the parts:

$$Assembly = (A, \prec, \gamma, \alpha) \quad (4.3)$$

where A is a finite set of *assembly operations* required to process the final product, $\prec \subset A \times A$ is the partial order *precedence relation* on the set A , $\gamma \subset A \times MTasks$ is the relation of machining task assignment (where $MTasks$ is the set of machining tasks), and $\alpha \subset A \times D$ is the relation of assembly station assignment.

The precedence relation among operations for the final product can be represented by a complex directed graph (digraph). In this digraph any node of degree 1, i.e., with the number of edges incident to the node equal to 1, denotes *the initial operation* a_0^i . The initial operation a_0^i is directly connected with the appropriate machining task $Task^i$ which prepared the initial part,

$$\gamma(a_0^i) = Task^i$$

Any node of degree greater than 1 denotes a subassembly operation and the root node denotes a final assembly operation. The arcs of the digraph correspond to precedence constraints. The partial order represents an operational precedence, i.e., $a \prec b$ means that the assembly operation a is to be completed before the assembly operation b can begin. Based on this description we can use the previously defined digraph as a representation of the final product. The nodes of the graph

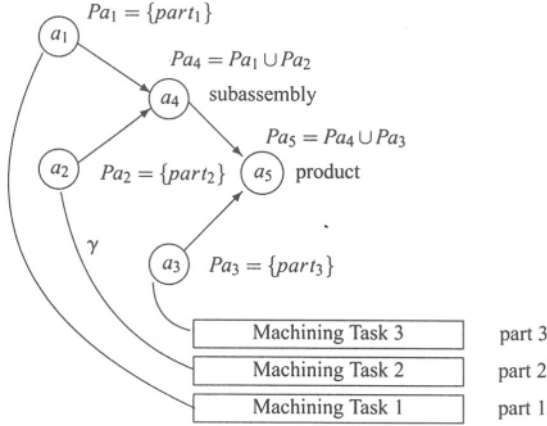


Figure 4.3. The assembly task.

represent the parts or subassemblies. Each initial node denotes a part (and corresponds to a machining task that prepared this part) and the rest of the nodes denote a subassembly or a final product. Each part is connected to one subassembly.

The mapping between initial operations (initial nodes, parts) and machining tasks is described by the relation γ . $(a_0^i, Task^j) \in \gamma$ means that the part (initial operation) a_0^i can be obtained as a result of the machining task $Task^j$. The graph representation allows assembly at a particular node of only one subassembly with any number of parts or any number of subassemblies without parts at all. Then each subassembly node can be represented by the set of initial parts, which are the initial nodes in the subassembly tree expanded from the subassembly node.

Given two subassemblies a_i and a_j characterized by their sets of parts Pa_i , respectively, and Pa_j , we say that joining a_i and a_j is an assembly operation if the set $Pa_k = Pa_i \cup Pa_j$ characterizes a new subassembly a_k . The subassemblies a_i and a_j are input subassemblies of the assembly operation, and a_k (with $Pa_k = Pa_i \cup Pa_j$) is the output subassembly of the assembly operation. Due to the assumption of unique geometry, an assembly operation can be characterized by its input subassemblies only, and it can be represented by a set of two subsets of parts.

The set of connections in the representation of an assembly operation corresponds to a cut-set of the graph of connections of the operation's output subassembly. Conversely, each cut-set of a subassembly's graph of connections corresponds to an assembly operation (Homem De Mello and Lee, 1991; Homem De Mello and Sanderson, 1990). Given the set of all cut-sets of a subassembly's graph of connections, the set of their corresponding assembly operations is referred to as the operations of the subassembly.

An example of an assembly task is shown in Figure 4.3.

4.2. Methods for Planning Robotic Cell Actions

Process planning for a mechanical product involves preparation of a plan that outlines the routes, operations, machine tools, and fixtures required to produce the part. In the last decade there has been the trend to automate process planning, since it increases production efficiency. Examples of such systems are CAPP, CPPP, MIPLAN, and GenPLAN.

There are two basic approaches to process planning: (1) variant approaches and (2) generative approaches (Kusiak, 1990). In the variant approach each part is classified based on a number of attributes and coded using a classification and coding system. The code determines the process plan. The variant approach can be useful in a case where there is a great deal of similarity between parts (Kusiak, 1990).

4.2.1. Assembly Task Planning

Assembling a mechanical structure is typically performed by a series of operations, such as insertions of a peg into a hole. Under this model, the first stage in the design of an assembly system must be to identify the operations necessary to manufacture the given assembly and to specify the sequence in which they are to be performed. The selection of such a sequence of operations is called the assembly planning process.

Definition 4.2.1 (The Assembly Process). Given an assembly task *Assembly* (Definition 4.3) that has I parts, an ordered set of $I - 1$ assembly operations

$$Assembly_Process = (a_1, a_2, \dots, a_{I-1})$$

is an *assembly sequence* or an *assembly process* if:

- no two operations have a common input subassembly
- the output subassembly of the last operation is the whole assembly
- the input subassembly to any operation a_i is either a one-part subassembly or the output subassembly of an operation that precedes a_i

To any assembly sequence $Assembly_Sequence = (a_1, a_2, \dots, a_{I-1})$ there corresponds an ordered sequence

$$Assembly_Trace = (as_1, as_2, \dots, as_I)$$

of I assembly states of the assembly process.

The state as_1 is the state in which all parts are separated. The state as_l is the state in which all parts are joined forming the whole assembly. Any two consecutive states as_i and as_{i+1} are such that only the two input subassemblies of the operation a_i are in as_i and not in as_{i+1} , and only the output subassembly of operation a_i is in as_{i+1} and not in as_i . Therefore, an assembly sequence can also be characterized by an ordered sequence of states. The assembly sequence states the assembly process.

A computer system for assembly planning must have a way to represent the assembly plans it generates. Several methodologies for representing assembly plans have been utilized. These include representations based on directed graphs, on AND/OR graphs, on establishment conditions, and on precedence relationships (Homem De Mello and Lee, 1991; Homem De Mello and Sanderson, 1990). An AND/OR graph can also be used to represent the set of all assembly sequences. The nodes in this graph are the subsets of parts that characterize stable subassemblies. The hyperarcs correspond to the geometrically and mechanically feasible assembly operations. Each hyperarc is an ordered pair in which the first element is a node that corresponds to a stable subassembly a_k and the second element is a set of two nodes $\{a_i, a_j\}$ such that $Pa_k = Pa_i \cup Pa_j$ and the assembly operation characterized by a_i and a_j is feasible. Each hyperarc is associated to a decomposition of the subassembly that corresponds to its first element and can also be characterized by this subassembly and the subset of all its connections that are not in the graphs of connections of the subassemblies in the hyperarc's second element. This subset of connections associated to a hyperarc corresponds to a cut-set in the graph of connections of the subassembly in the first elements of the hyperarc. The formal definition of an assembly AND/OR graph can be found in (Homem De Mello and Lee, 1991; Kusiak, 1990).

Every feasible assembly sequence in the directed graph of an assembly task (Definition 4.3) corresponds to a feasible assembly tree in the AND/OR graph of assembly sequences, and every feasible assembly tree in the graph AND/OR corresponds to one or more feasible assembly sequences in the directed graph of a feasible assembly task.

Assembly planning is a computationally intensive task. The problem of generating the *assembly* sequences for a product can be transformed into the problem of generating the *disassembly* sequences for the same product. Since assembly tasks are not necessarily reversible, the equivalence of the two problems will hold only if each operation used in disassembly is the reverse of a feasible assembly operation, regardless of whether this reverse operation itself is feasible or not.

In the disassembly problem each operation splits one subassembly into smaller subassemblies, maintaining all contacts between the parts in either of the smaller subassemblies.

This transformation leads to a decomposition approach in which the problem of disassembling one assembly is decomposed into distinct subproblems, each being to disassemble one subassembly. It is assumed that exactly two parts or

subassemblies are joined at each time, and that whenever parts are joined forming a subassembly all contacts between the parts in that subassembly are established.

The decomposition algorithm returns the AND/OR graph representation of the assembly process. The correctness of the algorithm is based on the assumption that it is always possible to decide correctly whether or not two subassemblies can be joined, based on geometrical and physical criteria.

Based on this approach, different computer aided assembly planning systems have been implemented, including LEGA, GRASP, XAP, BRAEN, COPLANNER, and DFA. More exact descriptions are given in (Homem De Mello and Lee, 1991).

In this chapter we focus on the machining planning problem in which only the general description of a task is provided (Definition 4.1.1).

4.2.2. Machining Task Planning

Machining process planning can be divided into two stages (Wang and Li, 1991):

- **machining operations design:**
i.e., selection of machining methods and machine tools for each operation, and specification of each operation
- **process route planning:**
i.e., determination of the sequence of machining operations

Process route planning is the overall planning of a part manufacturing process. The objective is to determine the sequence of operations in a process plan. The precedence of operations is based on *operation* constraints, *tooling* constraints and *part geometry* constraints. Several alternative process plans are evaluated and compared so that the best plan can be selected. The evaluation of the alternatives should be a synthetic analysis from the technological and manufacturing efficiency points of view.

Then the main purpose of an intelligent control of autonomous robotic workcell (CARC) for the machining process is to synthesize and execute a sequence of machine and robot actions so that the all operations are realized. Such systems should enable *automatic* generation and interpretation of robot and NC machine actions. There are two major problems in designing such complex control systems (Sacerdot, 1981; Saridis, 1983; Meystel, 1988). The first depends on coordination and integration at all levels in the manufacturing system. The second problem is that of automatic programming of the elements of the system. The control of manufacturing can be considered as having two elements. The first, which we shall call *logical control* relates to the coordination of *events* required to satisfy ordering constraints on event sequences and is adequate for a sequence of process operations. The second, termed *geometric and dynamic control*, relates to the

determination of the geometry and dynamic parameters of motion of the elements of the system.

In this chapter we focus on the production route planning problem in which only a general description of a task is provided (Definition 4.1.1).

In other words, the route planning problem is to find for a given task the most efficient sequence of machines through which parts must flow during repetitive task executions. Such a route is called the *fundamental plan* of the workcell action.

Searching for the most efficient plan requires that we define plan comparison criteria. One such criterion is that the sequence of operations and robot/machine actions related to it should minimize the mean flow time of parts (Kusiak, 1990; Bedworth et al., 1991). The information necessary to evaluate the efficiency of a plan is usually available during the actual execution of the plan in a real workcell or in its simulation model (Maimon, 1987; Jacak and Rozenblit, in press; Jacak and Rozenblit, 1994). The flow time of every job is the sum of the *processing* time, the *waiting* time, and the time of interoperational moves, called the *transfer* time. The waiting and transfer times depend on the order of operations in a task and on the interpretation of robot movements. Each route determines different topology of robot motion trajectories, different deadlock avoidance conditions, and a different job flow time. Therefore, route planning is a critical issue in all manufacturing problems.

4.2.2.1. Basic terminology of concurrent technological process planning. In the process planning phase, we must find an ordered sequence of technological operations from *Task* (Definition 4.1.1), called a *process*, with a minimum number of deadlock cases. This problem is related to operations scheduling (Kusiak, 1990; Jones and McLean, 1986). To explain it in more detail, we introduce some additional notions.

Definition 4.2.2 (The Machining Process). The ordered execution sequence of L operations of *Task* (Definition 4.1.1) is called a pipeline sequential machining process

$$\wp = (o_1, o_2, \dots, o_L) \quad (4.4)$$

if the following conditions hold:

- if for two operations o_i and o_j from *Task*, $o_i \prec o_j$, then $i < j$
- for each $i = 1, \dots, L - 1$ there exists a robot which can transfer a part from machine d_i (or store m_i) assigned to operation o_i to machine d_{i+1} assigned to operation o_{i+1} , i.e.,

$$(\exists r \in R)((o_i, o_{i+1}), r) \in \beta$$

A process can be realized by different sequences of technological devices (called *resources*) required by successive operations from the list \mathbf{p} at the time of their execution. This set of new sequences, denoted \mathbf{P} , is called the *production route*. A production route $\mathbf{p} \in \mathbf{P}$ is an ordered list of resources which has at most $2L + 1$ stages, where L denotes the length of the list \mathbf{p} . The route is created on-line during the execution of the technological operations.

Definition 4.2.3 (The Production Route). The production route is denoted by

$$\mathbf{p} = (m_f, \text{res}(o_1), \text{res}(o_2), \dots, \text{res}(o_L), m_o) \quad (4.5)$$

where $\text{res}(o_i) = \alpha(o_i) = d_i$ if there exists a robot $r \in \beta(o_i, o_{i+1})$ which can transfer parts directly from d_i to d_{i+1} and $\text{res}(o_i) = (d_i, m_i)$ if the robot r can transfer parts only from the production store m_i to d_{i+1} . We assume that there always exists a robot transferring parts from d_i to m_i . By m_f and m_o we denote the feeder and the output conveyor, respectively.

Each execution of a process is called a *job*. A job J_b is characterized by a production route \mathbf{p} , its *start time* t_{J_b} , and two *status functions*:

$$\eta_{J_b} : T \rightarrow \mathbf{p} \quad (4.6)$$

and

$$\omega_{J_b} : T \rightarrow \{0, 1\} \quad (4.7)$$

defined as follows:

The *stage function* $\eta_{J_b}(t) \in \mathbf{p}$ indicates the stage of job J_b with respect to production route \mathbf{p} at time t , where $\eta_{J_b}(t) = \mathbf{p}(0) = m_o$ means that the first resource in the production route $\mathbf{p}(1)$ has not yet been allocated to the job. $\eta_{J_b}(t) = \mathbf{p}(2L) = m_L$ indicates that the job has been completed. $\eta_{J_b}(t) = \mathbf{p}(i)$ means that the operation o_j is being executed and that a part is currently being processed by the workstation $\mathbf{p}(i)$ from the route \mathbf{p} (or is at a store).

The *wait function* $\omega_{J_b}(t) \in \{0, 1\}$ indicates whether the job J_b is waiting for the next resource in the production route. If $\omega_{J_b}(t) = 1$ the operation o_j has finished at the workstation $\mathbf{p}(i) = \eta_{J_b}(t)$ (where $i = 2j$ or $i = 2j - 1$) and the job is waiting for the next resource (workstation) $\mathbf{p}(i + 1)$. Otherwise, $\omega_{J_b}(t) = 0$. Transitions in the wait function (from 0 to 1) occur when a job finishes using the resource for the current stage.

The production route \mathbf{p} has L stages. During the stage i , the operation o_i is executed and thus the resource (machine) $d_i = \alpha(o_i)$ is required for a finite period of time. After this interval, if it is not possible to allocate the store $m_i = \alpha(o_i)$ to the job for some waiting time units, the next machine $d_{i+1} = \alpha(o_{i+1})$ must be assigned to the job. This allocation strategy places a higher priority on the allocation of machines than on the allocation of stores. The job continues to hold the resource

d_i or m_i and cannot advance to stage $i + 1$ until the machine d_{i+1} is allocated to it. This creates a potential for deadlocks among a set of successive executions of processes in the cell. As we noted previously, the process being planned should minimize the number of deadlock cases. Avoiding deadlock causes an increase in job waiting time and consequently an increase in the makespan. Therefore, in designing the cell planning and control algorithm one needs to examine carefully the deadlock-avoidance conditions and their effects on the entire process. A quality criterion of process planning should be defined to assess how well the algorithm performs.

In the workcell considered in this chapter, the so-called *circular-wait deadlock* among pipeline processes can occur.

Definition 4.2.4 (Circular-Wait Deadlock). Circular wait occurs if there is a closed chain of jobs in which each job is waiting for a machine held by the next job in the chain (Coffman et al., 1971; Deitel, 1983).

4.3. Production Routes — Fundamental Plans of Action

The route planning algorithm should take into account the conditions for deadlock avoidance. In order to formulate a quality criterion of process planning, we first describe the procedure of deadlock avoidance presented in (Banaszak and Roszkowska, 1988; Krogh and Banaszak, 1989).

4.3.1. Quality Criterion for Route Planning

To avoid blocking, the production route \mathbf{p} is partitioned into a unique set of Z sublists called *zones*,

$$\mathbf{p} = (z_k | k = 1, \dots, Z) \quad (4.8)$$

where every zone has the form $z_k = (s_k u_k)$, where $u_k = (u_k^i | i = 1, \dots, I(k))$ is the sublist of resources which appear only once in a production route. Such resources are called unshared resources. In addition, if $p(i) = p(i + 1)$ and there exists no $\mathbf{p}(k) = \mathbf{p}(i)$ ($k \neq i, i + 1$), then $p(i)$ and $p(i + 1)$ are unshared resources, too. In addition, $s_k = (s_k^j | j = 1, \dots, J(k))$ is the sublist of resources which are used more than once in a route.

The sublists u_k and s_k of the zone z_k are referred to as the unshared and shared subzones of z_k , respectively. $I(k)$ and $J(k)$ denote the number of unshared and shared resources in subzones u_k and s_k , respectively.

Now we consider the deadlock problem for a machining system and an algorithm for preventing the occurrence of deadlocks based on the method described

in (Banaszak and Roszkowska, 1988; Krogh and Banaszak, 1989). The deadlock-avoidance algorithms proposed for computer operating systems (Coffman et al., 1971; Deitel, 1983) are not efficient for pipeline processes. They do not incorporate the production route information which indicates the specific order in which resources of a cell must be allocated and deallocated to jobs. According to the definition of a production route \mathbf{p} , we can specify the function $Next_{Jb}(p(i), t) = p(i + 1)$. This function determines the next resource required by the job Jb holding the resource $p(i)$.

A *resource allocation policy* is a rule which assigns a resource d to a job Jb for which $Next_{Jb}(t) = d$. In general, unrestricted allocation of an available resource required by a job can lead to a deadlock among a set of jobs in the system. To avoid deadlocks, we use the *restricted allocation policy* proposed in (Banaszak and Roszkowska, 1988; Krogh and Banaszak, 1989).

Let $C(d)$ denote the capacity of a machine, i.e., the maximum number of jobs to which the machine (or store) can be allocated. $h(d)$ denotes the number of jobs which are currently allocated to the machine d . Assume that the machine $Next_{Jb}(t) = v$ belongs to the zone z_k , i.e.,

$$v \in z_k = (s_k^1, \dots, s_k^{J(k)}, u_k^1, \dots, u_k^{I(k)}) \quad (4.9)$$

The restricted allocation policy is defined by the following rules:

- A. If $(v = u_k^j \text{ and } j \neq 1 \text{ or } v = s_k^j \text{ and } h(v) < C(s_k^j) - 1) \text{ or } (v = u_k^1 \text{ and } h(u_k) < C(u_k) - 1)$, then resource v can be allocated to job Jb , i.e., $\eta_{Jb}(t^+) = v$.
- B. If $v = s_k^j$ and $h(v) = C(s_k^j) - 1$, then resource v can be allocated to job Jb if $(\forall s_k^i \neq s_k^j)(h(s_k^i) < C(s_k^i))$ and $h(u_k) < C(u_k)$.
- C. If $v = u_k^1$ and $h(u_k) = C(u_k) - 1$, then resource v can be allocated to job Jb if $(\forall s_k^i)(h(s_k^i) < C(s_k^i))$.

Here $C(u_k) = \sum_{i=1}^{I(k)} C(u_k^i)$.

It means that if the current capacity of the required resource v is $h(v) < C(v) - 1$, then the resource can be allocated to job Jb , or if the capacity of the required resource v is $h(v) = C(v) - 1$, then the resource can be allocated to job Jb only if there does not exist another fully allocated resource in the zone.

In both cases the entire unshared zone is treated as one resource. The complete formal explanation of the restriction allocation policy is presented in (Krogh and Banaszak, 1989).

Fact 4.3.1. *The circular-wait deadlock can never occur under the restricted allocation policy described in rules A–C) (Krogh and Banaszak, 1989).*

□

□

We use the above conditions to formulate the *planning quality criterion*. Our goal is to optimize the production rate, i.e., minimize the job waiting times. Fact 4.3.1 can be derived. Let zone z_k have $n(k) = J(k) + 1$ elements. $J(k)$ is the number of shared machines and 1 denotes all the unshared devices. Assume that the probability of a machine being completely full is equal to w , i.e., $\mathcal{P}[h(d) = C(d)] = w$.

Fact 4.3.2. *If there exists a job J_b such that $Next_{J_b}(t) = x \in z_k$ and $h(x) < C(x)$, then the probability that the job J_b waits for processing is*

$$\mathcal{P}[J_b = wait] = w(1 - (1 - w)^{n(k)-1}) \quad (4.10)$$

□

□

Proof A job J_b will have to wait if the resource required by it has only one place free in its input buffer and there exists a second, completely full resource in its zone. Thus

$$\begin{aligned} P\{\omega_{J_b}(t^+) = 1\} &= \binom{n(k)-1}{1} w^2 (1-w)^{n(k)-2} \\ &\quad + \binom{n(k)-2}{2} w^3 (1-w)^{n(k)-3} + \dots + \binom{n(k)-1}{n(k)-1} w^{n(k)} \\ &= w(1 - (1-w)^{n(k)-1}) \end{aligned}$$

The more elements (machines) that belong to a zone (the subzone u_k is treated as one element), the higher is the probability that a job will wait for resource allocation. Thus, the production routes should contain zones with a minimum number of elements. Based on previous remarks, we introduce the measure of route quality.

Let $\mathbf{p} = (z_k | k = 1, \dots, Z)$ be the production route of the process \wp , where $z_k = (s_k u_k)$, with $u_k = (u_k^i | i = 1, \dots, I(k))$ the unshared subzone and $s_k = (s_k^j | j = 1, \dots, J(k))$ the shared subzone. In several cases the first zone may contain only an unshared subzone (i.e., $z_1 = u_1$) and the last subzone may contain only a shared subzone (i.e., $z_Z = s_Z$). By $n(k)$ we denote the number of elements in the zone z_k . If $z_k = (s_k u_k)$, then $n(k) = J(k) + 1$; if $z_k = u_k$, then $n(k) = 1$, and if $z_k = s_k$ then $n(k) = J(k)$. Also, if there exist resources s_k^i and s_k^{i+1} in the subzone s_k such that $s_k^i = s_k^{i+1}$, then $n(k) = (J(k) - 1) + 1$.

The measure of route quality is defined as

$$v_1(\mathbf{p}) = \max_k \{n(k) | k = 1, \dots, Z\} \quad (4.11)$$

Fact 4.3.3. *If a production route is composed of only unshared resources, then the measure of route quality is minimal ($v_1 = 1$). If it contains only shared resources, then the measure of route quality has a maximum value.*

□

□

Another way to construct a heuristic measure of deadlock cases in the production route is to approximate the probability that the job will wait for processing, called the *probability of waiting* (Rogalinski, 1994).

In order to calculate the probability of waiting, let us define the probability that the device d is completely full as a function of the capacity of this device and of the times of the technological operations performed by it.

The probability that device d is completely full is given by

$$\mathcal{P}[h(d) = C(d)] = w(d) = \text{Time}(d)/(C(d) * MT) \quad (4.12)$$

where $\text{Time}(d) = \sum_{o \in (\alpha)^{-1}(d)} t_{\text{proc}}(o)$ and $MT = \max_{d \in D} (\text{Time}(d))$. Here $t_{\text{proc}}(o)$ denotes the processing time of operation o .

The definition just proposed of the probability that device d is completely full satisfies the property, $w(d) \in [0, 1]$, i.e., the probability of an impossible event (filling up a device which has a limitless capacity or does not perform any time-consuming operation) equals 0, and the probability of a certain event (filling up a device which has capacity equal to 1 and performs the most time-consuming operations) equals 1.

When the probability of waiting for the zone z_k is calculated the whole nonempty unshared subzone $u_k = (u_k^i | i = 1, \dots, I(k))$ is treated as one resource d^* which has a capacity given by

$$C(d^*) = \sum_{i=1}^{I(k)} C(u_k^i) \quad (4.13)$$

and $\text{Time}(d^*)$ is given by

$$\text{Time}(d^*) = \max\{\text{Time}(u_k^i) | i = 1, \dots, I(k)\} \quad (4.14)$$

Thus, the probability that the nonempty unshared subzone u_k is completely full, i.e.,

$$\sum_{i=1}^{I(k)} h(u_k^i) = \sum_{i=1}^{I(k)} C(u_k^i)$$

takes the form

$$w(d^*) = \text{Time}(d^*)/(C(d^*) * MT) \quad (4.15)$$

If the unshared subzone u_k is empty, the probability of filling it up equals zero.

The state of possible deadlock in the zone z_k occurs when there are at least two resources completely full in the zone z_k . Because one device can belong to various zones or to various production routes we assume that different resources are filled up independently. To calculate the deadlock factor of the zone z_k three cases are considered:

- There is no device in the zone z_k for which the probability of filling up equals 1
- There is exactly one resource in the zone z_k for which the probability of filling up equals 1
- There are at least two devices in the zone z_k for which the probability of filling up equals 1

Let $w(i)$ denote the probability that the i th resource in the zone z_k is completely full and $w(0)$ denote the probability that resource d^* which replaces the unshared subzone u_k is completely full. According to the assumption that resources are filled up independently, the waiting probability of the zone z_k takes the form

$$\mathcal{P}(z_k) = \begin{cases} 1 - \prod_{i=0}^{J(k)} (1 - w(i)) \left[1 + \sum_{i=0}^{J(k)} \frac{w(i)}{1 - w(i)} \right] & \text{if } w(i) \neq 1 \text{ for } i = 0, \dots, J(k) \\ 1 - \prod_{\substack{i=0 \\ i \neq m}}^{J(k)} (1 - w(i)) & \text{if } (\exists m)(w(m) = 1) \wedge (\forall i)(i \neq m \Rightarrow w(i) \neq 1) \\ 1 & \text{if } (\exists m, n)(m \neq n \wedge w(m) = w(n) = 1) \end{cases} \quad (4.16)$$

where $w(0) = w(d^*)$.

Fact 4.3.4. *It is easy to prove the following facts (Rogalinski, 1994):*

- Exchange of a succession of resources in the unshared subzone u_k or in the shared subzone s_k does not change the probability of waiting*
- Increase of the probability of filling up any resource in the zone z_k (all unshared resources are treated as one resource d^*) can at most increase the probability of waiting*
- Adding to the shared subzone s_k a new resource for which the probability of filling up is different from 0 can at most increase the probability of waiting*

□

□

Now we can state the production process planning problem with the minimization of the probability of waiting. The production route \mathbf{p} can be partitioned into a unique set of z_k zones. Then the quality measure of the production route \mathbf{p} is as follows:

$$\nu_2(\mathbf{p}) = \max_k \{\mathcal{P}(z_k) | k = 1, \dots, Z\} \quad (4.17)$$

The function $v(\mathbf{p})$ (v_1 or v_2) is used to evaluate the technological process being planned. Our task is to find an ordered sequence of operations from *Task* which is feasible and which minimizes the function $v(\mathbf{p})$. This problem may have more than one solution.

4.3.2. Process Route Planning Algorithm

The function $v(\mathbf{p})$ is used to evaluate the technological process being planned. The Process planning problem can be formulated as follows:

Find a feasible, ordered sequence \wp of operations from *Task* (Definition 4.1.1) and production route \mathbf{p} which minimizes the function $v(\mathbf{p})$.

This is a permutation problem which may have more than one solution. To solve it, we proceed as follows:

Task is represented by a directed acyclic graph. Let $\wp^K = (o_1, o_2, \dots, o_K)$ ($K < L$) be a subprocess of \wp and $\mathbf{p}^K = (z_k | k = 1, \dots, Z^K)$ be a subroute of minimal route \mathbf{p} . Let START denote the list of operations which have no immediate predecessors in relation \prec . To solve the planning problem under consideration, we propose the backtracking procedure in Figure 4.4.

The *evaluation function* $f(q)$ is calculated as follows: Let PATH be a list (o_1, \dots, o_j) . Let $q \in \text{Suc}(o_j)$.

- Create temporarily the subprocess $\wp^{j+1} = (o_1, o_2, \dots, o_j, q)$. Calculate its production subroute \mathbf{p}_{\min}^{j+1} and decompose it into zones $(z_k | k = 1, \dots, Z^{j+1})$.
- Calculate the measure of route quality $\nu(\mathbf{p}_{\min}^{j+1})$ and the $\text{set } f(q) = \nu(\mathbf{p}_{\min}^{j+1})$.

The output list PROCESSES in the route planning procedure contains only feasible, ordered sequences of technological operations realizing the *Task*. The procedure expands the OR-graph of operations depthwise, one at a time, along one path (\wp) with different possible choices of the return point in the case of failure. The backtracking node lies either at the next higher level, if operations at the $\text{path-}\wp$ level are not admissible, or at a much higher level, if a contradiction occurs.

Fact 4.3.5. *If there exists a production route for a given Task, then the backtracking procedure finds the optimal ordered sequence of operations \wp , computed by the function v .*

□

□

Proof Let $\wp^j = (o_1, o_2, \dots, o_j)$ ($j < L$) be a subprocess of \wp and $\mathbf{p}^j = (z_k | k = 1, \dots, Z^j)$ be a subroute of route \mathbf{p} . It can be seen that $\nu(\mathbf{p}^j) \leq \nu(\mathbf{p})$. According to step (*) of the procedure, we eliminate the subprocess $\wp^{j+1} =$

```

Form START,  $i = 1, m = 0, \nu^{opt} = \infty$ 
while (START  $\neq \emptyset$ ) {
    Remove an operation from START and put it on PATH
    BACK_i = START
    while (PATH  $\neq \emptyset$ ) {
        if (PATH =  $\emptyset$ ) {
             $m = m + 1$ 
             $\wp_m = \text{PATH}, \mathbf{p}^m = \mathbf{p}$ 
            Put  $\wp_m$  on PROCESSES
             $\nu^{opt} = \nu(\mathbf{p}_{min}^m)$ 
            (**) Remove nonoptimal sequences  $\wp$  from PROCESSES
        } else {
             $i = j$  for  $o_j$  last operation in PATH
            Generate  $Suc(o_j)$ 
            if ( $Suc(o_j) = \emptyset$ ) {
                remove  $o_j$  from PATH
                remove  $o_j$  from  $Suc(o_{j-1})$ 
                 $i = j - 1$ 
                continue
            }
            if ( $(\exists q \in Suc(o_j) \& \exists o_l \in \text{PATH}) (q \prec o_l)$ ) {
                remove all elements of PATH beginning with  $o_l$ 
                remove  $o_l$  from  $Suc(o_{l-1})$ 
                 $i = l - 1$ 
                continue
            }
             $i = i + 1$ 
            BACK_i =  $\{q \in Suc(o_j) | f(q) = \min_j f(o_j)\}$ 
            remove an operation  $o_i$  from BACK_i
            (*) if ( $f(o_i) \leq \nu^{opt}$ ) {
                insert  $o_i$  at end of PATH
                continue
            }
        }
    }

    Find max index  $i$  for which BACK_i  $\neq \emptyset$ 
    Remove all elements of PATH beginning with  $o_i$ 
}

```

Figure 4.4. Route planning procedure.

$(o_1, o_2, \dots, o_j, q)$ if $v(\mathbf{p}^{i+1}) > v(\mathbf{p}^*)$, where \mathbf{p}^* is a previously obtained optimal route. It is clear that

$$v(\mathbf{p}) \geq v(\mathbf{p}^{i+1}) > v(\mathbf{p}^*) \quad (4.18)$$

In step (**) the suboptimal solutions are eliminated.

Each \wp from the set PROCESSES determines a different fundamental plan of robot and machine actions and a different law of workcell control. To minimize the flow time of jobs, variants of the fundamental plan should be tested.

Example 4.3.1. For the task of the **Example_Cell** the following routes, among others, can be computed, with different values of quality measure function v :

$$\mathbf{p}_1 = (m_{01}, d_{01}, d_{02}, |m_{02}, d_{03}, m_{02}, d_{01}, m_{02}, d_{04}, |d_{03}, d_{05}, m_{03})$$

$$\mathbf{p}_2 = (m_{01}, d_{01}, m_{02}, d_{03}, d_{04}, |d_{03}, m_{02}, d_{02}, |d_{01}, m_{02}, d_{05}, m_{03})$$

$$\mathbf{p}_3 = (m_{01}, m_{02}, d_{03}, d_{04}, |d_{03}, m_{02}, d_{01}, d_{02}, |d_{01}, m_{02}, d_{05}, m_{03})$$

where $| \dots |$ denotes a zone.

The route planner generates only one optimal production route \mathbf{p}^* for which $v_1(\mathbf{p}^*) = 2$:

$$\mathbf{p}^* = (m_{01}, d_{01}, d_{02}, |d_{01}, m_{02}, |d_{03}, d_{04}, |d_{03}, d_{05}, m_{03})$$

The optimal route \mathbf{p}^* is shown in Figure 4.5.

4.3.3. Process Route Interpreter

Based on a sequence of operations \wp and its route \mathbf{p} the fundamental plan of workcell actions is first created. This fundamental plan describes the decomposition of a technological task into an ordered sequence of robot and machine actions which are used to realize the task.

The model of the robotic agent is reduced to a fundamental transfer action:

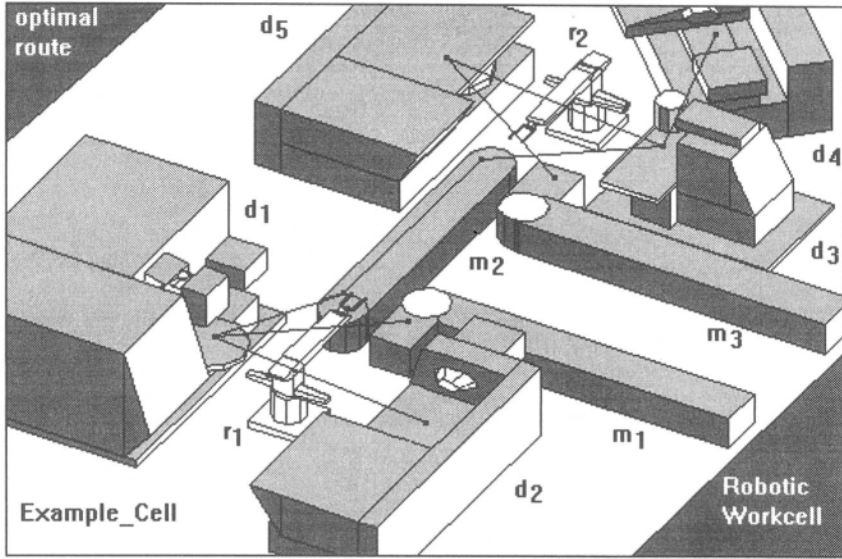
$$Action_robot(r) = \text{Transfer } \mathbf{part} \text{ From } \mathbf{a} \text{ To } \mathbf{b}$$

which denotes the transfer of a part from a workstation or a store \mathbf{a} to a workstation or a store \mathbf{b} .

The machine's activity is represented by

$$Action_machine(d) = \text{Execute } \mathbf{o} \text{ On } \mathbf{b}$$

which is interpreted as the beginning of the execution of an operation \mathbf{o} from the set O on the machine \mathbf{b} .

Figure 4.5. Optimal production route p^* .

Based on the sequence of operations ρ obtained through task planning, we create a fundamental plan of actions for the cell's components, i.e.,

$$Plan = (Action_i | i = 1, \dots, L) \quad (4.19)$$

Each $Action_i$ consists of three parts which determine: (1) the preconditions of an action, (2) the robot's motion parameters to execute the i th operation of ρ , and (3) the action execution parameters.

The i th action has the following form:

$$Action_i = [Cond(o_i), Transfer(o_{i-1}, o_i), Execute(o_i)]$$

The $Cond(o_i)$ segment describes the preconditions which must be satisfied in order for the operation o_i to be executed. It also establishes the geometric parameters for the robot's motion trajectories. The preconditions are formulated as a function of both the workcell and job state.

Let $i-B(d)$ denote the state of the i th position of the workstation d 's buffer $B(d)$, where $1 \leq i \leq C_b(d)$. This state can be characterized as follows:

- $i-B(d) = (0, 0)$: the position is free
- $i-B(d) = (j, 0)$: the position is occupied by a part before the operation o_j
- $i-B(d) = (j, 1)$: the position is occupied by a part after the operation o_j

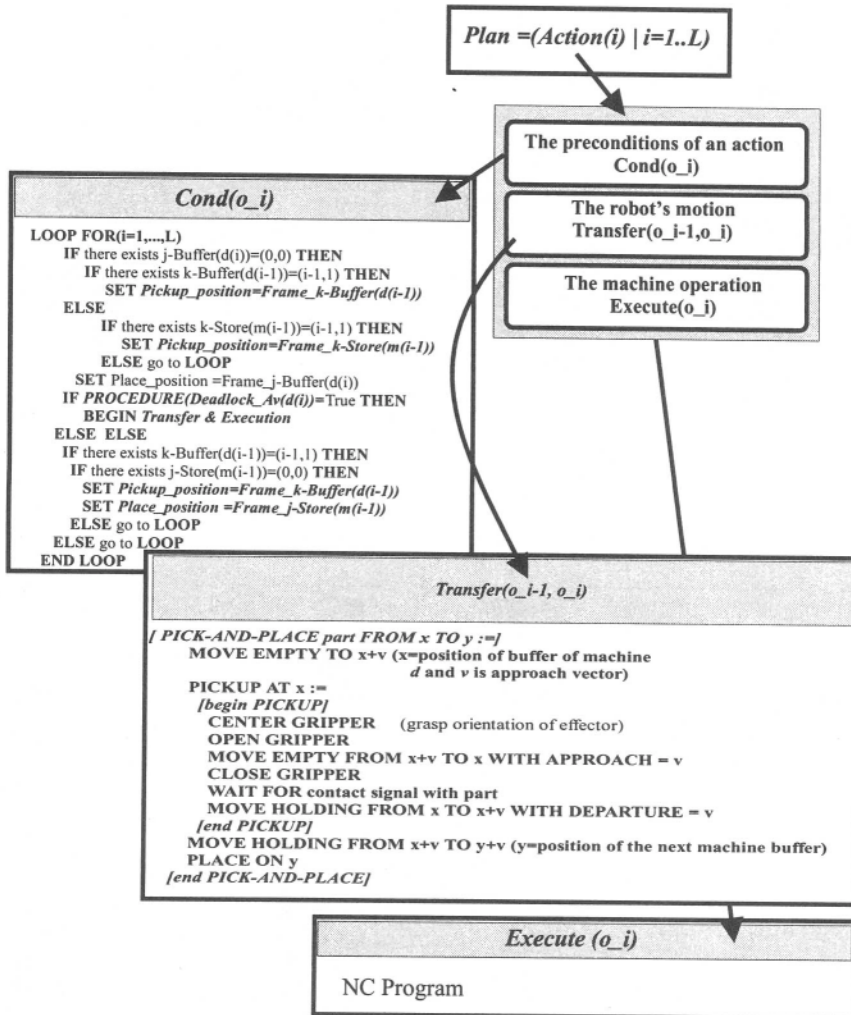


Figure 4.6. Interpretation of a fundamental action.

Using the above specification, the $\text{Cond}(o_i)$ segment can be defined by the following instructions of the task-oriented robot programming language (Figure 4.6):

```

LOOP FOR (i=1, ..., L)
  IF there exists j-Buffer(d(i))=(0,0) THEN
    IF there exists k-Buffer(d(i-1))=(i-1,1) THEN
      SET Pickup-position=Frame_k-Buffer (d(i-1))
    ELSE
      IF there exists k-Store (m(i-1))=(i-1,1) THEN
        SET Pickup_position=Frame_k-Store(m(i-1))
      ELSE go to LOOP
      SET Place_position =Frame_j-Buffer (d(i))
    IF PROCEDURE (Deadlock Av(d(i)) =True THEN
      BEGIN Transfer & Execution
    ELSE ELSE
      IF there exists k-Buffer(d(i-1))=(i-1,1) THEN
        IF there exists j-Store(m(i-1))=(0,0) THEN
          SET Pickup_position=Frame k-Buffer (d(i-1))
          SET Place_position =Frame_j-Store(m(i-1))
        ELSE go to LOOP
      ELSE go to LOOP
    END LOOP

```

By $d(i)$ and $m(i)$ we denote respectively the workstation and store assigned to the operation o_i . The call **PROCEDURE (Deadlock Av (d(i)))** checks to see if the deadlock avoidance conditions for device $d(i)$ are satisfied. If they are satisfied, then the subactions Transfer and Execution are activated.

In addition, the geometrical parameters of the Transfer subaction are established and stored in **Frames Table**.

The **Frames Table** determines the geometrical initial and final positions and orientations of the robot's effector for each robot movement. For movements which realize the transfer of parts, the initial and final positions result directly from the sequence of machines in route **p**. The parameters indicate the geometrical places between which Transfer has to be performed. This subaction is realized by a robot servicing the workstation $d(i)$.

The elementary robot actions can be expressed as basic macro-instructions. The basic macro-instructions are (Jacak and Rozenblit, 1992a; Jacak and Rozenblit, 1992b; Jacak and Rozenblit, in press):

```

MOVE (EMPTY, HOLDING) TO position
PICKUP AT position
PLACE ON position
WAIT FOR sensor input signal
INITIALIZE output signal
GRASP, OPEN GRIPPER, and CLOSE GRIPPER

```

The basic instructions may be combined into higher level macros, e.g., the **PICK-AND-PLACE** instruction (Jacak and Rozenblit, 1992b).

In this set of instructions, the action $\text{Transfer}(o_{i-1}, o_i)$ is interpreted as the **PICK-AND-PLACE part FROM x TO y** macro, where **x** denotes the geometrical data of a workstation's buffer (store), and **y** is the position and orientation

of the buffer of the next workstation. The parameters x and y are determined by $\text{segmentCond}(o_i)$, namely

$x = \text{Pickup_frame}$

$y = \text{Place_frame}$

This macroinstruction is decomposed into a sequence of more primitive instructions as illustrated below and in Figure 4.6:

```
[ PICK-AND-PLACE part FROM x TO y :=]
  MOVE EMPTY TO x+v (x=position of buffer of machine d
                    and v is approach vector)

  PICKUP AT x :=
  [begin PICKUP]
    CENTER GRIPPER      (grasp orientation of effector)
    OPEN GRIPPER
    MOVE EMPTY FROM x+v TO x WITH APPROACH = v
    CLOSE GRIPPER
    WAIT FOR contact signal with part
    MOVE HOLDING FROM x TO x+v WITH DEPARTURE = v
  [end PICKUP]
    MOVE HOLDING FROM x+v TO y+v (y=position of the
                                next machine buffer)

  PLACE ON y
[end PICK-AND-PLACE]
```

The above sequence of instructions is used to synthesize automatically the robot's motion program. The instructions for the action Execute can be translated in a similar manner.

The instructions may be interpreted in various ways. The implementation and interpretation of the fundamental plan is carried out using the *automatic task-level programming* approach in which detailed paths and trajectories, gross and fine motion, and grasping and sensing instructions are specified. Each route determines a specific topology of robot motion tracks, distinct deadlock-avoidance conditions, and a certain job flow time. Therefore, route planning is critical for problems such as the maximum-rate and minimum-jobs-in-process optimization problems.

Consequently, the automatic design of a geometrical motion route must determine, for each robot r servicing the process, a set of cell-state-dependent time trajectories of the robot's motions. To generate trajectories of robot motions, a geometrical model of the virtual robotic system as well as models of the robot's kinematics and dynamics need to be available.

CHAPTER 5

Off-Line Planning of Robot Motion

The production route \mathbf{p} for a machining task determines the parameters of the robot's movements and manipulations (such as initial and final positions) needed to carry out this task. The set of all of a robot's motions between devices and stores needed to perform a given process is called a *geometric route* or *geometric control*.

Consequently, the automatic design of a geometric route must determine, for each robot r servicing the process, a set of cell-state-dependent time trajectories of the robot's motions.

To generate trajectories of robot motions, a geometrical model of the virtual robotic system as well as models of the robot's kinematics and dynamics need to be available.

Based on the sequence of operations \mathbf{p} and its route \mathbf{p} the positions table (**Frames_Table**) for all motions of each robot is first created. The **Frames-Table** determines the geometrical initial and final positions and orientations of the robot's effector for each robot movement. For movements which realize the transfer of parts, the initial and final positions result directly from the sequence of machines in route \mathbf{p} .

The robot motion trajectory planning process is performed in two stages: (1) planning of the geometrical track of the motion, and (2) planning of the motion dynamics along a computed track.

5.1. Collision-Free Path Planning of Robot Manipulator

The path planner creates variants of the geometric tracks of the manipulator which executes a given action. It uses robot-dependent planning techniques and a discrete system formalism (Jacak, 1989b; Jacak, 1989a; Jacak, 1991). Such a path planner should be able to determine the collision-free track of robot motion from the initial to the final effector locations based on geometric and kinematic descriptions of the robot and its environment and the initial and final positions of the effector end.

The problem of moving in space while avoiding collisions with the environment is known as obstacle avoidance or path planning. For the robot's manipulator, the problem is more complicated than for a mobile robot. Not only must the effector end move to the desired destination without collisions with obstacles, but the links of the arm must also avoid collisions. This problem has been addressed in various ways and is widely reported in the literature (Brooks, 1983; Barraquand et al., 1992; Jacak, 1991; Kircanski and Timcenko, 1992; Latombe, 1991; Lozano-Perez, 1989).

Research in the area of obstacle avoidance can be broadly divided into two classes of methodologies: *global* and *local*. Global methods rely on the description of the obstacles in the configuration space of a manipulator (Latombe, 1991; Lozano-Perez, 1989). Local methods rely on the description of the obstacles and the manipulator directly in the Cartesian workspace (Brooks, 1983; Barraquand et al., 1992; Kircanski and Timcenko, 1992; Jacak, 1989b; Jacak, 1990; Latombe, 1991).

Global methodologies require that two main problems be addressed. First, the obstacles must be mapped into the configuration space of the manipulator. Second, a path through the configuration space must be found for the point representing the robot arm.

To obtain a uniform representation of the robot and its environment, a transformation of the geometrical model of the work scene from the base Cartesian frame into the joint space is often performed (Latombe, 1991; Lozano-Perez, 1989). Transformations called configuration-space methods are very complex and inefficient, particularly for the redundant robots. Such methodologies have several disadvantages. The algorithms necessary for configuration space generation are computationally intensive, and the computational costs increase quickly as a function of the robot's degrees of freedom, at least exponentially for geometric search techniques (Latombe, 1991; Lozano-Perez, 1989). Thus, they are suited only for off-line path planning and cannot be used for real-time obstacle avoidance. An immediate consequence is that global methods are difficult to use for obstacle avoidance in dynamic environments. Also, it is very difficult to describe complicated motion planning task using such algorithms, such as those tasks arising when two manipulators cooperate.

An alternative to global methodologies is provided by local ones (Latombe, 1991). The main advantage of local techniques is that they are less computationally intensive than global ones. Thus they can be used in real time control. Further, they provide the necessary framework to deal with dynamic environments.

Most planning methods are based on arbitrary discretization of either the *joint space* or the *Cartesian space* of the robot's manipulator. This discretization decides the accuracy with which the terminal point of motion can be reached, as well as the exactness of the relation between the manipulator and the obstacle (Jacak, 1989b; Jacak, 1989a; Jacak, 1991).

Due to the discretization, the number of possible manipulator configurations is finite. The search through the whole set of configurations requires a huge computational effort and is unrealistic in real time. However, the process of a local (partial) search of the configuration set, directed toward finding a path which is not always the shortest one, can be described by an implicit graph of configurations. The computational efficiency of such graph-searching methods depends first on the ease of generating new nodes (configurations) of the implicitly extended graph, the ease of testing whether the generated nodes are feasible and nonrepeatable, and the ease of calculating the cost and heuristic evaluation functions. Hence, a model of robot kinematics should be a mathematical system endowed with an ability to produce a new configuration of the manipulator on the basis of an old configuration and a desired input signal. The model should facilitate direct analysis of a robot location with respect to objects in its real world environment and should be not too complex computationally.

5.1. 1. Neural and Discrete Models of Robot Kinematics

The most suitable model of robot kinematics is a discrete dynamic system M defined as

$$M = (Q, U, Y, f, g) \quad (5.1)$$

where:

- The set Q denotes the state of the manipulator. The state $q \in Q$ determines the position of the manipulator, called the manipulator's configuration, in the real space in which the physical robot operates. The most frequently used description of a state of the manipulator with n degrees of freedom is its representation as a vector of joint variables:

$$q = (q_i | i = 1, \dots, n)^T \quad (5.2)$$

where $q_i \in Q_i = [q_i^{\min}, q_i^{\max}]$ is the range of change of the i th joint angle.

- U denotes the set of input signals of M .
- The output Y ensures the possibility of geometrically representing the robot's body in a 3D base frame. For this purpose it is convenient to use a skeleton model of the manipulator described as the vector

$$y = (P_i | i = 0, 1, \dots, n)^T \quad (5.3)$$

where $P_i = (x_i, y_i, z_i) \in E_0$ is the point in the base coordinate frame describing the current position of the i th joint and P_n is the position of the effector end.

- The function $f: \mathcal{Q} \times U \rightarrow \mathcal{Q}$ is a discrete one-step transition function of the form

$$q(k+1) = f(q(k), u(k)) \quad (5.4)$$

where k is the discrete moment of time.

- $g: \mathcal{Q} \rightarrow Y$ is an output function of the form

$$y(k) = g(q(k)) = (g_i(q(k)) | i = 0, 1, \dots, n)^T, \quad g_0(q(k)) = P_0 = \text{const} \quad (5.5)$$

The properties of such a model of the robot kinematics depend on the method of specification of its components, specifically on the input set U .

There are two ways to construct such a model. The first one involves discretely changing the state in the joint space and then translating it into the base Cartesian space. The second one involves constructing a model operating on the robot arm directly in the discretized base frame. These two approaches to the trajectory planning problem are referred to as *joint space planning* and *Cartesian space planning*, respectively.

5.1.2. Neural Network-Based Path Planning in Joint Space

One way to construct a model of the robot's kinematics is based on an arbitrary discretization of angle increments of the manipulator joints (Jacak, 1989b).

5.1.2.1. Discrete model of robot kinematics. In order to specify the input set U of the model M the discretization of the robot's joint space \mathcal{Q} is performed,

$$q_i = q_i^{\min} + j \cdot \delta q_i \quad \text{and} \quad j \in \{0, 1, \dots, I_i\} \quad (5.6)$$

where $q_i^{\max} = q_i^{\min} + I_i \cdot \delta q_i$.

Using the fact that all the angles can change only by a define increment, we define the input set U of the model M as

$$U = \times \{u_i | i = 1, \dots, n\} \quad (5.7)$$

where $u_i = \{-\delta q_i, 0, \delta q_i\}$ is the set of possible (admissible) directions of changes of the i th joint angle.

Having defined the set U , it is possible to describe the changes of successive configurations of the robot's link as a discrete linear dynamic system of the form

$$q(k+1) = q(k) + \Lambda \cdot u(k) \quad (5.8)$$

where $\mathbf{u}(k) \in U$ is the vector of increments of angles of the joint, and Λ is a diagonal $n \times n$ matrix describing the length of the angle's step changes at each joint, i.e., $\Lambda = \text{diag}[\lambda_i]$.

In order to make it possible to check the configuration with respect to obstacle locations, it is necessary to create an output function g . As stated previously, the manipulator's position in the base frame is represented by the skeleton of the robot arm.

Recall that the i th joint position in Cartesian base space, assuming that all the joint variables q_i are known, is described by the Denavit–Hartenberg matrix (Brady, 1986).

$$T_i = A_1 \cdot A_2 \cdot \dots \cdot A_i \quad \text{and} \quad T_i = [r_i, t_i] \quad (5.9)$$

where A_i is the transformation matrix between the coordinate frame E_i of the i th joint and the coordinate frame E_{i-1} of the $(i - 1)$ th joint, r_i is the rotation matrix, and t_i is the translation vector. The last column of the matrix T_i , i.e., t_i can be used to determine the output function of model M as

$$g(q(k)) = (P_0, t_i(q(k)) | i = 1, \dots, n)^T = (g_i | i = 0, 1, \dots, n)^T \quad (5.10)$$

where $g_i = t_i(q(k)) = P_i$ is the last column of the matrix T_i .

The components of the output function g are highly nonlinear trigonometric functions. Path planning and next control based on the computation of such forward kinematics is computationally expensive and requires frequent calibration to maintain accuracy. In particular, the use of multiple and redundant robot arms makes path planning based on numerical computation extremely difficult.

Therefore it is attractive to develop a neural network which automatically generates the robot kinematics. Neural networks can be used to reduce the computational complexity of the model of robot kinematics. Additionally, neural network can give robots the ability to learn and to self-calibrate (Jacak, 1994b; Kung and Hwang, 1989; Lee and Bekey, 1991).

Neural network approach to the modeling of the output function g . The recent surge of interest in massively parallel algorithms has prompted researchers to investigate the use of neural networks for robotics applications. While not as accurate as traditional numerical methods, these new approaches promise speedups for problems where accuracy is not a fundamental issue. The main idea is to use these methods to obtain good initial values for more accurate fine-tuning operations or to use external feedback loops to increase the accuracy of these methods (Guez and Ahmad, 1998a).

Many researchers in the field of robotics have used neural networks to implement kinematics calculations. The theoretical basis for the applicability of neural networks to computationally complex problems can be found in reports on function approximation by the superposition of elementary functions. This research [(Gallant and White, 1988; Hornik et al., 1990; White, 1990; Hornik,

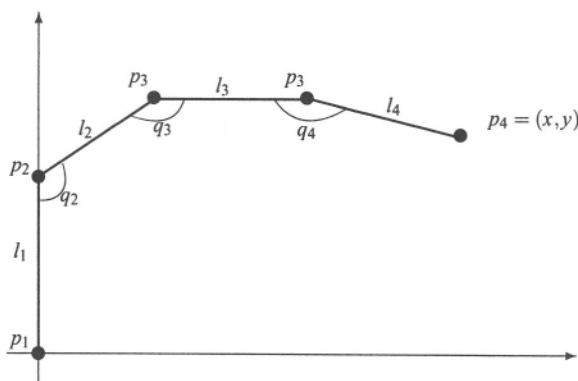


Figure 5.1. Skeleton model of two-dimensional manipulator.

1991; Kreinovich, 1991), among others] shows that *any* function can be approximated arbitrarily closely by a feedforward neural network with only one hidden layer. To reduce the size of the network, it is sometimes beneficial to introduce a second hidden layer (Lippman, 1987; Chester, 1991).

Using this notion of neural networks as universal approximators, there have been several attempts to apply them to solve robot kinematics problems, e.g., (Elsley, 1988; Guez and Ahmad, 1998b; Josin et al., 1988). Further research has shown that the networks used for these problems (mostly feedforward networks trained by backpropagation) provide adequate behavior for small robots, but do not scale up well for larger problems, i.e., to robots with more than three degrees of freedom.

Various improvements have been suggested to remedy this situation. Yeung and Bekey (1989) used *context-sensitive networks* to achieve better scale-up properties. Guo and Cherkassy (1989) proposed the use of Hopfield-type networks. Kung and Hwang (1989) discussed possible neural implementations by a ring VLSI systolic architecture.

SIGMOIDAL NEURAL NETWORK APPROACH: Most neural networks that are used to learn mapping problems (as is the case for the robot kinematics problem) are multilayer feedforward networks composed of sigmoidal units. The terms *multilayer* and *feedforward* describe the topology of the network — in this case, the activation values propagate forward from an input through at least one hidden layer to the output layer. The term *sigmoidal* refers to the activation function of these neurons, which is monotonic, bounded, and differentiated. Such networks can be trained by the most widely used network training algorithm, the *backpropagation algorithm*.

Example 5.1.1. As an example, we investigate the forward and inverse kinematics of a simple two-dimensional 3-DOF manipulator. Such a manipulator is

the simplest redundant robot in two dimensions. Figure 5.1 shows the skeleton of such a robot. For this example, the joint angles q_2 , q_3 , and q_4 were all chosen in the interval $[\pi/2, 3\pi/2]$.

The Cartesian position (x,y) can be calculated from the joint angles (q_2, q_3, q_4) as follows (using trigonometric simplifications):

$$\begin{aligned} x &= t_4^x(q_2, q_3, q_4) = l_2 \sin(q_2) - l_3 \sin(q_2 + q_3) + l_4 \sin(q_2 + q_3 + q_4) \\ y &= t_4^y(q_2, q_3, q_4) = l_1 - l_2 \cos(q_2) + l_3 \cos(q_2 + q_3) - l_4 \cos(q_2 + q_3 + q_4) \end{aligned}$$

The Jacobian matrix of partial derivatives can easily be calculated as

$$\begin{aligned} \frac{\partial t_4^x(q_2, q_3, q_4)}{\partial q_2} &= l_2 \cos(q_2) - l_3 \cos(q_2 + q_3) + l_4 \cos(q_2 + q_3 + q_4) \\ \frac{\partial t_4^x(q_2, q_3, q_4)}{\partial q_3} &= -l_3 \cos(q_2 + q_3) + l_4 \cos(q_2 + q_3 + q_4) \\ \frac{\partial t_4^x(q_2, q_3, q_4)}{\partial q_4} &= l_4 \cos(q_2 + q_3 + q_4) \\ \frac{\partial t_4^y(q_2, q_3, q_4)}{\partial q_2} &= l_2 \sin(q_2) - l_3 \sin(q_2 + q_3) + l_4 \sin(q_2 + q_3 + q_4) \\ \frac{\partial t_4^y(q_2, q_3, q_4)}{\partial q_3} &= -l_3 \sin(q_2 + q_3) + l_4 \sin(q_2 + q_3 + q_4) \\ \frac{\partial t_4^y(q_2, q_3, q_4)}{\partial q_4} &= l_4 \sin(q_2 + q_3 + q_4) \end{aligned}$$

Preliminary testing of the backpropagation algorithm on highly nonlinear functions (Chester, 1991; Lee and Bekey, 1991) revealed that an architecture with two hidden layers shows better performance than one with only one hidden layer.

Experiments with the algorithm have also shown it to be advantageous to use a decoupled network architecture. In such a network, the hidden units are divided into disjoint subsets, each subset connecting to only one output neuron. For large networks, this would also allow each subnet to be trained separately, thus reducing the total training times.

Comparing the equations for direct and inverse kinematics, one can immediately see the similarity. We therefore start our investigations on the mathematically "easier" problem of direct kinematics, since the results obtained there can be extended to the "harder" problem of inverse kinematics. In the neural network approach, the only difference lies in the size of the network needed to represent the solution.

The three topologies on which we focus in the present experiment have 8, 10, and 12 hidden neurons, respectively, for each output neuron, arranged into two hidden layers of 4, 5, and 6 neurons each. Figure 5.2 shows the topology of the decoupled network with 10 hidden units per output neuron. The behavior of the network with only 8 hidden units per output neuron proved to have inferior behavior to the other two networks (both in the training and the testing phase). To study the effect that the size of the training set has on the performance of the network, we trained each network topology with two different pattern sets.

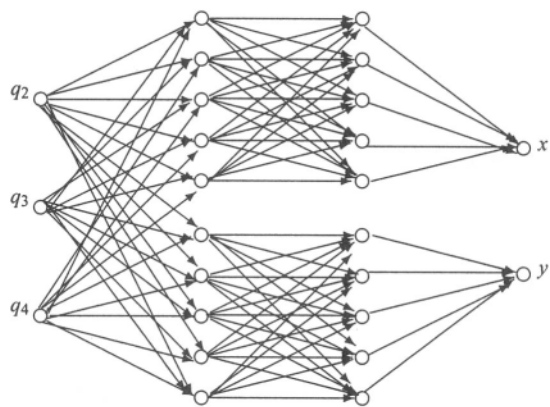


Figure 5.2. The topology of a decoupled network.

For the first set, seven equidistant sample points were chosen in the joint range interval $[\pi/2, 3\pi/2]$. For three joints, this results in a total of $7^3 = 343$ patterns. Likewise, one can also pick 13 equidistant points in $[\pi/2, 3\pi/2]$, obtaining a larger pattern set with $13^3 = 2197$ elements. In the remainder of this section, these two pattern sets will be referred to as “small” and “regular,” respectively.

Since the network should not only display acceptable behavior (i.e., small errors) on the patterns it has used for training, but also interpolate reasonably well between those patterns, a network is generally tested with a pattern set that is different from the one used in training. To this end, 1000 patterns were created, of which the input components were chosen randomly in $[\pi/2, 3\pi/2]$.

It should be kept in mind that networks which use the logistic activation function in all neurons can only produce output values in the range between 0 and 1. Therefore, the pattern sets were all scaled to this range.

The standard backpropagation algorithm was used to train the networks. Table 5.1 displays the result of training the two network topologies with the two pattern sets. The error values in the table are the average Euclidean distances between desired and actual outputs.

The rows in this table are to be read as follows: In the rows labeled “Training,” the average error of the network on the pattern set with which it was trained is given. In the rows labeled “Testing,” the results of testing the networks with a randomly generated pattern set are displayed. The labels “small” and “regular” indicate which pattern set was used to train the network. The table shows how the behavior of the network gets better (i.e., the average error decreases) as training times increase.

A surprising, albeit satisfactory result is that the network seems to interpolate

Table 5.1. Errors for Various Combinations of Topologies and Training Data

	After 150,000 cycles		After 250,000 cycles		After 450,000 cycles	
	10 neurons	12 neurons	10 neurons	12 neurons	10 neurons	12 neurons
Training						
Small	0.01266	0.01461	0.01090	0.01185	0.01059	0.01032
Regular	0.01309	0.01213	0.00903	0.00819	0.00905	0.007548
Testing						
Small	0.01242	0.01313	0.01070	0.01049	0.01047	0.00907
Regular	0.01236	0.01134	0.00834	0.00777	0.00830	0.00712

very well, as the average errors for the random testing set are even smaller than the errors for the training sets. It can also be seen that the “capacity” of the network with only 10 hidden neurons per output unit is already reached after 250,000 epochs. Further training for this net did not result in a noticeable decrease in the error. On the other hand, the network with 12 hidden neurons per output unit still shows an improvement of about 10% in the last 200,000 training cycles. These training runs also verified the expectation that error size is dependent on the size of the training set. It can be seen that networks which used the larger training set have much better behavior than those that used the smaller set.

JACOBIAN MATRIX CALCULATIONS BASED ON A SIGMOID NEURAL NETWORK: The Jacobian matrix of a kinematic equation is necessary to compute the inverse of the kinematics. For this reason we analyze briefly the possibilities for the neural modeling of the Jacobian components. The functions that appear in the Jacobian matrix of the end effector are very similar in nature to the equations that describe the forward kinematics.

Example 5.1.2. For Jacobian network training we fixed a topology—using 2×5 hidden neurons for each output neuron—and limited our attention to comparing how the average error grew for a larger network. The problem scaled up very well, as Table 5.2 shows. In this table, the third column from Table 5.1 is compared with the average error of a network calculating the Jacobian matrix of the end effector. Both networks use 10 hidden neurons per output unit and were trained for 250,000 cycles.

Considering that there are three times as many output neurons in the Jacobian network as in the direct kinematic network, the results are satisfactory in the sense that the average error does indeed grow less than linearly. For the Euclidean distance between desired and actual outputs as error measure, it can easily be seen that the theoretical growth of the error is the square root of the number of output neurons.

SINUSOIDAL NEURAL NETWORK FOR DIRECT KINEMATIC MODELING: It can be seen from the equations describing the position of the end effector of a manipulator that in general the position of a robot manipulator can be described by trigonometric

Table 5.2. Error Comparison between Direct and Inverse Kinematic Network

	Direct kinematic network	Jacobian matrix network
Training		
Small	0.01090	0.01870
Regular	0.00903	0.01611
Testing		
Small	0.01070	0.01755
Regular	0.00834	0.01554

functions. In the case of a two-dimensional manipulator, the position is determined by the weighted sum of such functions. For three-dimensional manipulators, the equations can involve multiplication of trigonometric functions as well.

We can use a multilayer feedforward neural network with hidden units having sinusoidal activation functions (Kung and Hwang, 1989; Lee and Bekey, 1991; Lee and Kil, 1989; Lee and Kil, 1990). To reduce the training time we can apply hybrid techniques which automatically create the full network topology and values of the neural weights based on symbolic computation of forward kinematic model components.

Each component t_i of an output function g can be represented in the form of

$$\prod_{i=1}^n a_i \cdot sc(q_i) \quad (5.11)$$

where $sc(q_i)$ is either $\sin q_i$, $\cos q_i$, or 1

Then, after simplification, the output function g representing the forward kinematics of a robot manipulator with n rotary joints can be described by the weighted sum of sinusoidal functions

$$t_i^s(q) = g_i^s(q) = \sum_{j \in I} l_j^s \sin(\mathbf{w}_j^T \mathbf{q}) \quad \text{for } i = 1, \dots, n \quad \text{and } s = x, y, z \quad (5.12)$$

where $g_i^s(q) = \mathbf{c} \mathbf{r} d^s \mathbf{P}_i$ defines the (i, s) output representing the s th Cartesian variable of i th joint position; q is the joint state vector; and $\mathbf{w}_j^s = (\mathbf{w}_{j1}^s, \dots, \mathbf{w}_{jn}^s)^T$ represents the weight vector of the j th sinusoidal function with $\mathbf{w}_{ji}^s = \{-1, 0, 1\}$.

The parameter I is understood to be large enough that the sum can accommodate all summands for each i .

If the manipulator has prismatic joints, some of the q_i will have to be treated as constants, with some l_j^s taking their place as variables.

From this description, it can be seen that a network with one hidden layer of sinusoidal units can implement such a function. There are two restrictions that have to be kept in mind:

First, the size of the net needed to implement the functions in Equation (5.12) depends on the complexity of the manipulator, and cannot be determined beforehand.

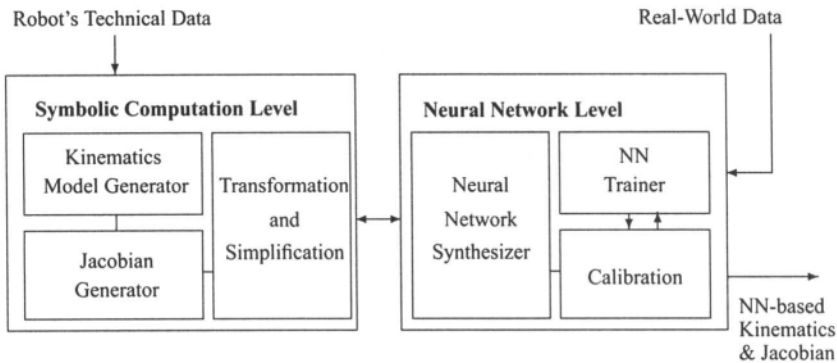


Figure 5.3. Symbolic computation system for neural kinematics and Jacobian model design.

Second, the weights l_i^s need to be calculated (the vectors w_i^s can be determined directly and are not changed).

There are two possibilities to deal with the task of obtaining the appropriate network topology for such a network: One is to start with the largest possible network, learn all the l_i^s values, and then “prune” those parts of the network which do not contribute to the output, i.e., cut those sinusoidal units that have links of weight zero connecting them to the output. The maximal number of sinusoidal units for a manipulator with n rotary joints is 3^n , as there are 3^n possibilities to weight the inputs q_1, \dots, q_n with values in $\{-1, 0, 1\}$. The other possibility is to use an algorithm that starts with a small number of sinusoidal units and dynamically creates more as they are needed.

As a another method, we can use symbolic computation methods to create the neural networks implementing the kinematic models. The structure of the system for neural kinematics and Jacobian model design is presented in Figure 5.3. The system consists of six elements grouped into two levels. The symbolic computation level consists of the following modules:

- the procedure for computation of the direct kinematics
- the procedure for computation of the Jacobian matrix
- the set of simplification and expansion rules

The neural network level is made up of

- the neural network synthesizer
- the neural network trainer
- the calibration module

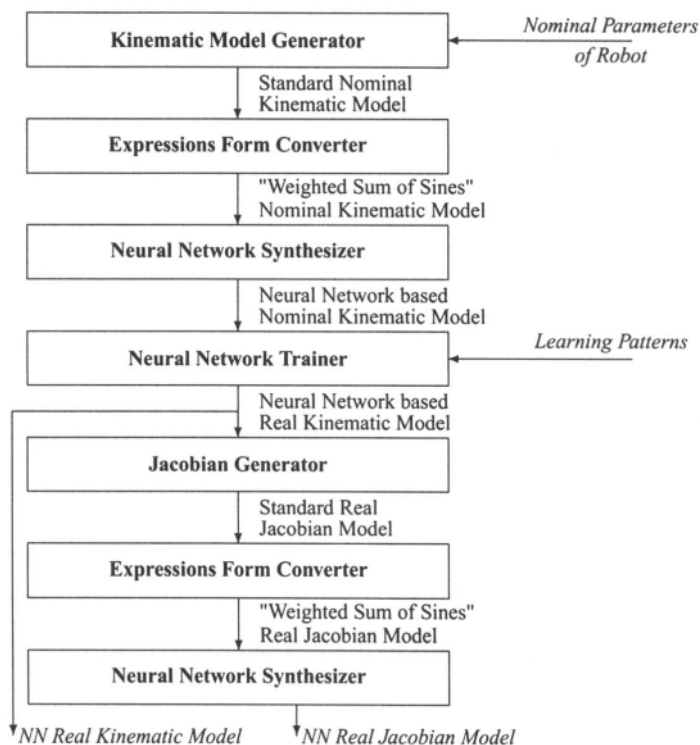


Figure 5.4. Generation of the real Jacobian and the real kinematic models.

The input to the symbolic computation level is the set of Denavit–Hartenberg parameters $\{\theta_i, d_i, a_i, \alpha_i | i = 1, \dots, n\}$ (Paul, 1981), which describe the robot's geometry. The symbolic output from this level goes to the neural network level, where the neural network implementation of the kinematic and the Jacobian matrices is generated and the calibration process is performed.

Figure 5.4 shows how the parameters of a manipulator are transformed step by step into the neural kinematic and Jacobian models. The procedure is started by calculating the standard nominal kinematic matrix (by the kinematic model generator). The matrix entries are then transformed into a weighted sum-of-sines format which can directly be implemented as a neural network. This implementation is done in the neural network synthesizer. Next, the neural model is trained, using some measurement data, to implement the actual kinematics. The trained kinematic matrix is now expressed in *Cartesian and RPY* coordinates (Paul, 1981). Based on this model, the Jacobian generator computes the Jacobian matrix, which is also expressed in *Cartesian and RPY* coordinates. In the next step, this matrix is transformed into a weighted sum-of-sines format and implemented as a neural

network.

Example 5.1.3 (*Symbolic computation-based neural model of the kinematics*). Recall that the whole model of the robot kinematics T_n can be calculated as (Paul, 1981)

$$T_n = A_1 \cdot A_2 \cdot \dots \cdot A_n \quad (5.13)$$

where

$$T_n = \begin{bmatrix} r_1 & r_2 & r_3 & t_n^x \\ r_4 & r_5 & r_6 & t_n^y \\ r_7 & r_8 & r_9 & t_n^z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$A_i = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \cos \alpha_i & \sin \theta_i \sin \alpha_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \theta_i \cos \alpha_i & -\cos \theta_i \sin \alpha_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As the example manipulator we use a rotary planar manipulator with four joints. For such a manipulator, the kinematic matrix generated by the kinematic model generator has the form

$$\begin{bmatrix} c_1 A(q_2, q_3, q_4) & c_1 B(q_2, q_3, q_4) & s_1 & c_1 (0.3s_2 - 0.6(s_2c_3 + c_2s_3) + 0.2A(q_2, q_3, q_4)) \\ s_1 A(q_2, q_3, q_4) & s_1 B(q_2, q_3, q_4) & -c_1 & s_1 (0.3s_2 - 0.6(s_2c_3 + c_2s_3) + 0.2A(q_2, q_3, q_4)) \\ -B(q_2, q_3, q_4) & A(q_2, q_3, q_4) & 0 & 0.7 - (0.3c_2 + 0.6(c_2c_3 - s_2s_3) - 0.2B(q_2, q_3, q_4)) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$A(q_2, q_3, q_4) = c_2s_3c_4 + c_2c_3s_4 + s_2c_3c_4 - s_2s_3s_4$$

$$B(q_2, q_3, q_4) = c_2c_3c_4 - s_2s_3c_4 - s_2c_3s_4 - c_2s_3s_4$$

and $s_i = \sin(q_i)$, $c_i = \cos(q_i)$, for $i = 1, \dots, 4$.

Note that this model takes into account only the nonzero parameters of the manipulator.

In general, multiplications of sines and cosines can occur in the kinematic matrix obtained from the kinematic model generator. In order to allow a neural network implementation using only linear and sinusoidal neurons, we have to transform multiplicative expressions into the weighted sum-of-sines format. Here,

sums of input angles are used as arguments to the sine functions. More precisely, the ik th element $f_{ik}(q)$ ($f_{ik} = t_i^k$ or r_i^k) of the kinematic matrix T_n is transformed into the weighted sum-of-sines format

$$f_{ik}(q) = \sum_{j \in I} l_j^{ik} \sin(\mathbf{w}_j^{ikT} \mathbf{q} + b_j^{ik}) + a^{ik} \quad (5.14)$$

Notice the similarity of this equation to Equation (5.12). Here we also use \mathbf{w}_j^{ik} as the weight vector of the joint angles and l_j^{ik} as the weight vector of the sines; however, we use explicit biases b_j^{ik} and a^{ik} , whereas Equation (5.12) used an augmented weight vector \mathbf{w}_j^{ik} . The difference is negligible when calibrating a neural implementation of $f_{ik}(q)$, but Equation (5.14) allows an exact representation of the symbolic expression returned by the transformation module. Again, the parameter I is understood to be large enough that the sum can accommodate all summands for each ik .

Using the neural representation of such a sum, the calibration can be performed by additional training for relearning the weights in the network. Using trigonometric identities, one can write

$$\begin{aligned} \sin(x) \sin(y) &= \frac{1}{2} (\cos(x-y) - \cos(x+y)) \\ \sin(x) \cos(y) &= \frac{1}{2} (\sin(x-y) + \sin(x+y)) \end{aligned}$$

These expressions are used as the transformation rules in the expression form converter to obtain the required format of the kinematic elements.

In the presented example the 4×4 kinematic matrix for the 4-DOF manipulator takes the form

$$\begin{bmatrix} 0.5s_{1+2+3+4} - & 0.5c_{1+2+3+4} + & s_1 & 0.15s_{1+2} - 0.15s_{1-2} + \\ 0.5s_{1-2-3-4} & 0.5c_{1-2-3-4} & & 0.3s_{1-2-3} - 0.3s_{1+2+3} + \\ & & & 0.1s_{1+2+3+4} - 0.1s_{1-2-3-4} \\ 0.5c_{1-2-3-4} - & 0.5s_{1+2+3+4} + & -c_1 & 0.15c_{1-2} - 0.15c_{1+2} + \\ 0.5c_{1+2+3+4} & 0.5s_{1-2-3-4} & & 0.3c_{1+2+3} - 0.3c_{1-2-3} + \\ & & & 0.1c_{1-2-3-4} - 0.1c_{1+2+3+4} \\ -c_{2+3+4} & s_{2+3+4} & 0 & 0.7 - 0.3c_2 + 0.6c_{2+3} + \\ & & & 0.2c_{2+3+4} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.15)$$

with

$$\begin{aligned} s_{i+j+k+l} &= \sin(q_i + q_j + q_k + q_l), & c_{i+j+k+l} &= \sin\left(q_i + q_j + q_k + q_l + \frac{\pi}{2}\right) \\ s_{i-j-k-l} &= \sin(q_i - q_j - q_k - q_l), & c_{i-j-k-l} &= \sin\left(q_i - q_j - q_k - q_l + \frac{\pi}{2}\right) \end{aligned}$$

for $i, j, k, l = 1, \dots, 4$.

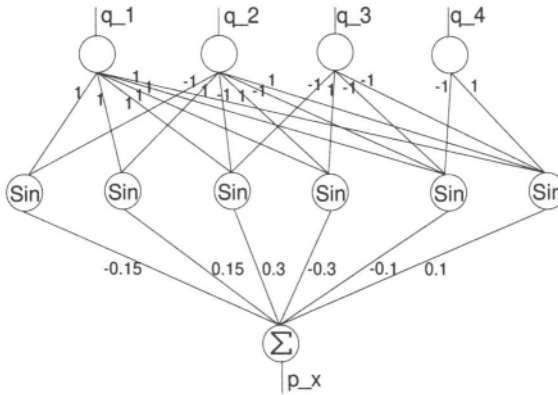


Figure 5.5. Neural network designed for Equation (5.16).

The x coordinate of the translation vector t_n (the last column of T_n) is given by

$$t_4^x = 0.15s_{1+2} - 0.15s_{1-2} + 0.3s_{1-2-3} - 0.3s_{1+2+3} + 0.1s_{1+2+3+4} - 0.1s_{1-2-3-4} \quad (5.16)$$

with $s_{i+j+k+l}$ and $s_{i-j-k-l}$ as defined in Equation (5.15).

This is the form of the matrix entry when only nonzero nominal elements are considered. To compare the effects of calibration for different complexities of networks, two neural networks are investigated. One was constructed for the Equation (5.16) and the other for the equation

$$\bar{t}_4^x = 0s_1 + 0c_1 + t_x \quad (5.17)$$

where all length parameters of the manipulator are taken into account. The topologies of these neural networks are shown in Figures 5.5 and 5.6. For the RPY-angles, the resulting network has a slightly different form.

CALIBRATION OF NEURAL NETWORK KINEMATIC MODEL: Robot calibration is a necessary step in realizing a true on-line robot programming environment on a modern factory floor. Calibration is a process by which the accuracy of the manipulator is improved by modifying its control software. The model of kinematics encoded in a typical controller software refers to some standard geometry of the manipulator's mechanisms. Such a model is usually called *nominal*. Due to inevitable deviations of the kinematics from the standard mentioned above (caused by link parameter errors, clearances in the mechanism's connections, thermal effects, flexibility of the links and the gear train, gear backlash, and encoder resolution errors), the actual kinematics may differ from the nominal kinematics.

The problem of recovering the actual kinematics on the basis of the nominal kinematics and some measurements is referred to in the robotic literature as the *kinematic calibration problem*.

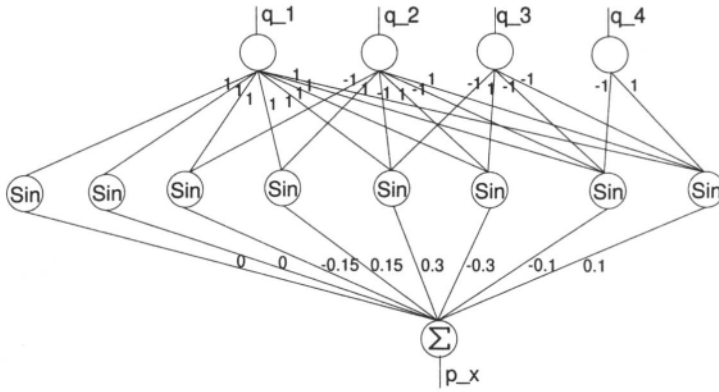


Figure 5.6. Neural network designed for Equation (5.17).

Since the network implementing the nominal robot kinematics is a network with one hidden layer of sinusoidal units, it is possible to train either only the set of weights connecting the hidden to the output layer, or the weights between the input and hidden layer as well.

The former approach is considerably simpler than the latter, as it only involves training of the outermost layer of the network. For such a task, the *delta rule* (Yeung and Bekey, 1989) learning algorithm which minimizes the mean square distance between actual and nominal kinematics is sufficient. In mathematical formulation, the update rule for the weights l_j^{ik} can be stated as

$$l_j^{ik(\text{new})} = l_j^{ik(\text{old})} + \eta(f_{ik}(q) - \bar{f}_{ik}) \sin(\mathbf{w}_j^{ikT} \mathbf{q})$$

with η the step size (usually between 0 and 1) and \bar{f}_{ik} the ik th actual entry in the matrix of kinematic values. The meaning of the other symbols is as in Equation (5.14). This update process sequentially changes the l_j^{ik} such that the difference between the \bar{f}_{ik} and f_{ik} tends to zero.

Example 5.1.4. Now we describe experiments for two types of neural kinematic models to illustrate the importance of selecting the proper model.

In the first experiment we take into account only nonzero manipulator parameters (the *simple model*), and in the second one (the *extended model*) we consider all length parameters of the manipulator.

We present the results of applying the calibration procedure to the neural model of the x component of the translation vector \mathbf{t}_n . For this purpose two output data types are compared: the location ℓ_i^{net} predicted by the neural model, and the actual location ℓ_i^{act} .

The equations describing the x component of the translation vector \mathbf{t}_n in

Table 5.3. Calibration Results

Error source ^a	Parameter (mm)	Without calibration	All weights learning	
			simple model	extended model
000001	ϵ_{\max}	8.24	0.00580	0.00670
	ϵ_{mean}	2.45	0.00175	0.00185
000011	ϵ_{\max}	21.7	2.56	0.0942
	ϵ_{mean}	11.1	0.72	0.0197
000101	ϵ_{\max}	22.2	0.0812	0.880
	ϵ_{mean}	5.6	0.0246	0.220
001001	ϵ_{\max}	22.8	0.0750	0.740
	ϵ_{mean}	3.3	0.0249	0.185
010001	ϵ_{\max}	16.2	0.00670	0.00569
	ϵ_{mean}	8.8	0.00150	0.00164
100001	ϵ_{\max}	8.24	0.379	0.173
	ϵ_{mean}	2.45	0.096	0.061

^aSee text for code.

the kinematic matrix for the simple and the extended model are given in Equations (5.16) and (5.17), respectively.

The topologies of the networks for the two types of models for \mathbf{r}_4^x and \mathbf{r}_4^y are shown in Figures 5.5 and 5.6. The results in Table 5.3 show improvements in mean error and maximum error when various error sources are present. The first column in the table defines the combination of error sources by using the following binary coding:

errors in the link lengths	000001
errors in the link offsets	000010
errors in the joint offsets	000100
errors in the transducers' convergence rates	001000
errors in the robot's base system	010000
noise in the location measurements	100000

The absolute error ϵ_{\max} is defined as

$$\epsilon_{\max} = \max_i |\ell_i^{\text{net}} - \ell_i^{\text{act}}|$$

Similarly, the mean absolute error ϵ_{mean} is given by

$$\epsilon_{\text{mean}} = \frac{1}{\text{Num}} \sum_{i=1}^{\text{Num}} |\ell_i^{\text{net}} - \ell_i^{\text{act}}|$$

with Num being the number of control points.

In these experiments 1000 points from a uniform grid in the workspace of the manipulator were taken as the control points for testing the calibration results. One

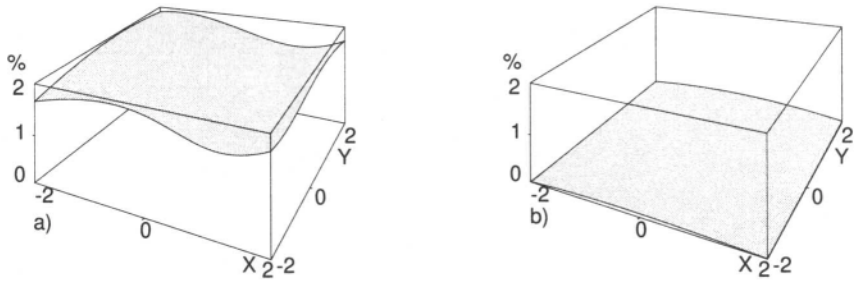


Figure 5.7. The mean error of effector position in the XY plane for (a) noncalibrated and (b) calibrated neural-based robot kinematics.

hundred points, also from the uniform grid, were used as the learning patterns. The number of learning cycles was 1000. The mean error of end-effector position in XY plane is shown in Figure 5.7.

In each case it was possible to reduce the maximal error after calibration to the level of 10^{-2} mm.

To compare the calibration method presented above with results obtained by calibration based on identification methods, let us recall some other researchers' results. Wu and Young (1993) proposed to use a robot accuracy compensator. With this compensator applied to a PUMA 560 manipulator they reduced the PUMA's position errors from 7.1 mm to 0.4 mm. The theoretical value of the position error they obtained was 0.2 mm.

Mooring *et al.* (1991) used two different manipulator models (modified Denavit–Hartenberg model and zero-reference-position model) for a PUMA 560 robot and reduced the position error from 43 mm to 0.8 mm.

Borm and Menq (1991) pointed out the importance of determining the optimal measurement configuration. Using an iterative least square based estimation algorithm with a set of optimal measured positions for an RM-501 robot, they reduced the position errors from 28 mm to 0.6 mm.

Duelen and Schröer (1991) presented a calibration procedure which allows the automatic determination of all kinematic parameters, i.e., those of geometric origin as well as nongeometric ones. For a SCARA robot, their method reduced the mean position error from 5.7 mm to 0.4 mm.

As we can see, the methods mentioned above reduce the position errors of manipulators to tenths of a millimeter. The accuracy of our method seems to be limited only by measurement equipment precision.

APPLYING THE CALIBRATED KINEMATIC MODEL TO NEURAL JACOBIAN CALCULATION: Similarly as in the previous section, we analyze now the possibility of modeling the Jacobian matrix with sinusoidal neuron networks. One well-known method for calculating the Jacobian matrix of a robot's manipulator is described by Spong and Vidyasagar (1989). This method calculates the Jacobian matrix

without taking derivatives of the kinematic equations. However, the algorithm requires all partial kinematic matrices of the manipulator, described in terms of Denavit–Hartenberg parameters, to be known beforehand. The Jacobian obtained from this algorithm describes the dependence

$$J\dot{q} = \begin{bmatrix} v \\ \omega \end{bmatrix}$$

where v is the vector of linear velocities and ω the vector of angular velocities.

For the linear velocities it is not difficult to find the coordinate frame, as they are described in the base coordinate frame. It is, however, impossible to do so for the angular velocities, because there does not exist an orientation vector whose first derivatives are equal to these velocities.

Due to this fact, we can express the Jacobian matrix in *Cartesian RPY* coordinates. Although there exists a method that directly calculates the Jacobian matrix in these coordinates, we will calculate it using a method that yields a form that is more useful for neural network implementation.

Each of the elements t_n^x , t_n^y , and t_n^z of the translation part of the kinematic matrix has a form given by Equation (5.14). The partial derivative of t_n^x with respect to q_i is described by

$$\frac{\partial t_n^x}{\partial q_i} = \sum_{j=1}^I l_j^x w_{ji}^x \cos((w_j^x)^T q + b_j^x) \quad (5.18)$$

where w_{ji}^x is the i th component of the weight vector w_j^x associated with t_n^x .

The partial derivatives of t_n^y and t_n^z are similar to that for t_n^x , as only the weights and bias need to be changed.

It is easy to observe that the structure of a neural network for such elements of Jacobian matrix is the same as for the corresponding kinematic matrix elements. The difference lies only in the parameters of the net.

For the *RPY* (θ, φ, ψ) angles we obtain

$$\begin{aligned} \frac{\partial \varphi}{\partial q_j} &= \frac{1}{\cos^2 \theta} \left(\frac{\partial r_4}{\partial q_j} r_1 - \frac{\partial r_1}{\partial q_j} r_4 \right) \\ \frac{\partial \theta}{\partial q_j} &= -\frac{1}{|\cos \theta|} \frac{\partial r_7}{\partial q_j} \\ \frac{\partial \psi}{\partial q_j} &= \frac{1}{\cos^2 \theta} \left(\frac{\partial r_8}{\partial q_j} r_9 - \frac{\partial r_9}{\partial q_j} r_8 \right) \end{aligned}$$

where r_i are the elements of the rotation part of the kinematic matrix T_n . Note that only the five elements r_1, r_4, r_7, r_8 , and r_9 from the rotation part of the matrix T_n [Equation (5.13)] are used.

In order to show the weighted sum-of-sines format of the above derivatives, we can use the corresponding format of the rotation elements \mathbf{r}_k from our example of the manipulator (Example 5.1.3), to write, e.g.,

$$\begin{aligned} \frac{\partial \varphi}{\partial q_j} = \frac{1}{\cos^2 \theta} & \left(a_1 \sum_{r=1}^I l_{4r} w_{4rj} \sin(\mathbf{w}_{4r}^T \mathbf{q} + b_{4r} + \pi/2) \right. \\ & + a_4 \sum_{r=1}^I l_{1r} w_{1rj} \sin(\mathbf{w}_{1r}^T \mathbf{q} + b_{1r} + \pi/2) \\ & + \frac{1}{2} \sum_{r,s=1}^I l_{1s} l_{4r} (w_{4rj} + w_{1sj}) \sin((\mathbf{w}_{1s} - \mathbf{w}_{4r})^T \mathbf{q} + b_{1s} - b_{4r}) \\ & \left. + \frac{1}{2} \sum_{r,s=1}^I l_{1s} l_{4r} (w_{4rj} - w_{1sj}) \sin((\mathbf{w}_{1s} + \mathbf{w}_{4r})^T \mathbf{q} + b_{1s} + b_{4r}) \right) \quad (5.19) \end{aligned}$$

The scalar factor $1/\cos^2 \theta$ cannot be implemented by the neural network and needs to be calculated with the help of additional parallel hardware.

Repeating this calculation for the derivatives of ψ leads to an expression of the same form, but with different indexes. The derivative of θ is similar to the right-hand side of Equation (5.18), again with appropriate weights and bias. It can be seen that the topologies of the neural networks implementing the denominators of these derivatives are much more complicated than those implementing the corresponding elements of the kinematic matrix. Given the form of the above expressions, it can be seen that they are already in the weighted sum of sines format and need no further transformation.

The overall structure of the resulting system that calculates the Jacobian matrix elements in *RPY Cartesian* coordinates is shown in Figure 5.8. It can be seen in Equations (5.18) and (5.19) how the weights \mathbf{w}_j^k and \mathbf{l}_j^k of the Jacobian matrix elements can be calculated from the corresponding weights and biases of the kinematic networks.

The discrete model of robot kinematics based on neural network computation is shown in Figure 5.9. Such a form, obtained from the symbolic level of network generation, represents the input–output function of a neural network with sinusoidal hidden units.

5.1.2.2. Finite-state machine model of robot kinematics. In order to apply fast methods of geometrical path planning, the robot kinematics model should facilitate the direct analysis of robot location with respect to objects in its environment modeled in Cartesian space. The discrete model of a robot's kinematics with n degrees of freedom can be simplified when the robot has a planar manipulator, i.e., all the joints lie on a plane which can rotate around the Z axis (Jacak, 1989b; Jacak, 1991).

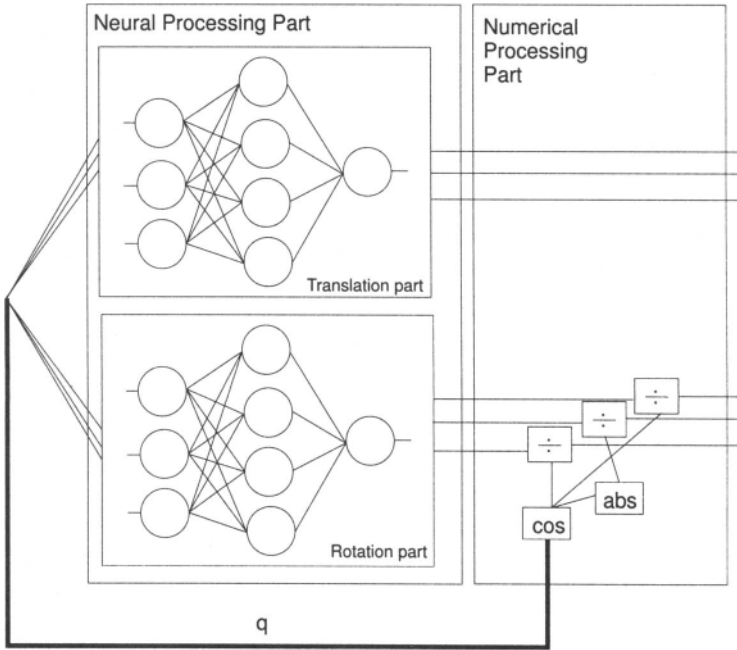


Figure 5.8. Neural and numerical processing system for Jacobian calculation.

In this case the model of robot kinematics is reduced to the following finite state machine:

$$M = (Y, U, f) \quad (5.20)$$

where:

- Y denotes the set of robot configurations in the Cartesian base frame, i.e., the configuration

$$y = (P_i | i = 0, 1, \dots, n)^T \in Y \quad (5.21)$$

where $P_i = (x_i, y_i, z_i) \in E_0$ is the point in the Cartesian base frame E_0 describing the actual position of the i th joint. For such a specification the state of the robot model is equal to the skeleton of the manipulator and the output function g can be omitted.

- U is the input signal set. Recall that in order to specify the set U the discretization of the robot's joint space Q is performed [Equation (5.6)]. In this case the input signal set $U = \{-1, 0, +1\}^n$ and $u_i = \pm 1$ denotes the **δq_i -increment** increase (decrease) of the appropriate joint angle q_i (Jacak, 1991).

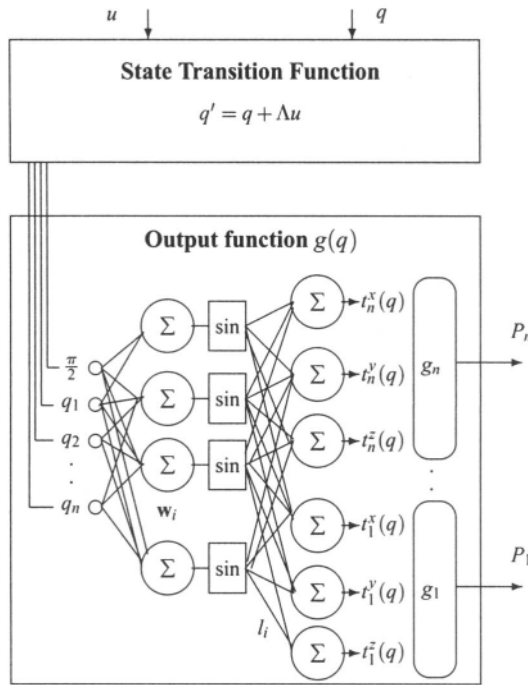


Figure 5.9. Neural network-based model of robot kinematics.

- $f : Y \times U \rightarrow Y$ is the one-step state transition function of a finite-state machine of the form

$$y(k+1) = f(y(k), u(k)) \quad (5.22)$$

and is defined as follows (Jacak, 1991):

Let $y' = f(y, u)$, where $y' = (P'_i | i = 0, 1, \dots, n)$ and $y = (P_i | i = 0, 1, \dots, n)$; then

$$P'_i = \begin{bmatrix} R_1(u_1) & 0 \\ 0 & 1 \end{bmatrix} \cdot \mathcal{H}^{-1}(s'_i) \quad (5.23)$$

where $s_i = \mathcal{H}(P_i) = (||P_i||_{xy}, z_i)$ is the transformation between the base frame and the manipulator arm plane.

The component s'_i is recursively calculated as follows:

$$s'_i = s'_{i-1} + \prod_{k=2}^{i-1} R_k(u_k) \cdot [\mathcal{H}(P_i) - \mathcal{H}(P_{i-1})] \quad (5.24)$$

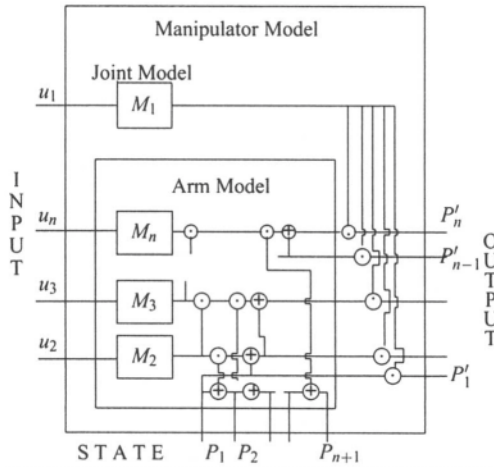


Figure 5.10. Finite-state machine model of robot kinematics.

and each $R_i(u_i)$ is a constant matrix,

$$R(u_i) = \begin{bmatrix} \cos \delta q_i & u_i \sin \delta q_i \\ -u_i \sin \delta q_i & \cos \delta q_i \end{bmatrix} \quad (5.25)$$

The construction of the transition function f of the robot kinematics model allows us to obtain, by simple computations, successive configurations of the robot with respect to the Cartesian base frame. The complete formal explanation of the FSM model of robot kinematics, shown in Figure 5.10, is presented in (Jacak, 1989b; Jacak, 1991).

5.1.2.3. Searching strategies for collision-free robot movements. The problem of collision-free robot movement planning based on a discrete model M amounts to finding a sequence

$$u_K^* = (u(1), \dots, u(K)) \quad (5.26)$$

of input signal vectors such that:

- a. The terminal configuration reaches the effector's final position P_{Final} , i.e.,

$$f^*(q(0), u_K^*) = q(K) \quad \text{and} \quad g_n(q(K)) = t_n(q(K)) = P_{\text{Final}} \quad (5.27)$$

where $q(0) = q_{\text{init}}$ and $q(K) = q_{\text{final}}$.

- b. Every configuration $q(i)$ for $i = 1, \dots, K$ does not collide with any obstacle in the robot's environment.

c. The sequence u_k^* should minimize the motion-cost function

$$v : U^* \rightarrow \mathfrak{R}$$

expressing, e.g., the length of the effector path.

In order to solve the path planning problem we apply a *graph searching* procedure to the manipulator's state transition graph. The development of the search graph will start from the node (configuration) $q(0)$ by the action f for all possible input signals from the set U . Then the successor set of the node $q(k)$ is as follows:

$$\text{Succ}(q(k)) = \{f(q(k), u) | u \in U\} \quad (5.28)$$

Every node of the graph has $3^n - 1$ successors. Thus, it becomes essential to check rapidly for the nonrepeatability of the nodes generated and their feasibility.

Configuration feasibility testing. A configuration q is said to be feasible if does not collide with any obstacle in the work scene. The space occupied by the robot manipulator at configuration q can be approximated by rotary cylinders whose axes are individual links of the skeleton model. In order to check, using only its skeleton model, whether the manipulator moves in a collision-free way, we extend the objects in each direction by the value of the maximal radius of the cylinders which approximate the links.

To obtain fast and fully computerized methods for collision detection, we use additional geometric representations of each object on the scene. We introduce the ellipsoidal representation of 3D objects [Equation (3.16)], which uses ellipsoids for filling the volume. The ellipsoidal representation of a virtual object is convenient to test the feasibility of robot configurations. Checking for the collision-freeness of the robot configuration can be reduced to the "*broken line-ellipsoid*" *intersection detection problem*, which in this case has easy analytical solution (Fact 3.3.1). Let $O_i \subset E$, for $i = 1, \dots, W$, denote an obstacle in the scene. Taking into account the above statements, we define the set of feasible configurations as

$$\text{Succ}'(q) = \{q \in \text{Succ}(q) | \text{Seg}(q) \cap \bigcup_{i=1}^W O_i = \emptyset\}$$

where $\text{Seg}(q)$ is a broken line joining points P_0, P_1, \dots, P_n .

Search technique for the track of the motion. In order to solve the reachability problem of the terminal configuration, we apply a graph search procedure to the state-transition graph of the model M . This procedure generates (makes explicit) part of an implicit specified graph. For this purpose we exploit the state-transition graph of model M generated implicitly by applying the function f . The process of applying the transition function to the start node (the initial configuration) we call expanding the node. Expanding q_{init} , successors of q_{init} , etc., ad infinitum makes

explicit the graph that is implicitly defined by q_{init} and the function f . The graph search control strategy is a process of making explicit a portion of an implicit graph sufficient to include a goal node. The way of expanding the graph will depend on the form of the cost function, the evaluation function, and the way of expanding the node. The cost of a path between two nodes is the sum of the cost of all the arcs connecting the nodes along the path. Let the function $c(q)$ give the actual cost of the path in the search tree from start node q_{init} to the node q for all q accessible from q_{init} , and the function $h(q)$ give the cost estimate of the path from node q to the goal node q_{final} . We call h the heuristic function. The evaluation function $v(q)$ at any node q gives the sum of the cost of the part from q_{init} to q plus the cost of the part from q to the goal node i.e.,

$$v(q) = c(q) + h(q)$$

A configuration q is said to be feasible if it does not collide with any object in the cell. The development of the search graph will start from the initial position, represented by the q_{init} or y_{init} configuration, by the action of function f for all possible inputs u .

To find the shortest path in the state-transition graph connecting the initial node q_{init} and the final node q_{final} (such that n th component of the final node $y_{\text{final}}^n = P_{\text{Final}}$ is), we use the A* algorithm with penalty function (Pearl, 1984; Pearl, 1988; Nilsson, 1980; Jacak, 1989b). As the cost function between two nodes q, q' joined by an arc, we shall assume the distance traveled by the effector while passing from configuration q to q' , i.e.,

$$c(q, q') = \|t_n(q) - t_n(q')\| + p(q, q') \quad (5.29)$$

or

$$c(q, q') = \sum_{i=2}^n \|t_i(q) - t_i(q')\| + p(q, q')$$

or

$$c(q, q') = \sum_{i=1}^n \alpha_i \|q_i - q'_i\| + p(q, q')$$

Let K_i denote the center of gravity of the i th obstacle, and let $q' = f(q, u)$. Any motion of the effector toward the obstacle will be penalized by increasing the value of the evaluation function at configuration q' . This is realized by adding the value of a penalty function p . The value of p will depend only on those obstacles that potentially may be found between the actual position and the goal point P_{Final} .

The set of potentially active obstacles for configuration q is defined as

$$VO(q) = \{i = 1, \dots, W \mid \|K_i - t_n(q)\| \leq \|P_{\text{Final}} - t_n(q)\|\}$$

Denote by

$$r_j(q) = [t_n(q) - K_j]$$

the vector joining the center of gravity of the j th obstacle to the actual position of the end of the effector in configuration q , and by

$$d(q, q') = [t_n(q') - t_n(q)]$$

the translation vector of the end of the effector while passing from configuration q to q' .

The component of the penalty function generated by the j th obstacle from the set $VO(q)$ upon passing from configuration q to q' can be expressed as

$$p_j(q, q') = \begin{cases} \frac{r_j(q)^T d(q, q')}{\|d(q, q')\|} & \text{if } \frac{r_j(q)^T d(q, q')}{\|d(q, q')\| \|r_j(q)\|} \geq \cos \beta_r \\ 0 & \text{otherwise} \end{cases} \quad (5.30)$$

where the angle β_r defines a cone around the direction along which the motion is performed, and the penalty function is in effect.

The penalty function for approaching the obstacles is equal to the sum of functions p_j , i.e.,

$$p(q, q') = \sum_{j \in VO(q)} p_j(q, q')$$

The heuristic function h is defined as the rectilinear distance between the current effector position and the final position P_{Final} ,

$$h(q) = \|t_n(q) - P_{\text{Final}}\|$$

Fact 5.1.1. *It is also easily seen that the heuristic function h satisfies the monotone restriction whenever $\beta_r \leq \pi/2$.*

□

□

The structure of such a path planner is shown in Figure 5.11.

Remarks. The basic advantages of a such technique lie in the possibility of expressing successive robot configurations immediately in the basic Cartesian frame by means of a computationally simple recurrent transition function of the finite-state model M or neural implementation of the manipulator's kinematics. This function can be calculated using only the operations of addition, multiplication by constant 2×2 matrices, and taking square roots. Some obvious disadvantages of this approach come from restricting the modeling only to planar robots. However, an extension of the model to nonplanar manipulators complicates only the state transition function by introducing an additional coordinate frame. In this case, the

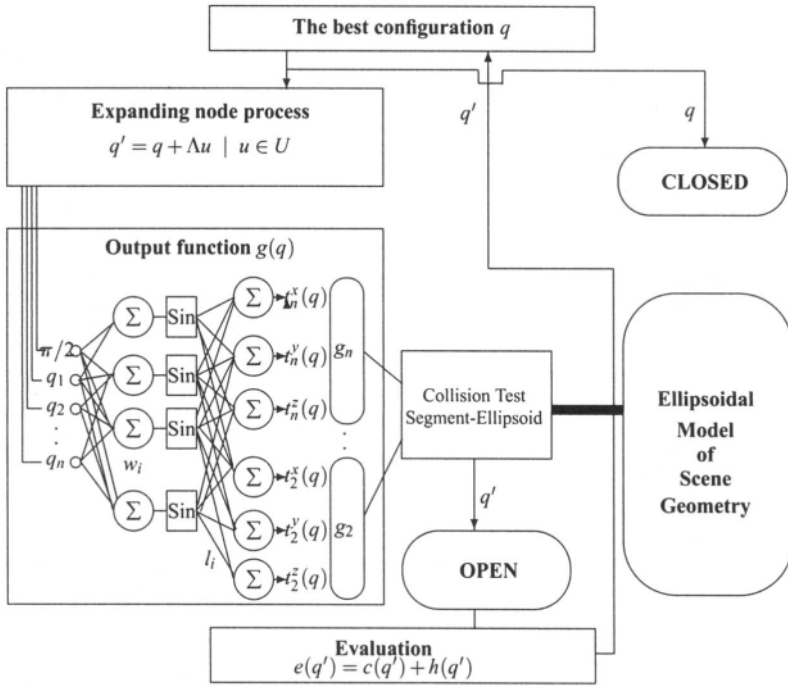


Figure 5.11. Structure of a path planner based on joint space discretization.

general model [Equation (5.1)] can be applied with the neural implementation of the output function g .

The possibility of expressing the robot's configuration directly in the Cartesian space allows one to abandon the transformation of the workspace with obstacles into the joint space. This fact becomes of particular importance from the viewpoint of effectiveness of testing solution feasibility. Checking for the collision-freeness of the configuration has been reduced to the *broken line-ellipsoid* intersection detection problem, which has a simple analytical solution (Fact 3.3.1). The technique of searching for collision-free paths has been reduced to the problem of model state graph searching. Due to the cardinality of the model's input set, this planning technique is especially applicable to robots that do not have a large number of degrees of freedom n — typically, $n < 7$. Indeed the number of discrete configurations that are the successors of the given configuration q increases exponentially with the number of degrees of freedom. Testing all would be too time consuming for robots with many degrees of freedom. As long as $n < 7$ this raises no difficulty.

However, when n becomes too big, the size of the successor set becomes too large. In this case, at each step of the expanding node process, a limited

(and relatively small) number of configurations in the successor set of the current configuration q is iteratively considered. Each iteration consists of randomly selecting the successors of q using a uniform probability distribution law. The number of configurations is limited. As the algorithm uses a random procedure to build the search graph, it is not guaranteed to find a solution whenever one exists. Such a method is only probabilistically complete.

5.1.3. Neural Network Based Path Planning in Cartesian Space

Due to the discretization of the joint space, the number of possible states of the robot arm is finite, but too large to be determined explicitly. Thus, it is very important to find a method to reduce the number of possible states. This can be done by performing the discretization of the Cartesian space. This leads to the assumption that an input signal of M has to be a generalized vector of displacement of the effector end. The location of the effector end is generally described as a vector

$$\ell = (x, y, z, \alpha, \beta, \gamma)^T \quad (5.31)$$

where $(x, y, z) = (t_n^x, t_n^y, t_n^z) = P_n$ is the position of the effector end and (α, β, γ) are the Euler angles (or RPY angles θ, φ, ψ) representing the orientation of the effector (Brady, 1986; Jacak, 1991; Sciavicco and Siciliano, 1988). By a generalized vector of the effector's displacement in the base frame we mean a vector

$$\delta\ell = (\delta x, \delta y, \delta z, \delta\alpha, \delta\beta, \delta\gamma) \quad (5.32)$$

which consists of the displacement vector of the work point and the orientation change vector. Assuming that the effector position and orientation can change only by a defined increment $\delta\ell$, we can construct the set of inputs of the model M as

$$U \subset V_x \times V_y \times V_z \times V_\alpha \times V_\beta \times V_\gamma \quad (5.33)$$

where $V_x = \{-\delta x, 0, +\delta x\}$ is the set of possible increments of the effector displacement along the X axis. Similarly, we define the sets $V_y, V_z, V_\alpha, V_\beta, V_\gamma$. The number of elements of the set U depends on the type of task and the type of discrete geometry used to discretize the base frame.

5.1.3.1. Inverse model of robot kinematics. For the above reasons the model of manipulator kinematics must be able to generate a sequence of robot configurations which realize the motion along the desired effector displacement. To determine the one-step transition function f , the inverse kinematics of the robot manipulator is needed. We can use the relationship between the joint and the Cartesian velocity.

The direct kinematic equation allows us to establish the relationship between the joint configuration q and the generalized position ℓ of the effector end:

$$\ell = t(q) = (t_n(q), \alpha, \beta, \gamma)^T \quad (5.34)$$

where t_n is the last column of the matrix T_n [Equation (5.13)]. This equation can be transformed into an equation describing the relationship between joint velocities and Cartesian velocities:

$$d\ell = J(q)dq \quad (5.35)$$

where $J(q)$ is the Jacobian matrix $[\partial t / \partial q]$. For a kinematically redundant manipulator the dimension of the joint space is greater than the dimension of the Cartesian space: $\dim(q) = n > \dim(\ell) = m$. Consequently, the Jacobian matrix $J(q)$ is rectangular, and there exists no unique inverse velocity transformation. A rectangular $n \times m$ matrix J does not have a unique inverse J^{-1} , but it is possible to find a generalized inverse J^+ so that $JJ^+ = I$. In fact, there exists an infinite number of matrices J^+ with the above property. The problem then is choosing one of these. The most commonly used generalized inverse is the pseudo-inverse, or the Moore–Penrose inverse (Klein and Huang, 1983; Nakamura and Hanafusa, 1987; Luca et al., 1991). For $n > m$ this is

$$\delta q = J^+(q)\delta\ell \quad (5.36)$$

where $J^+(q) = J(q)^T(J(q)J(q)^T)^{-1}$. A general solution can be found by adding a null vector to the specific solution

$$\delta q = J^+(q)\delta\ell + [I - J^+(q)J(q)]x \quad (5.37)$$

where $x = \text{grad}_q h(q)$ is the joint velocity vector minimizing the performance function h , and $[I - J^+(q)J(q)]x$ is its projection into the null space of J corresponding to a self-motion of the linkage that does not move the effector end. Recall that adding a null vector does not affect the Cartesian space velocity vector $d\ell$. Most of the approaches to the kinematic control of a redundant manipulator focus on resolving redundancy by applying the generalized inverse to the manipulator Jacobians based on a performance function such as the manipulability measure (Yoshikawa, 1985), condition number, manipulator-velocity ratio (Dubey and Luh, 1987), compatibility index, or minors of the Jacobian matrix (Dubey and Luh, 1987; Nakamura and Hanafusa, 1987; Chung et al., 1992). However, when a manipulator with a large number of redundant degrees of freedom is required, the pseudo-inverse approaches have great difficulty in formulating a performance function and finding its gradient vector even with the aid of symbolic calculations. It makes real-world applications unrealistic. For these reasons we will use a method which requires only the computation of direct kinematics based on neural

processing. Such a solution of inverse kinematic problems can be obtained by attaching a feedback network around a feedforward network to form a recurrent loop such that, given the desired Cartesian location ℓ_{Final} of the feedforward network, the feedback network iteratively generates joint angle correction terms to move the output of the forward network toward the given location (Kung and Hwang, 1989; Lee and Bekey, 1991). This coupled neural network is the neural implementation of the gradient method (Goldenberg et al., 1985; Chang, 1987) of position error minimization.

Let

$$e(q) = [\ell_{\text{Final}} - t(q)] \quad (5.38)$$

denote the location error between the current position and orientation of the effector-end calculated by forward kinematics and a desired Cartesian location ℓ_{Final} . Then the inverse kinematic problem can be transformed to an optimization problem with constraints as follows:

For a given Cartesian location ℓ_{Final} find the joint configuration q^* that minimizes the performance criterion

$$v(q) = \frac{1}{2} e(q)^T e(q)$$

The performance criterion v can be treated as a Lyapunov error function. Calculating the time derivative of v , we obtain

$$\frac{dv}{dt} = e^T \cdot \dot{e} = -e^T \cdot J(q) \cdot \dot{q}$$

With the choice of

$$\dot{q} = \alpha \cdot J^T(q) \cdot e = \alpha \cdot J^T(q) \cdot (\ell_{\text{Final}} - t(q)) \quad (5.39)$$

the dynamic system ensures that e converges to zero, and the time derivative of v

$$\frac{dv}{dt} = -\alpha \cdot e^T \cdot J(q) J^T(q) \cdot e$$

is negative if α is positive.

The dynamic system given by Equation (5.39) can be transformed into the following discrete dynamic system:

$$q(k+1) = q(k) + \alpha \cdot J^T(q(k)) \cdot [\ell_{\text{Final}} - t(q(k))] \quad (5.40)$$

Convergence. It is easy to observe that the above solution represents a so-called *gradient system* of the form

$$\dot{q} = -\alpha \cdot \text{grad}_q v(q)^T$$

The function $v(q)$ is positive definite and

$$\dot{v}(q) = -\alpha \cdot \|\text{grad}_q v(q)\|^2$$

For positive α such a system is stable and it can easily be shown that isolated minima are asymptotically stable (Hirsch and Smale, 1974).

The speed of convergence is strongly influenced by the parameter α . To find the best parameter we can calculate the generalized least square solution as the solution of the discrete dynamical system:

$$q(k+1) = q(k) + C(q(k))^{-1} J(q(k))^T \cdot [\ell_{\text{Final}} - t(q(k))] \quad (5.41)$$

where $C(q)$ is the normal equation matrix of the form

$$C(q(k)) = J(q(k))^T J(q(k))$$

We present three different possibilities for calculating the parameter values for α in Equation (5.40).

- A. The system (5.41) may be regarded as the system (5.40) with the parameter α taken as the inverse of the normal equation matrix, i.e.,

$$\alpha_1 = C(q(k))^{-1}$$

Such a generalized least squares procedure will often fail to converge because the errors of the initial approximations are too large. Moreover, such a system needs to calculate the inverse of matrix $C(q(k))$ for each iteration step.

- B. The system (5.40) is equivalent to the gradient method. The parameter α is generally referred to as the *convergence rate* and should be selected at each step of the iterative process to minimize

$$v(q(k+1)) = v(q(k) - \alpha_2 \cdot \text{grad}_q v(q(k))^T)$$

The solution obtained in this way is

$$\alpha_2 = \frac{\text{grad}_q v(q(k)) \cdot \text{grad}_q v(q(k))^T}{\text{grad}_q v(q(k)) \cdot H(t(q(k))) \cdot \text{grad}_q v(q(k))^T}$$

More exactly,

$$\alpha_2 = \frac{e^T \cdot J J^T \cdot e}{e^T \cdot J H J^T \cdot e}$$

where $H(t(q)) = [\partial^2 t(q) / \partial q_i \partial q_j]$ is the Hessian matrix. This choice of parameter α_2 requires the computation of the Hessian of $t(q)$, which is generally numerically intensive.

C. Let us try setting the parameter α to

$$\alpha_3 = \frac{v(q(k))}{\text{grad}_q v(q(k)) \cdot \text{grad}_q v(q(k))^T} = \frac{e^T \cdot e}{e^T \cdot J J^T \cdot e}$$

It can be shown (Lee and Bekey, 1991) that the dynamic system (5.39) with such a parameter α_3 has an equilibrium point in q^* if and only if q^* is a solution vector of the error e , i.e.,

$$\dot{q} = \alpha \cdot J^T(q^*) \cdot e = 0 \quad \text{iff} \quad e(q^*) = 0$$

The formula for updating $q(k+1)$ determined by Equation (5.40) and α_3 guarantees the convergence as long as \dot{q} exists.

Neural processing-based inverse kinematics for redundant manipulators. We apply the discrete dynamical system presented above to calculate the inverse kinematics by neural processing. The system needs only to calculate the direct kinematics of a robot manipulator. Recall that the equations for the forward kinematics of a manipulator can be reduced to a weighted sum of sines through trigonometric transformations.

More precisely, the k th element of the kinematic matrix (a function of joint angles q) is transformed into the weighted sum-of-sines format

$$f_{ik}(q) = \sum_j l_j^{ik} \sin(w_j^{ikT} q + b_j^{ik}) + a^{ik}$$

Here we use w^{ik} as the weight vector of the joint angles and l^{ik} as the weight vector of the sinusoidal units, as well as biases b^{ik} and a^{ik} . Neural networks can be used to reduce the computational complexity of the robot kinematic model. Additionally, neural networks can give robots the ability to self-calibrate in a learning-like manner.

The solution of the inverse kinematic problem can be obtained by attaching a feedback network around a feedforward network to form a recurrent loop. Given a desired Cartesian location ℓ_{Final} , the feedback network iteratively generates joint angle correction terms to move the output of the forward network toward the given location. This coupled neural network, shown in Figure 5.12, is the neural

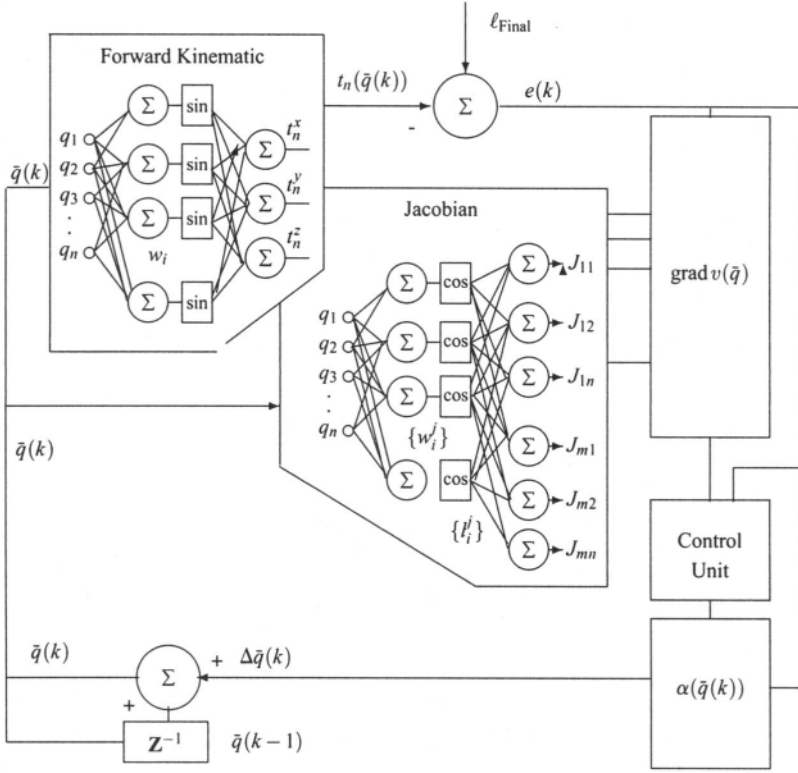


Figure 5.12. Coupled neural network for unconstrained inverse calculation.

implementation of the gradient method [Equation (5.40)] (Goldenberg et al., 1985; Chang, 1987) of position error minimization.

In the case of a redundant manipulator with joint-range constraints, the inverse kinematic problem can be transformed into an optimization problem with constraints as follows:

For a given Cartesian location ℓ_{Final} , find the joint configuration q^* that minimizes the performance criterion v ,

$$v(q^*) = \frac{1}{2} e(q)^T e(q) \rightarrow \min \quad \text{and} \quad q^{\min} \leq q^* \leq q^{\max} \quad (5.42)$$

The above problem can be transformed into a problem without constraints by introducing a penalty function ω to obtain the joint limits. To this purpose we use additional errors for each degree of freedom, i.e.,

$$e_i^{\text{m}} = q_i^{\min} - q_i \quad \text{and} \quad e_i^{\text{M}} = q_i - q_i^{\max} \quad \text{for} \quad i = 1, \dots, n \quad (5.43)$$

The penalty function is defined as

$$\omega(q) = \frac{1}{2} e_m^T \Phi_m e_m + \frac{1}{2} e_M^T \Phi_M e_M \quad (5.44)$$

where $e_m = (e_1^m, \dots, e_n^m)^T$, $e_M = (e_1^M, \dots, e_n^M)^T$, Φ_m is a diagonal matrix with entries $\varphi(e_i^m)$, and Φ_M is a diagonal matrix with entries $\varphi(e_i^M)$.

$$\varphi(x) = \frac{1}{1 + e^{-\beta x}}$$

Then the modified performance function is given by

$$v(q) = \frac{1}{2} e^T e + \frac{1}{2} e_m^T \Phi_m e_m + \frac{1}{2} e_M^T \Phi_M e_M \quad (5.45)$$

By using the sigmoid function φ in the penalty function ω with a large value of β , we activate the errors e_i^m and e_i^M only if $q_i < q_i^{\min}$ or $q_i > q_i^{\max}$, respectively.

The modified function v can be interpreted as the Lyapunov function of a dynamic system. Error dynamics is involved to ensure convergence of $t(q)$ to ℓ_{Final} . With the choice of

$$\dot{q} = \alpha \cdot \frac{e^T e + (e_m)^T \Phi_m e_m + (e_M)^T \Phi_M e_M}{\text{grad}_q v(q)^T \cdot \text{grad}_q v(q)} \text{grad}_q v(q) \quad (5.46)$$

the error e converges to zero when the constraints are satisfied.

This issue can be recognized by considering the time derivative of the Lyapunov function for which the time derivative of v is negative if α is positive. The gradient of v has the following form:

$$\text{grad}_q v(q) = -J(q)^T e - \Phi_m e_m - D(e_m) \Phi_{m'} e_m + \Phi_M e_M + D(e_M) \Phi_{M'} e_M$$

where $J(q)$ is the Jacobian matrix, $D(e_m)$ is the diagonal matrix with entries e_i^m , and $\Phi_{m'}$ is the diagonal matrix of first derivatives of $\varphi(e_i^m)$.

It can easily be seen that $\varphi' = \beta \varphi(1 - \varphi)$.

The discrete form of Equation (5.46) represents the variant of an iterative gradient method (Han and Sayeh, 1989) written as

$$q(k+1) = q(k) + c(q(k)) \cdot \text{grad}_q v(q(k)) \quad (5.47)$$

In our case

$$c(q(k)) = \alpha \cdot \frac{e^T e + (e_m)^T \Phi_m e_m + (e_M)^T \Phi_M e_M}{\text{grad}_q v(q(k))^T \cdot \text{grad}_q v(q(k))}$$

Table 5.4. Position Errors

Iteration number	0	2	4	6	8	10	12
Position error	1.6929	0.6700	0.4838	0.1335	0.0504	0.0218	0.0099

The solution of the inverse kinematic problem with limited joint ranges presented here uses only direct kinematic models, namely $\mathbf{t}(\mathbf{q})$ and Jacobian $\mathbf{J}(\mathbf{q})$. The computational burden can be drastically reduced by neural implementation of the performance function and its gradient.

The Jacobian $\mathbf{J}(\mathbf{q})$ can be obtained directly from the output of sinusoidal units of a feedforward neural network in a similar way as for the forward kinematics $\mathbf{t}(\mathbf{q})$.

The penalty function ω and its gradient can be computed by a neural network that implements the joint range limits. Each joint's limit is represented by two neurons with the sigmoid activation function ϕ . The outputs of these neurons are used directly to calculate the gradient of ω .

A neural implementation of Equation (5.46) defining the update rule for a feedback network is shown in Figure 5.13. The update of \mathbf{q} based on $\dot{\mathbf{q}}$ guarantees the convergence of the feedback network output to a given location ℓ_{Final} if this location lies in the joint's range. The control unit of the network stops the iterative calculations when the location error e is less than a given limit or if the number of iterations exceed some other given limit. The above approach corresponds with the method developed by Sciavicco and Siciliano (1988) and used by Lee (1991). When the neural network implementation of the inverse kinematic solution is extended to the case of redundant manipulators, constraints on the joint variables can be automatically incorporated in the solution algorithm. Neural implementation of the inverse kinematics yields the continuity of the solution, drastic reduction of computational time, and generation of joint velocities without additional cost. The end point of the manipulator keeps tracking the desired trajectory and the resulting joint trajectories avoid violating the mechanical limits on the joint variable. The activation and deactivation of constraints are automatically performed by the neural subnetwork representing the limit constraints.

Example 5.1.5. The Figure 5.14 presents the behavior of the neural gradient algorithm in the case of the three-link and 2D arm model with link length equal to 0.5 (the column length is 0.8).

In the simulation run the algorithm had to move the manipulator end from the start point $S = (0.8, 1.5)$ to the final point $F = (0.0, 1.8)$.

It can be seen in Figure 5.14 that only 12 iterations were necessary to achieve a solution with error value less than 0.01 (see Table 5.4). The error measure is the Euclidean distance between the current and desired positions of the arm end.

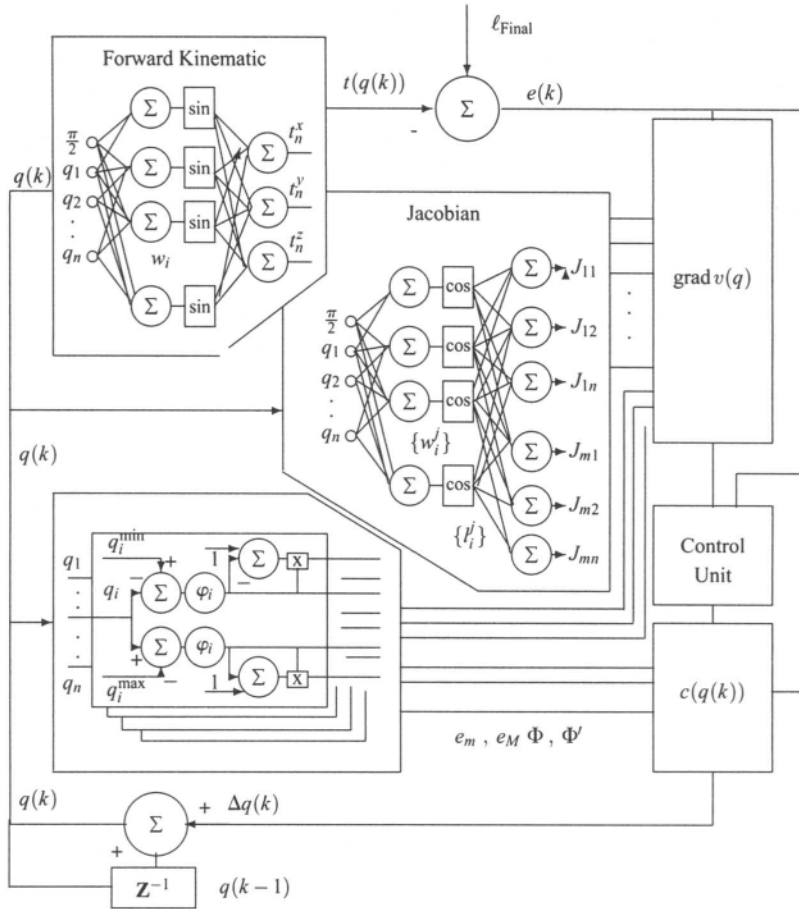


Figure 5.13. Inverse kinematics for manipulators constrained with joint ranges.

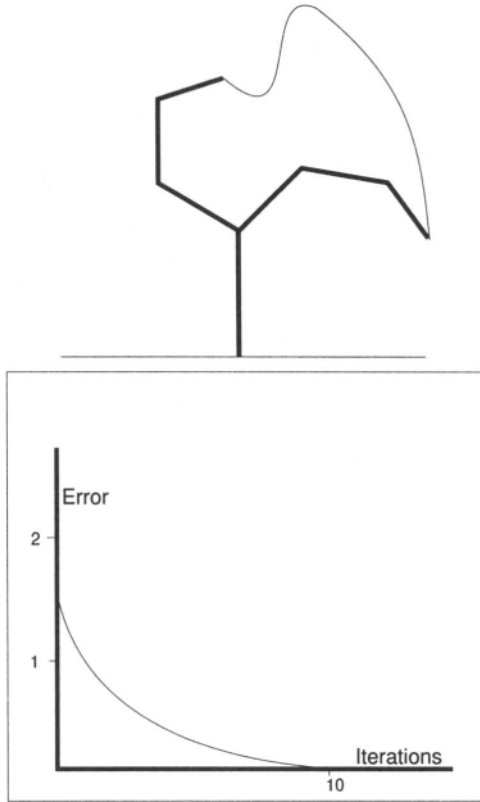


Figure 5.14. Experimental results.

5.1.3.2. *State transition f of the system M .* Based on the above method we can define the state transition function f of the system M as follows:

$$q(k+1) = f(q(k), u(k)) = \text{Inverse}(t(q(k)) + u(k)) \quad (5.48)$$

where $u = \delta \ell$.

The output function g is the same as in the previous model M [Equation (5.10)], i.e.,

$$g(q(k)) = (P_0, t_i(q(k)) | i = 1, \dots, n)^T \quad (5.49)$$

This completes the full description of the model M with respect to the Cartesian space discretization.

5.1.3.3. *Search for collision-free path.* Recall that the problem of collision-free robot movement planning amounts to finding a sequence $u_k^* = (u(1), \dots,$

$u(K)$) of input signals of the model M such that the terminal configuration reaches the effector's final position P_{Final} , no configuration collides with any obstacle in the robot's environment, and the input sequence minimizes the motion-cost function, expressing, e.g., the length of the effector path.

In order to solve the path planning problem we apply the same graph-searching procedure to the state transition graph of the model M as for the previously presented model specification. The development of the search graph starts from the node(configuration) $q(0)$ by the action f for all possible input signals from the set U , i.e., the successor set of each node $q(k)$ is given by

$$\text{Succ}(q(k)) = \{\text{Inverse}(t(q(k)) + u) | u \in U\} \quad (5.50)$$

The number of successors of the node q depends on the type of discrete geometry used to discretize the Cartesian space.

When the 6-conjunctive discrete geometry is used, the generalized displacement vector $\delta\ell$ has one of the forms

$$\delta\ell \in \{(\pm\delta x, 0, 0, 0, 0, 0), \dots, (0, 0, 0, 0, 0, \pm\delta y)\}$$

Then, for a given configuration q , the state transition function f generates 12 successors of q , independent of the number of degrees of freedom of the manipulator. It reduces greatly the computational complexity of the graph-searching procedure.

The successor set generation using the kinematic model. For a kinematically redundant manipulator the dimension of the joint space is greater than the dimension of the Cartesian space $\text{dim}(q) = n > \text{dim}(\ell) = m$. These extra degrees of freedom of a redundant manipulator can be used to achieve some subgoals such as singularity avoidance or obstacle avoidance (Maciejewski and Klein, 1985; Chirikjian and Budrick, 1990; Lee and Lee, 1990). Although one or two degrees of redundancy can be used to satisfy the above subgoals, the use of hyperredundancy becomes very attractive because of the flexibility and dexterity in motion, it offers for given tasks in a complex environment. However, studies on the kinematic control of redundant manipulators with a large number of degrees of redundancy have not been performed extensively because of the lack of appropriate techniques (Lee and Lee, 1990).

Note that a link avoids a joint's range limit if its distance from the limit is greater than some epsilon distance depending on the value of β of the sigmoid function φ . If all links satisfy this condition, there is no reason to modify the current solution, and the model will select one of the ∞^{n-m} possible configurations, depending on the initial joint configuration. Such a configuration is said to be not feasible if it collides with the obstacles in the work scene, but this does not mean that there does not exist another collision-free configuration from the ∞^{n-m} possible configurations.

COLLISION-FREE ROBOT CONFIGURATIONS: Since the inverse kinematic algorithm provides joint configurations which are adjacent to each other as the manipulator proceeds through the successive effector-end locations, one or more constraints need to be introduced directly during the calculation of the successor set in order to avoid collisions with obstacles. One of the potential advantages of a kinematically redundant manipulator is the use of the extra (redundant) degrees of freedom to maneuver in a complex workspace and avoid contact with obstacles.

It can be assumed that a link has avoided a convex obstacle if its minimum distance from the obstacle is greater than a preplanned threshold distance (Sciavicco and Siciliano, 1988).

If the distance between one of the links and an obstacle becomes less than the threshold, the current solution is to be modified. It is understood that at most $n - m$ constraints of such type can be activated.

The coupled neural network realizing the gradient algorithm [Equation (5.47)] calculates a joint configuration that is a solution to the inverse kinematic problem. Nevertheless, it can happen that even though the end-effector position is collision-free, collisions occur on some of the manipulator links. In this case the manipulator configuration has to be changed. To do so, we can use the same gradient algorithm with a modified performance function. Such a function can be defined similarly to the performance function for joint limitation. Below we briefly present the definition of the distance performance function.

Let

$$p_i(s) = t_i(q) + s \cdot [t_{i+1}(q) - t_i(q)] = t_i(q, s), \quad s \in [0, 1]$$

indicate the position of a point situated on the i th link of the robot skeleton, and $o \in OB$ denote the point of an obstacle at a minimum distance from the link L_i to the obstacle, i.e., the i th link-obstacle distance is $\rho(L_i, OB) = \|o - p_i(s)\|$ (see Figure 5.15).

Additionally, let τ be the threshold distance. If the distance $\rho(L_i, OB)$ becomes less than the threshold distance τ , there is a danger of collision, and the joint changes need to be modified according to the new constraints.

In analogy to the penalty function definition for the avoidance of joint limits, we use the additional errors for each link L_i

$$e_i^L = \tau^2 - [p_i(s) - o]^T [p_i(s) - o] = \tau^2 - [t_i(q, s) - o]^T [t_i(q, s) - o] \quad (5.51)$$

The penalty function can be defined as

$$\omega_L(q) = \frac{1}{2} e_L^T \phi_L \quad (5.52)$$

where $e_L = (e_i^L | i = 1, \dots, n)^T$, and $\phi_L = (\varphi_i^L | i = 1, \dots, n)^T$ is the vector of sigmoid functions

$$\varphi_i^L = \frac{1}{1 + e^{-\beta \cdot e_i^L}} \quad (5.53)$$

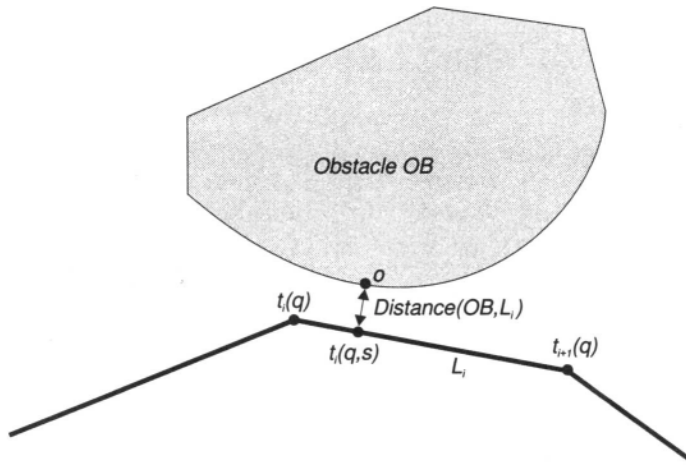


Figure 5.15. Distance between an obstacle and a link of the manipulator.

Then the modified performance function is

$$w(q) = v(q) + \omega_L(q) \quad (5.54)$$

By using the sigmoid function ϕ in the penalty function ω_L with a large value of β we activate the error e_i^L only if

$$\rho(L_i, OB) < \tau$$

The gradient of the modified function w has the following form:

$$\mathbf{grad} w(q) = \mathbf{grad} v(q) - \sum_2^n \phi_i^L (1 - \beta e_i^L (1 - \phi_i^L)) J_i(q, s)^T [t_i(q, s) - o] \quad (5.55)$$

where $J_i(q, s)$ is the Jacobian matrix of the point $p_i(s)$ closest to the obstacle.

If there are many obstacles in the scene, we consider only the obstacle which is closest to the link. Such a penalty function corresponds to the approach proposed in (Sciavicco and Siciliano, 1988). For computational experiments we use the neural network representation of the function ω .

Example 5.1.6. The simulation results are presented in Figure 5.16. The task of the manipulator was to trace a path (from a start point **Start** to the end point **Final**) that had been specified by a linear path planner.

The gradient algorithm used for the simulations is

$$q(k+1) = q(k) - \eta \frac{v + \omega}{\|\mathbf{grad} v + \gamma \mathbf{grad} \omega\|^2} (\mathbf{grad} v + \gamma \mathbf{grad} \omega) \quad (5.56)$$

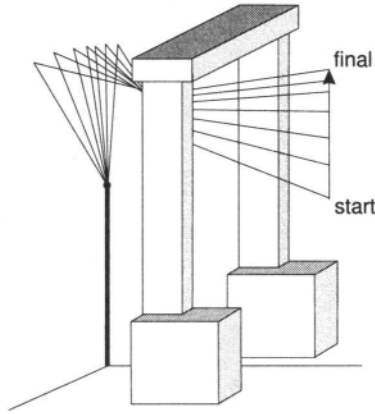


Figure 5.16. Skeleton tracks of collision-free movements.

where γ is the weight of the gradient of ω .

The length of each manipulator link L_i was set to 1.0. The parameter values used in the gradient algorithm were $\eta = 0.1$ and $\gamma = 100$ and the initial manipulator configuration was $q_{ini} = (0^\circ, 150^\circ, 65^\circ, 190^\circ)$.

As can be seen in Figure 5.16, the neural-processed gradient algorithm produced collision free manipulator configurations. In each step of the algorithm the pairs of nearest link-to-obstacle points were calculated and then used in the parametrized Jacobian calculation.

As mentioned, the gradient algorithm is not able to find a solution if too many constraints are active. Testing revealed that in such cases the gradient algorithm produces undesirable solutions. One more disadvantage of the proposed method is that the configurations obtained during path tracing depend on the initial configuration. This sometimes leads the algorithm to a bad solution, even if a correct solution exists.

Search process. To find the shortest path in the state-transition graph connecting the initial node $q(0)$ and the final node $q_{final} = q(K)$, we use the A* algorithm (Pearl, 1984; Pearl, 1988; Nilsson, 1980). As the cost function between two nodes q, q' joined by an arc, we shall assume the Euclidean distance in Cartesian space traveled by the effector while passing from configuration q to q' [Equation (5.29)]. The function which estimates the distance from the actual configuration q to the final position P_{Final} is defined as the rectilinear distance between the effector end in configuration q and the terminal position. It is easily seen that the heuristic function is the lower bound of the function calculated for the path between q and the goal node.

Let configuration q be chosen as the best node from the list OPEN (Nilsson, 1980; Jacak, 1989b). For q let us find the set of collision free successors based on

the above kinematic model M . Remove from this set the configurations which are ancestors of q . In order to simplify the procedure of successor generation, we can additionally introduce a list of initially empty forbidden nodes FORBID and add to it the nodes for which there does not exist a collision-free configuration.

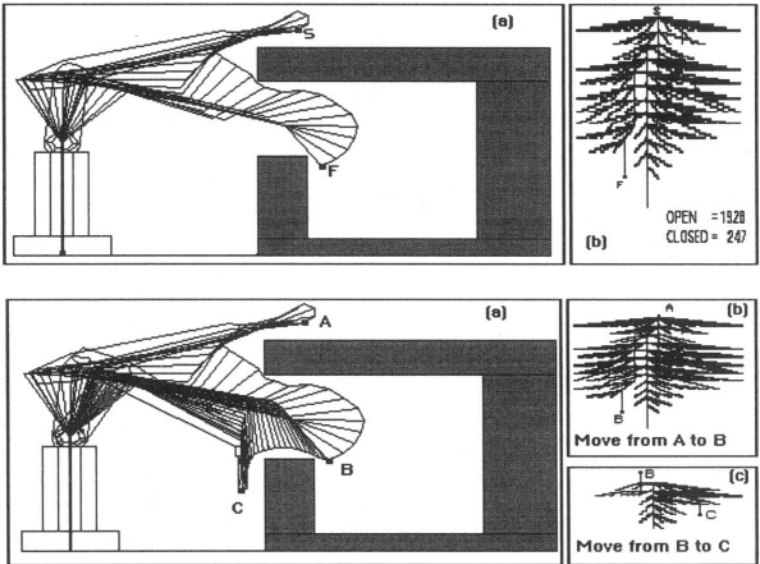
From the reduced set of successors, we choose those configurations which have not been present until now on the list OPEN or CLOSED, and introduce the configuration into OPEN, establishing q as their predecessor. For all new and newly marked nodes of the list OPEN we calculate the evaluation function, and order the list according to increasing values of it. Then we repeat the procedure.

Remarks. In this chapter we have presented two search methods for collision-free robot motion planning using two different models of robot kinematics, with neural processing as the background. Both models have the form of a discrete dynamical system which can be directly used for search-graph generation.

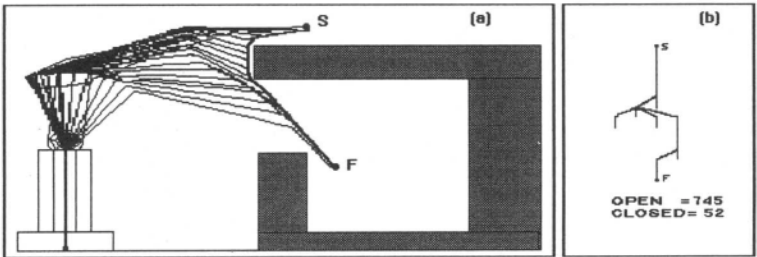
The first model, based on joint space discretization, allows us to use a few efficient algorithms of graph searching (Pearl, 1984; Pearl, 1988; Nilsson, 1980) with different costs and heuristic functions. Basic advantages of such a model lie in the possibility of expressing successive robot configurations in the basic Cartesian frame by means of neural implementation of its output function. Due to the cardinality of the model's input set, this planning technique is well applicable to robots that do not have a large number of degrees of freedom—typically, $n < 7$. Indeed the number of discrete configurations that are successors of the given configuration q increases exponentially with the number of degrees of freedom. An illustration of this approach of planning a collision-free path for the manipulator is shown in the top part of Figure 5.17, where simulation results are presented for a robot with 4 degrees of freedom in macromovements. Figure 5.17 visualizes the track of the skeleton from initial to final position and shows the expanded planning search tree. However, when n becomes too big, the size of the successor set becomes too large.

In this case, the second method, employing a kinematic model based on Cartesian space discretization, is more efficient. The simulation results for the same manipulator are shown in the bottom part of Figure 5.17. The scene is identical to the scene in the previous example. Figure 5.17 illustrates successively selected configurations of the whole manipulator and the expanded planning search tree. To increase the efficiency of the search procedure we can additionally introduce to the heuristic function precomputed values of the numerical potential field using the wavefront expansion algorithm (Barraquand et al., 1992; Latombe, 1991). However, since it is a heuristic method, it does not always guarantee finding the optimal track.

The second model also can be a good tool for searching for the robot motion trajectory in the presence of obstacles in the real world, by using proximity sensors (Espiau and Boulic, 1985).



Collision-free robot movements (joint space planning)



Collision-free robot movements (Cartesian space planning)

Figure 5.17. Skeleton tracks of collision-free robot movements.

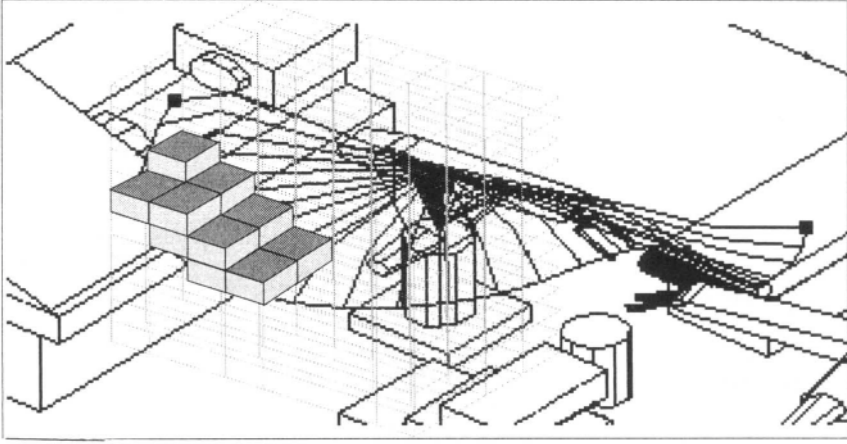


Figure 5.18. Voxel tracks of collision-free motion.

5.1.3.4. *Continuous track calculation.* From the path planner, we obtain an ordered sequence

$$\text{track}^* = (q(0), \dots, q(K)) \quad (5.57)$$

of configurations which defines how to move the effector end from the initial to the final position. Then a geometric track can be constructed by connecting configurations from track^* . This is accomplished by using cubic splines (Jacak et al., 1992). Finally, the geometric track is given in the form of a parametrized curve:

$$q_{\text{track}} : [s_0, s_{\text{fin}}] \rightarrow Q \quad (5.58)$$

where the initial and final configurations of the track $q(0)$ and $q(K)$ correspond to the points $s = s_0$ and $s = s_{\text{fin}}$, respectively.

Additionally, we introduce the raster representation of the geometric track of robot movement. The service space of each robot $\text{Srv_Sp}(r)$ can be discretized in the form of a cubic raster, i.e.,

$$\text{Srv_Sp}(r) = \{\text{cub}_{ijk} | (i, j, k) \in I_r \times J_r \times K_r\} \quad (5.59)$$

The path planner generates all possible geometric tracks of robot movements. For a given track $q_{\text{track}}(s)$ we can establish the subset of raster elements (voxels) which are visited by the robot manipulator during the motion (see Figure 5.18):

$$\text{Tor}^r(q_{\text{track}}) = \{\text{cub} \in \text{Serv_Sp} | (\exists s)(\text{Seg}(g(q(s)))) \cap \text{cub} \neq \emptyset\} \quad (5.60)$$

where *Seg* is the broken line joining the points of the output function *g* of model *M* [Equation (5.1)] and represents the robot's skeleton. The set of volume elements *Tor* is a gross approximation of the geometric track of robot movement and can be used for fast collision detection between two moving robots. Based on the methods presented above, the CARC path planner is capable of automatically calculating collision-free tracks for all motions of the robots in a cell with respect to a given production route **p**, as well as for one selected movement.

Example 5.1.7. The result shown in Figure 5.19 demonstrate the collision-free paths for the holding motions from the optimal route **p*** calculated by the process planner for **Example_Cell**.

Now the optimal speed and acceleration of movements along the computed tracks can be found by the trajectory planning package of CARC.

5.2. Time-Trajectory Planner

The trajectory planner receives the geometrical tracks as input and determines a time history of position, velocity, acceleration, and input torques which then can be fed to the trajectory tracker. In the trajectory planner, the robot is represented by a model of the manipulator dynamics (Brady, 1986; Shin and McKay, 1986; Shin and McKay, 1985).

5.2.1. Modeling of Robot Dynamics

There are a number of ways to obtain the dynamic equations of the robot manipulator. The trajectory planner uses the Lagrange description, which yields a set of differential equations in the form

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{N}(\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{R}\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) = \mathbf{F} \quad (5.61)$$

or

$$\sum_{j=1}^n M_{ij}(\mathbf{q})\ddot{q}_j + \sum_{j=1}^n \sum_{k=1}^n N_{jk}^i(\mathbf{q})\dot{q}_j\dot{q}_k + \sum_{j=1}^n R_{ij}\dot{q}_j + G_i(\mathbf{q}) = F_i \quad i = 1, \dots, n \quad (5.62)$$

where $\mathbf{M}(\mathbf{q})$ is the $(n \times n)$ inertia matrix of the robot, $\mathbf{N}(\mathbf{q})$ is the $(n \times n \times n)$ array of centrifugal and Coriolis terms for the robot, \mathbf{R} is the $(n \times n)$ diagonal matrix of viscous friction coefficients for the robot, $\mathbf{G}(\mathbf{q})$ is the $(n \times 1)$ vector of the robot's center-of-gravity terms, \mathbf{F} is the $(n \times 1)$ vector of the robot's input torques and forces, \mathbf{q} is the $(n \times 1)$ vector of generalized coordinates (configuration), and n is the number of degrees of freedom (DOF).

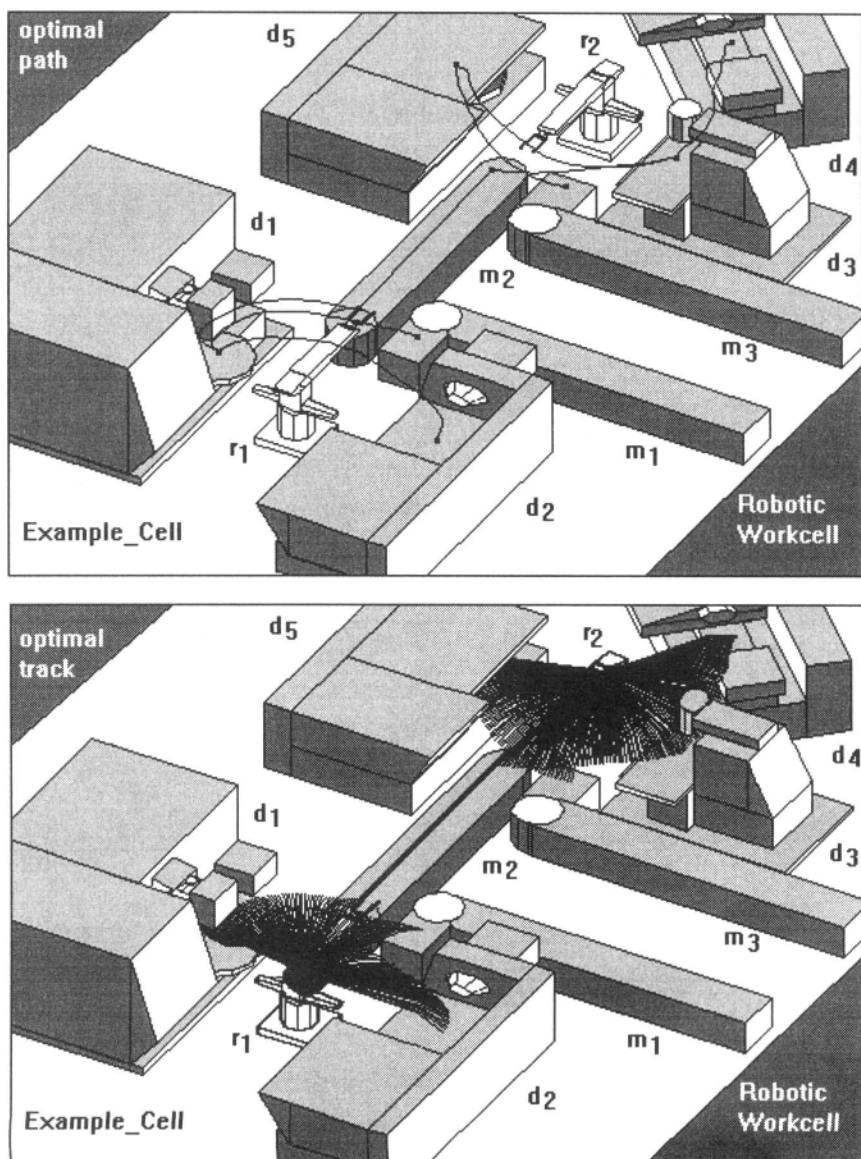


Figure 5.19. Collision-free motion realizing the optimal route p^* .

The first step to obtain the equations consists of modeling the robot body. The body can be considered as a set of elementary parts (cones, cylinders, cubes, etc.) that make up the links of the robot. Then the dynamic parameters of the robot's links (masses, inertia matrix coefficients, centers of gravity) are determined as functions of the parameters of elementary parts (lengths, radii, heights, etc.).

The equations of the robot's dynamics can be obtained by applying the Euler–Lagrange formalism or the Newton's approach.

In the first approach the principle of minimal action is exploited to derive the Euler–Lagrange equation:

$$\frac{d}{dt} \frac{\partial L(q, \dot{q})}{\partial \dot{q}} - \frac{\partial L(q, \dot{q})}{\partial q} = F \quad (5.63)$$

where $L(q, \dot{q}) = E_k(q, \dot{q}) - E_p(q)$ is the Lagrange function (difference between kinetic and potential energy of a robot) and F denotes the external forces/torques acting on the robot. In this approach the equations of the robot's dynamics are obtained in closed form.

In the second approach, Newton equations of motion are formulated for each component of the robot. In this approach the resulting equations offer the robot's dynamics most often take a recursive form.

The two approaches are equivalent not only in the sense that they lead to the same equations, but also in that the computational effort to calculate them is almost the same. In the past, the Newton approach was preferred when calculations were performed on sequential computers.

For modeling the robot's dynamics we will take advantage of the Euler–Lagrange approach. There are many schemes to compute the dynamics in such a manner as to get the resulting equations as quickly as possible. We concentrate on the scheme described by Luh *et al.* (1986), in which the equations of the robot's dynamics take the form:

$$F_i = \sum_{j=i}^n \left\{ \sum_{k=1}^j \text{tr}[U_{jk} I_j (U_{ji})^T] \ddot{q}_k \right\} + \sum_{j=i}^n \left\{ \sum_{k=1}^j \sum_{m=1}^j \text{tr}[U_{jkm} I_j (U_{ji})^T] \dot{q}_k \dot{q}_m \right\} - \sum_{j=i}^n m_j g_{\text{rav}}^T U_{ji} r_j \quad (5.64)$$

where

$$U_{jk} = \begin{cases} T_0^{k-1} Q_k T_{k-1}^T & \text{for } j \geq k \\ 0 & \text{otherwise} \end{cases}$$

$$U_{jkm} = \begin{cases} T_0^{k-1} Q_k T_{k-1}^{m-1} & \text{for } j \geq m \geq k \\ T_0^{m-1} Q_m T_{m-1}^{k-1} & \text{for } j \geq k \geq m \\ 0 & \text{otherwise} \end{cases}$$

$$Q_k = \begin{cases} \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \text{if joint } k \text{ is rotational} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \text{if joint } k \text{ is translational} \end{cases}$$

where T_j^k is the 4×4 matrix transforming coordinates from the k th frame to the j th frame, i.e., $T_j^k = A_j \cdot A_{j+1} \cdots A_k$; I_i is the pseudo-inertia matrix for the i th link (expressed in the i th coordinate frame); \mathbf{g}_{rav} is the gravity vector; m_i is the mass of the i th link; \mathbf{r}_i is the vector connecting the center of gravity of the i th link with the origin of the i th coordinate frame (and expressed in the i th coordinate frame), and tr is the trace operator.

Fact 5.2.1. *The components U_{jk} are*

$$U_{jk} = \frac{\partial T_0^j}{\partial q_k}$$

and

$$U_{jkm} = \frac{\partial^2 T_0^j}{\partial q_k \partial q_m}$$

Computing partial derivatives can be avoided by multiplying subchain matrices by the matrix Q .

□

□

In order to derive Equation (5.64), the kinematic and dynamic parameters should be known. The kinematic parameters influence the transformation matrices [Equation (5.13)] $T_0^i = T_i$ for $i = 1, \dots, n$, while the dynamic ones (together with the kinematic parameters) influence the pseudo-inertia matrices I_i (where $i = 1, \dots, n$). Kinematic parameters were discussed in the previous chapter, so here we concentrate on dynamic ones. There are two possible ways to obtain these parameters: by modeling the robot's body, and by doing measurements on its parts.

The first method is usually applied in CAD/CAM systems while the second is used by robot manufacturers and when doing research on robots. We follow the first route and model the robot. As a rule, some primitive objects are needed to model the robot's body. Primitives need to be simple in shape and appropriate for approximating any volume. The most popular elementary objects are balls, ellipsoids, cones, and cylinders. The derivation of the pseudo-inertia matrix I_0 for

every primitive object (expressed in its own coordinate frame attached to its center of gravity) is a simple task. As can be seen from Equation (5.64), pseudo-inertia matrices should be expressed in appropriate frames, usually different from the frames attached to the center of gravity (CG) of each link. The Steiner Theorem can be used to transform pseudo-inertia matrices from the CG frame to the link frame:

$$I = T \cdot I_0 \cdot T^T \quad (5.65)$$

where I_0 is the pseudo-inertia matrix in the frame attached to the CG, I is the pseudo-inertia matrix in the link's frame, and T is the transformation matrix from the CG frame to the link's frame. Superscript T denotes matrix transposition.

It should be pointed out that equation (5.64) is an *idealized* version of the equation of a real robot. There are at least two main sources of differences between the two sets of equations:

- The formalism to derive equation (5.64) is based on an energy-preserving principle, so dissipative phenomena are not taken into account (e.g., viscous friction is determined by doing experiments on a real robot).
- When modeling a robot's body by simple (unified) components, any irregularity in material density or shape cannot be properly modeled; additionally, real robot parameters can vary from one item to another and even parameters of the robot as given by a manufacturer can be used only as approximate values. Moreover, some parameters can vary over the usage time in an unpredictable manner.

The coefficients of Equation (5.62) are not independent of one another. Coriolis and centripetal terms can be obtained as follows (McKerrow, 1991):

$$N_{jk}^i(q) = \frac{1}{2} \left(\frac{\partial M_{ik}(q)}{\partial q_j} + \frac{\partial M_{ij}(q)}{\partial q_k} - \frac{\partial M_{jk}(q)}{\partial q_i} \right) \quad (5.66)$$

Below we list the most important properties of the coefficients:

1. $M_{ij} = M_{ji}$ for $i, j = 1, \dots, n$.
2. $N_{jk}^i = N_{kj}^i$ for $i, j, k = 1, \dots, n$.
3. $N_{jk}^i = -N_{ik}^j$ for $i, j \geq k$.

Example 5.2.1. Following the Euler-Lagrange formalism, we modeled the PUMA-like robot depicted in Figure 5.20; its kinematic parameters are presented in Table 5.5 and a complete list of parameters of the PUMA-like robot is presented

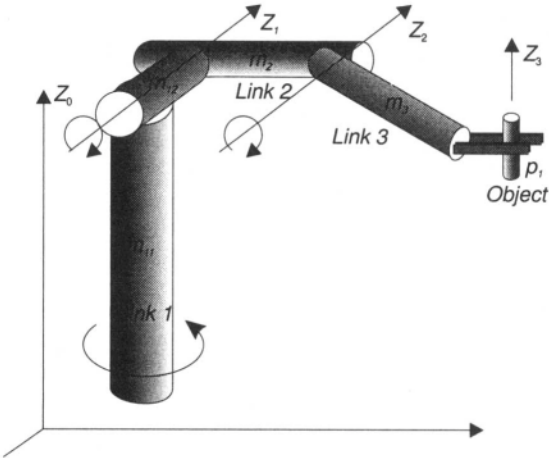


Figure 5.20. A PUMA-like robot with coordinate axes.

in Table 5.6. The pseudo-inertia matrices for the PUMA-like robot modeled as a set of cylinders (cf. Figure 5.20) are

$$I_1 = \begin{bmatrix} j_{xx1} & 0 & 0 & 0 \\ 0 & j_{yy1} & 0 & m_1 y_1 \\ 0 & 0 & j_{zz1} & m_1 z_1 \\ 0 & m_1 y_1 & m_1 z_1 & m_1 \end{bmatrix}$$

$$I_2 = \begin{bmatrix} j_{xx2} & 0 & 0 & m_2 x_2 \\ 0 & j_{yy2} & 0 & 0 \\ 0 & 0 & j_{zz2} & m_2 z_2 \\ m_2 x_2 & 0 & m_2 z_2 & m_2 \end{bmatrix}$$

Table 5.5.
Denavit-Hartenberg
Parameters for a
PUMA-Like Robot

Link	Q_i	d_i	a_i	α_i
1	q_1	0	0	$-\pi/2$
2	q_2	0	a_2	0
3	q_3	d_3	a_3	$\pi/2$

Table 5.6. Dynamic Parameters of a PUMA-Like Robot

gr [m/s ²]	earth's gravity
m_{11} [kg]	mass of body No. 1
m_{12} [kg]	mass of body No. 2
h_1 [m]	height of body No. 1
h_2 [m]	height of body No. 2
r_1 [m]	radius of body No. 1
r_2 [m]	radius of body No. 2
k_1 [m]	distance from CG11 to 1st frame's origin along y-axis
k_2 [m]	distance from CG12 to 1st frame's origin along z-axis
m_2 [kg]	mass of link No. 2
r_3 [m]	radius of link No. 2
h_3 [m]	height of link No. 2
l_1 [m]	distance from CG2 to 2nd frame's origin along x-axis
l_2 [m]	distance from CG2 to 2nd frame's origin along z-axis
m_3 [kg]	mass of link No. 3
r_4 [m]	radius of link No. 3
h_4 [m]	height of link No. 3
l_4 [m]	distance from CG3 to 3rd frame's origin along x-axis
op_1 [kg]	mass of an object
op_2 [m]	radius of an object
op_3 [m]	height of an object
a_2	kinematic parameter of Link No. 2
d_3	kinematic parameter of Link No. 2
a_3	kinematic parameter of Link No. 3

$$I_3 = \begin{bmatrix} j_{xx3} + j_{xx} & 0 & 0 & (m_3 + op_1)x_3 \\ 0 & j_{yy3} + j_{yy} & 0 & 0 \\ 0 & 0 & j_{zz3} + j_{zz} & 0 \\ (m_3 + op_1)x_3 & 0 & 0 & m_3 + op_1 \end{bmatrix}$$

The following equations connect the robot parameters with the terms of the pseudoinertia matrices.

Link No. 1:

$$m_1 = m_{11} + m_{12}, \quad j_{xx1} = m_{11}r_1^2/4 + m_{12}r_2^2/4 \quad (5.67)$$

$$j_{yy1} = m_{11}h_1^2/12 + m_{11}k_1^2 + m_{12}r_2^2/4, \quad j_{zz1} = m_{11}r_1^2/4 + m_{12}h_2^2/12 + m_{12}k_2^2 \quad (5.68)$$

$$y_1 = k_1 m_{11}/m_1, \quad z_1 = k_2 m_{12}/m_1 \quad (5.69)$$

Link No. 2:

$$j_{xx2} = m_2 h_3^2/12 + m_2 l_1^2, \quad j_{yy2} = m_2 r_3^2/4, \quad j_{zz2} = m_2 r_3^2/4 + m_2 l_2^2 \quad (5.70)$$

$$x_2 = -l_1, \quad z_2 = l_2 \quad (5.71)$$

Link No. 3 and an object:

$$j_{xx3} = m_3 h_4^2/12 + m_3 l_4^2, \quad j_{yy3} = m_3 r_4^2/4 \quad (5.72)$$

$$j_{zz3} = m_3 r_4^2/4, \quad x_3 = -l_4 m_3/(m_3 + op_1) \quad (5.73)$$

$$j_{xx} = op_1 op_2^2/4, \quad j_{yy} = op_1 op_2^2/4, \quad j_{zz} = op_1 op_3^2/12 \quad (5.74)$$

The pseudo-inertia matrix for an object is defined as

$$I = \begin{bmatrix} \int_m x^2 dm & \int_m xy dm & \int_m xz dm & \int_m x dm \\ \int_m yx dm & \int_m y^2 dm & \int_m yz dm & \int_m y dm \\ \int_m zx dm & \int_m zy dm & \int_m z^2 dm & \int_m z dm \\ \int_m x dm & \int_m y dm & \int_m z dm & \int_m 1 dm \end{bmatrix}$$

where dm is an infinitesimal element of a mass, and (x, y, z) are the coordinates of the elements in the frame in which the pseudo-inertia matrix is calculated.

We assume that the cylinder has a height ph_1 , a radius pr_2 , and a constant density ρ . For a cylinder with a mass distributed uniformly (mass density is a constant) and a coordinate frame attached at the cylinder center of gravity, and for the z -axis lying along the height of the cylinder, the matrix takes the form

$$I = \begin{bmatrix} mpr_2^2/4 & 0 & 0 & 0 \\ 0 & mpr_2^2/4 & 0 & 0 \\ 0 & 0 & mph_1^2/12 & 0 \\ 0 & 0 & 0 & m \end{bmatrix}$$

The following examples give the coefficients of the equations of the robot's dynamics derived using the Euler-Lagrange approach and technical data for a PUMA-like robot:

$$\begin{aligned} M_{11} = & j_{xx}/2 + j_{xx1}/2 + j_{xx2}/2 + j_{xx3}/2 + j_{yy} + j_{yy2}/2 + j_{yy3} + j_{zz}/2 + j_{zz1} + j_{zz2} \\ & + j_{zz3}/2 + a_2^2(m_2 + m_3 + op_1)/2 + a_3^2(m_3 + op_1)/2 + d_3^2(m_3 + op_1) \\ & + a_2 m_2 x_2 + a_3 x_3(m_3 + op_1) + (j_{xx2} - j_{yy2}) \cos(2q_2)/2 \\ & + a_2^2(m_2 + m_3 + op_1) \cos(2q_2)/2 \\ & + a_2 m_2 x_2 \cos(2q_2) + a_2(a_3 + x_3)(m_3 + op_1) \cos(q_3) \\ & + a_2(a_3 + x_3)(m_3 + op_1) \cos(2q_2 + q_3) \\ & + (j_{xx} + j_{xx3} - j_{zz} - j_{zz3}) \cos(2q_2 + 2q_3)/2 \\ & + a_3(a_3(m_3 + op_1)/2 + x_3(m_3 + op_1)) \cos(2q_2 + 2q_3) \end{aligned}$$

$$\begin{aligned} M_{12} = & (a_2 d_3(m_3 + op_1) + (a_2 + x_2)m_2 z_2) \sin(q_2) \\ & + d_3(a_3 + x_3)(m_3 + op_1) \sin(q_2 + q_3) \end{aligned}$$

$$M_{13} = d_3(a_3 + x_3)(m_3 + op_1) \sin(q_2 + q_3)$$

$$N_{11}^1 = 0$$

$$\begin{aligned} N_{12}^1 = & -(j_{xx_2} + j_{yy_2}) \sin(2q_2)/2 - a_2^2(m_2 + m_3 + op_1) \sin(2q_2)/2 \\ & - a_2 m_2 x_2 \sin(2q_2) - a_2(a_3 + x_3)(m_3 + op_1) \sin(2q_2 + q_3) \\ & - (j_{xx} + j_{xx_3} - j_{zz} - j_{zz_3}) \sin(2q_2 + 2q_3)/2 \\ & - a_3(a_3(m_3 + op_1)/2 - x_3(m_3 + op_1)) \sin(2q_2 + 2q_3) \end{aligned}$$

$$\begin{aligned} N_{13}^1 = & -a_2(a_3 + x_3)(m_3 + op_1) \sin(q_3)/2 \\ & - a_2(a_3 + x_3)(m_3 + op_1) \sin(2q_2 + q_3)/2 \\ & + (-j_{xx} - j_{xx_3} + j_{zz} + j_{zz_3}) \sin(2q_2 + 2q_3)/2 \\ & - a_3^2(m_3 + op_1) \sin(2q_2 + 2q_3)/2 - a_3 x_3(m_3 + op_1) \sin(2q_2 + 2q_3) \end{aligned}$$

$$\begin{aligned} g_{\text{rav}}^2 = & gr(((a_2 + x_2)m_2 + a_2(m_3 + op_1)) \cos(q_2) \\ & + (a_3 + x_3)(m_3 + op_1) \cos(q_2 + q_3)) \end{aligned}$$

$$g_{\text{rav}}^3 = gr \cdot (a_3 + x_3) \cdot (m_3 + op_1) \cos(q_2 + q_3)$$

5.2.2. Symbolic and Neural Network-Computed Robot Dynamics

In order to fully utilize a robot's capability its dynamics should be taken into account. Because a robot's controllers have to calculate the dynamics in each control interval (usually tens of milliseconds), steps need to be taken to make the calculations possible. Approaches used to reduce the computational complexity of calculating the dynamic model include the following:

- Ignore terms which are in some sense small. The omission can be general in scope or only local, valid for particular trajectories or points. Leaving out some terms may cause problems in preserving the robot structure (e.g., symmetry and positive definiteness of its inertia matrix).
- Measure some quantities instead of calculating some terms. Some dynamic parameters can be estimated on the basis of the measurements.

Unfortunately these methods introduce new error sources. Computation on parallel computers significantly reduces the amount of time needed to compute a robot's dynamics. The true solution of the problem of implementing the dynamics will come through the use of powerful integrated chips, which will make it possible to apply neural network techniques that can exploit the speed of optoelectronics and give flexibility and robustness.

For robotic applications the most promising activation function of a single neuron is the sinusoidal function (Jacak et al., 1994b; Lee and Bekey, 1991), although general (not specialized) structures are in use as well (Miller et al., 1990). The reason for this is that the main robotic transformations — kinematic, dynamic, Jacobian matrix — use sine and cosine (sines with biases in neural terms) functions as elementary blocks of expressions. It is worth noting here that by knowing the structure of a robot's kinematics or dynamics in advance, the problem of deriving the equations governing the behavior of a real robot reduces to merely identifying coefficients. In this way the problem of structure identification is reduced to the much simpler problem of parameter estimation. First let us observe that having fixed the structure of a robot's dynamics, the whole information about the dynamics is found in its coefficients $(G_i(q), M_{ij}(q), N_{jk}^i(q), R_i, i, j, k = 1, \dots, n)$. So it is enough to transform the coefficients into neural network form to get the neural representation of the robot's dynamics. As can easily be seen, the coefficients are functions of the robot configuration q and are in fact kinematic-like in nature. The transformation assumes that the robot has only rotary joints, so the motion variables $(q_i s)$ enter into coefficients only as functions of sines or cosines. Therefore any coefficient can be expressed as follows:

$$\text{coe}^i = \sum_j a_j^i \sin \left(\sum_k^n w_{jk}^i q_k + \text{bias}_j^i \right) \quad i = 1, \dots, I_{\text{coe}} \quad (5.75)$$

where $a_j, w_{jk}, \text{bias}_j \in \mathbb{R}$, coe^i is an expression from the set $\{G_i(q), M_{ij}(q), N_{jk}^i(q) | i, j, k = 1, \dots, n\}$, and I_{coe} is the number of coefficients.

It is a simple exercise to transform any expression in the form of a multiplication of sines/cosines into the form described by Equation (5.75). Assuming that a sinusoidal neuron is available (a neuron with sinusoidal activation function), any of the coefficients can be implemented as a neural network. The structure of the neural network model of a robot's dynamics is shown in Figure 5.21.

Because any neural network learning algorithm is sensitive to the choice of some parameters, there is a need to normalize the coefficients of the robot's dynamics in order to avoid a decrease in the effectiveness of the learning process due to the parameters of a particular robot. An advantage of the normalization lies in making the process of learning independent of a robot's parameters, so that the learning procedure can be treated in quite a general way. The normalization consists in dividing any coefficient of the robot's dynamics by a properly chosen

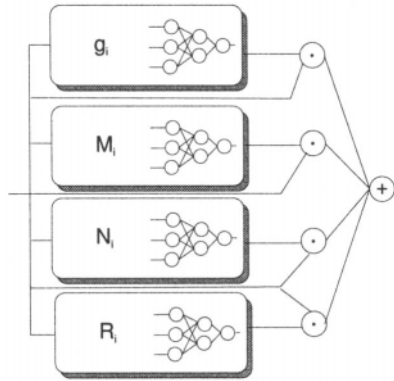


Figure 5.21. Neural network structure for robot dynamics.

constant value, a normalization factor ($NormF$). It is natural to take the value

$$NormF = \max_{i \in I_{coe}} \max_j |a_j^i| \quad (5.76)$$

So the normalized coefficients take the form

$$coe_{norm}^i = \sum_j \frac{a_j^i}{NormF} \sin \left(\sum_k w_{jk}^i q_k + bias_j^i \right) \quad i = 1, \dots, I_{coe} \quad (5.77)$$

5.2.2.1. Calibration of neural model of robot dynamics. Calibration is performed to model properly a real object (robot), using as input data a rough model (e.g., obtained with some simplifying assumption). In our case the calibration process is carried out in two main steps: the active and passive modes of the robot's dynamics learning. The main difference between the two modes lies in the ability (*active mode*) or inability (*passive mode*) to perform active experiments on the robot, i.e., the robot can be controlled by allowable controls.

The active mode of learning requires that one perform experiments on the robot manipulator for which the networks were built. From this step of calibration the real dynamics of the robot with zero payload is obtained. The passive mode of learning is applied to tune the net coefficients while performing a real task in the robotic workcell. It has to be done because of change in the work conditions of a manipulator (i.e., different payloads).

Active mode of learning. The aim of the active mode of learning is to obtain the *real dynamics of a robot with zero payload* (RD0). RD0 is the dynamics of a robot when no external forces/torques act on it and its gripper does not hold any object. A characteristic feature of this mode is that active experiments on a real robot are allowed as well. An active experiment relies on forcing a robot to follow prescribed trajectories and measuring important robotic data during this

process: configurations, velocities, accelerations and resulting torques/forces. The trajectories in the active mode can be chosen freely, but in such a way as to make identification of the robot's dynamic parameters as easy as possible. A trajectory can be shrunk just to a single point $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$ in phase space. The hypothesis for the active mode of learning is as follows:

By choosing appropriate trajectories (points) in phase space, the problem of identifying dynamic parameters of a robot can be decomposed into a set of simpler kinematic problems.

This is of great importance when neural techniques are used to derive the robot's dynamics. Looking back at Equation (5.62), it is easy to observe that many of the parameters depending on the configuration \mathbf{q} are multiplied by velocities or accelerations. The classical algorithm used to train neural networks (back-propagation method) does not work in this case. Other algorithms for training fail as well. By taking advantage of the possibility to choose freely a measuring (identification) point, we can avoid the difficulties. A RD0 is identified in terms of four subsequent submodes:

A. Quasistatic submode:

In this submode $\dot{\mathbf{q}} = \ddot{\mathbf{q}} = \mathbf{0}$, so Equation (5.62) becomes

$$\mathbf{G}_i(\mathbf{q}) = \mathbf{F}_i \quad \text{for } i = 1, \dots, n$$

and gravitational terms can be identified. By performing a set of experiments we get patterns $(\mathbf{G}_i, \mathbf{F}_i)$ ready to be used to train the gravitational net,

B. Constant-velocity submodes:

Two submodes are characterized by a condition of constant velocity. In the viscous friction submode a robot passes a trajectory with velocities chosen to be $\dot{\mathbf{q}}_i = \text{const} \neq \mathbf{0}, \dot{\mathbf{q}}_j = \mathbf{0}, j \neq i, j = 1, \dots, n$. Then

$$\mathbf{R}_i \dot{\mathbf{q}}_i = \mathbf{F}_i - \mathbf{G}_i(\mathbf{q}) \quad (5.78)$$

$$\mathbf{N}_{ii}^j(\mathbf{q}) \dot{\mathbf{q}}_i^2 = \mathbf{F}_j - \mathbf{G}_j(\mathbf{q}) \quad \text{for } j = 1, \dots, i-1, i+1, \dots, n \quad (5.79)$$

This way, viscous friction coefficients and centripetal terms are identified. Let us observe that \mathbf{F} is measured, while $\mathbf{G}(\mathbf{q})$ is computed based on results of the previous submode.

Other Coriolis terms are identified in accordance with the following formula $(\dot{\mathbf{q}}_j, \dot{\mathbf{q}}_k \neq \mathbf{0}, \dot{\mathbf{q}}_l = \mathbf{0}, l \neq j \neq k, i = 1, \dots, n)$:

$$2 \cdot \mathbf{N}_{jk}^i(\mathbf{q}) \dot{\mathbf{q}}_j \dot{\mathbf{q}}_k = \mathbf{F}_i - \mathbf{G}_i(\mathbf{q}) - \mathbf{N}_{jj}^i(\mathbf{q}) \dot{\mathbf{q}}_j^2 - \mathbf{N}_{kk}^i(\mathbf{q}) \dot{\mathbf{q}}_k^2 \quad \text{for } i = 1, \dots, n \quad (5.80)$$

C. Inertia submode:

Having identified $G(q)$, R , and $N(q)$, only $M(q)$ remains to be identified. The identification is easier when $\ddot{q}_i \neq 0$ while $\dot{q}_j = 0$, $j = 1, \dots, i-1, i+1, \dots, n$. In this case the equations of the robot's dynamics can be written as follows:

$$M_{ii}(q)\ddot{q}_i = F_i - G_i(q) - R_i\dot{q}_i \quad (5.81)$$

$$M_{ji}(q)\ddot{q}_i = F_j - G_j(q) - N_{ji}^j(q)\dot{q}_i^2 \quad \text{for } j = 1, \dots, i-1, i+1, \dots, n \quad (5.82)$$

A direct measurement of accelerations is hardly ever possible. Therefore the following procedure to get values for the \ddot{q}_i is suggested: *Rotate a single axis when measuring forces/torques F acting in the joint configuration for q and velocity \dot{q} over an interval of time. The measurement is accurate and easy to perform. Then approximate the value of \ddot{q}_i by examining $\dot{q}_i(t)$ in the interval.* The approximate acceleration value is better than when the ordinary approximation of acceleration is used (division of the differences of velocity at two time points by the time interval between them).

Experiments for the submodes are performed using as input the data for the training neural network which realize the coefficients of Equation (5.62). By choosing proper values for velocities and accelerations, the identification of dynamic parameters has been decomposed into a set of kinematic problems that can easily be solved by applying a standard backpropagation algorithm.

Example 5.2.2. In this example only the active mode of learning is considered. Some tests were performed in order to evaluate the active mode of learning. We distinguish among noncalibrated dynamics, calibrated dynamics, and real dynamics taken as a pattern dynamics. The nominal dynamics (*noncalibrated dynamics*) can be uniquely described by the following set of parameters:

$$\begin{aligned} m_{11} = 9.0 \quad m_{12} = 4.0 \quad h_1 = 1.0 \quad h_2 = 0.3 \quad r_1 = 0.15 \quad r_2 = 0.1 \quad k_1 = 0.45 \quad k_2 = 0.13 \\ m_2 = 22.0 \quad r_3 = 0.15 \quad h_3 = 0.72 \quad l_1 = 0.26 \quad l_2 = 0.4 \quad m_3 = 5.0 \quad r_4 = 0.05 \quad h_4 = 0.5 \\ l_4 = 0.3 \quad op_1 = 0.0 \quad op_2 = 0.0 \quad op_3 = 0.0 \quad a_2 = 0.43 \quad d_3 = 0.15 \quad a_3 = 0.44. \end{aligned}$$

The variation of the nominal dynamics was taken as the real dynamics. The parameters differ by approximately $\pm 5\%$ of their nominal values:

$$\begin{aligned} m_{11} = 9.2 \quad m_{12} = 3.9 \quad h_1 = 1.02 \quad h_2 = 0.29 \quad r_1 = 0.155 \quad r_2 = 0.098 \quad k_1 = 0.46 \\ k_2 = 0.125 \quad m_2 = 21.5 \quad r_3 = 0.155 \quad h_3 = 0.7 \quad l_1 = 0.265 \quad l_2 = 0.39 \quad m_3 = 4.9 \\ r_4 = 0.051 \quad h_4 = 0.49 \quad l_4 = 0.305 \quad op_1 = 0.0 \quad op_2 = 0.0 \quad op_3 = 0.0 \quad a_2 = 0.43 \\ d_3 = 0.15 \quad a_3 = 0.44 \quad R_1 = 0.5 \quad R_2 = 0.2 \quad R_3 = 0.1. \end{aligned}$$

To evaluate the learning process, some indices need to be introduced such as the mean of the difference between an identified coefficient of the neural dynamics

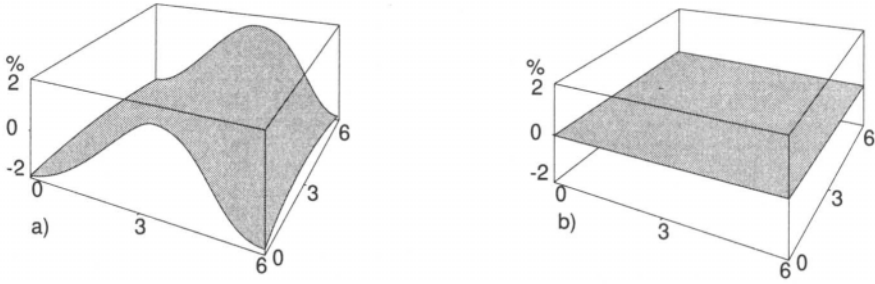


Figure 5.22. The mean error for (a) noncalibrated and (b) calibrated g_{rav}^2 term of the model of robot dynamics.

and its real value,

$$\varepsilon_{\text{mean}} = \frac{1}{\text{Num}} \sum_{i=1}^{\text{Num}} |pr_i^{\text{net}} - pr_i^{\text{real}}|$$

Superscripts net and real are abbreviated forms of neural network and real dynamics, respectively, and pr stands for a particular identified parameter. The chosen test results for the coefficient g_{rav}^2 of the neural-based dynamics model are shown in Figure 5.22.

Passive mode of learning. Although after performing the active mode of learning, the dynamics of a robot with zero payload is known, the coefficients of the robot dynamics depend on the payload parameters. The coefficients can vary when different payloads are carried. So there is a need for tuning the coefficients while a real task is being performed in the robotic cell. In classical robotics, the tuning is performed as an adaptation algorithm (Craig, 1981; Spong and Vidyasagar, 1989). In the approach here based on neural techniques we propose an algorithm to change the weights of the network to react properly to different payloads.

The algorithm is not as simple as in the active mode of learning when, by proper choice of learning sequences, neural network-modeled coefficients were obtained. When doing real work, we can only get information about coefficients by *passive experiments*, i.e., by following trajectories prescribed by a user.

We consider two approaches to identify properly dynamic coefficients, both of which take advantage of the neural representation. In Figure 5.23, the general idea of the approaches is depicted. Index d denotes the desired value of a particular variable. The robot's dynamics will be properly modeled when $q = q_d$, $\dot{q} = \dot{q}_d$, and $\ddot{q} = \ddot{q}_d$. In the case when at least one of the above conditions is not satisfied, some weights of the network have to be changed to satisfy the conditions. The tuning can be performed either by a backpropagation algorithm adapted for our purposes or by a tuning process based on a negative gradient method coupled

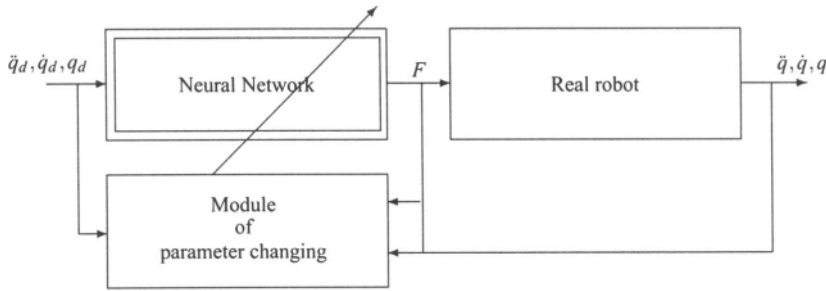


Figure 5.23. Passive training of neural dynamics.

with an optimization procedure for finding the minimum of a goal function along particular lines in parameter space.

PARAMETER TUNING BASED ON OPTIMIZATION PRINCIPLE: The idea behind the method is as follows: when obtaining the nominal dynamics a payload is modeled as well (the robot dynamics with zero payload is just the dynamics obtained by zeroing all the payload parameters, masses, inertia terms, and center of gravity positions), so the information about which parameter of the payload influences which coefficient of the robot dynamics is kept. When an unknown payload is identified, an algorithm modifying the parameter values is applied.

Let us assume that the parameters are independent of each other and call them p_1, p_2, \dots, p_{np} , where np is the number of payload parameters. The algorithm is based on a numerical negative gradient method combined with a procedure for minimizing a goal function along the negative gradient.

The algorithm consists in the following steps:

- Step 1. Read in the parameter space: $\Pi_{j=1}^{np} [p_j^{\min}, p_j^{\max}]$.
- Step 2. Set the iteration counter to zero, $i \leftarrow 0$, and read in the initial value for p^i .
- Step 3. Read in the state (q, \dot{q}, \ddot{q}) and the torques F_{real} acting on the robot.
- Step 4. Numerically determine the negative gradient of the goal function GJ defined as follows:

$$GJ(p) = \|F_{\text{real}} - F(p)\| \quad (5.83)$$

where F_{real} are the measured torques acting on the robot, while $F(p)$ is the output of the neural networks, given state (q, \dot{q}, \ddot{q}) as input.

Obviously the value depends on the current set of parameters p . The negative gradient is determined in a sequential manner. Two values are taken for each parameter $(j = 1, \dots, n)$ to determine the gradient:

$$p^{\text{up}} = (p_1^i, \dots, p_{j-1}^i, p_j^i + \Delta p_j^i, p_{j+1}^i, \dots, p_{np}^i)$$

and

$$\mathbf{p}^{\text{down}} = (\mathbf{p}_1^i, \dots, \mathbf{p}_{j-1}^i, \mathbf{p}_j^i - \Delta \mathbf{p}_j^i, \mathbf{p}_{j+1}^i, \dots, \mathbf{p}_{np}^i)$$

Then $\mathbf{F}^{\text{up}} = \text{Inverse_Dynamics}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{p}^{\text{up}})$ and $\mathbf{F}^{\text{down}} = \text{Inverse_Dynamics}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{p}^{\text{down}})$ are computed by a neural network with weights set according to the value of the parameter vector. The smallest value from the set $\mathbf{F}^{\text{down}}, \mathbf{F}, \mathbf{F}^{\text{up}}$ is found and the appropriate (leading to the smallest values) $\Delta \mathbf{p}_j^i$ is stored. The procedure is repeated for all parameters. As a result a vector of $\Delta^i \mathbf{p}$ is obtained.

Step 5. Along a line in parameter space starting at the point \mathbf{p}^i and passing through the point $\mathbf{p}^i + \Delta^i \mathbf{p}$ a minimization is performed (e.g., golden division method). As a result, a new point \mathbf{p}^{i*} in parameter space is obtained.

Step 6. Set $\mathbf{p}^{i+1} = \mathbf{p}^{i*}$.

Step 7. Check the stop condition:

if $\|\mathbf{F}_{\text{real}} - \mathbf{F}(\mathbf{p}^{i*})\| = \|\mathbf{F}_{\text{real}} - \mathbf{F}(\mathbf{p}^{(i-1)*})\|$, then STOP; the best set of parameters has been found;
otherwise, after incrementing $i \leftarrow i + 1$, goto Step 4.

The metric used in Step 7 can be any metric, although from a numerical standpoint, metrics of types 1 and ∞ are preferable. Step 5 is introduced to allow global search and to penetrate more promising areas of parameter space. The algorithm can be executed for different states (in Step 3 the state was fixed). Any intermediate result (\mathbf{p}^{i*}) can influence the neural network immediately.

Example 5.2.3. In order to examine the method of parameter tuning based on the optimization principle, the following test was performed. Let us assume that in a state $\mathbf{q} = (1, 1, 1)$, $\dot{\mathbf{q}} = (1, 1, 1)$, $\ddot{\mathbf{q}} = (1, 1, 1)$ the robot (Figure 5.20) running till then without any payload grabs a cylinder with a mass of 1.5 kg, height 0.3 m, and radius 0.05 m. The algorithm presented in the previous section, generates the sequence of points in parameter space as depicted in Figure 5.24.

The parameter space was chosen to be

$$op_1 \times op_2 \times op_3 = [0, 2] \times [0, 0.5] \times [0, 0.1]$$

The process of parameter change has the following phases: iteration 1, the starting point; iterations 2–4, establishment of the negative gradient, iterations 5–14, traveling along the negative gradient direction.

The number of iteration can be decreased by finishing computations when the error becomes less than a given minimal value.

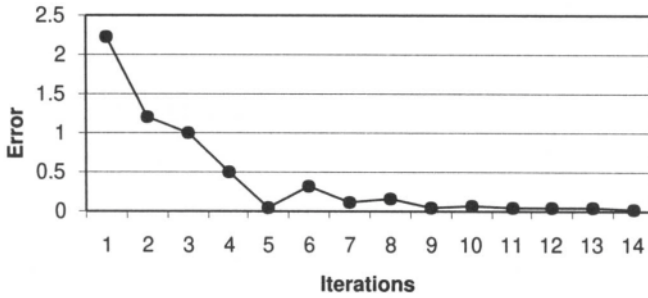


Figure 5.24. Passive learning of a parameter of the dynamics.

NEURAL NETWORK TUNING BASED ON A VARIANT OF THE BACKPROPAGATION ALGORITHM: The other method of performing passive learning applies a slightly modified backpropagation algorithm for training the neural network. In order to apply the backpropagation algorithm, the error must be known for each neural network modeling a coefficient of the robot's dynamics. Unfortunately, the only parameters available to measure on a real robot are the current state (i.e., configuration, velocity, and acceleration of each joint) and torques/forces F acting on the robot. To get the error in terms of F , the current state is entered into the neural network as input data. After computing the inverse dynamics, the network gives an output F_{net} . When $F = F_{\text{net}}$ the robot's dynamics is modeled correctly, otherwise the difference between the two values can be treated as an error.

Then the error, now expressed in terms of F , must be transformed into a set of errors for particular networks. The classical backpropagation algorithm uses the following rule:

- (*Classical backpropagation rule*).

$$\delta_{\text{input}} = w_i \cdot \delta_{\text{output}} \quad (5.84)$$

As the simplest possible rule, it has an obvious drawback. If we add all errors coming from nets and weight them by appropriate terms, we get

$$\sum_i w_i \delta_{\text{input}} = \sum_i w_i^2 \delta_{\text{output}} = \delta_{\text{output}} \sum_i w_i^2$$

and the last expression is usually different than δ_{output} .

The rule should respect the common-sense rule that $\sum_i w_i \delta_{\text{input}} = \delta_{\text{output}}$.

- (*Common-sense rule*). Let us define

$$W_{\Sigma} = \sum_i |w_i|$$

Then the error transformation is governed by

$$\delta_{\text{input}} = \frac{\text{sgn}(w_i)}{W_{\Sigma}} \cdot \delta_{\text{output}} \quad (5.85)$$

It is easy to check that the total error is just the sum of weighted errors coming from the nets. This rule is not unique and any metric can be used.

Being more natural, this rule leads to obvious errors when applied to robot dynamics. Let us consider the branch N_{11}^1 . From theoretical considerations we know that this entry is constant at zero. But the common-sense rule prompts us to use $\delta_{\text{input}} = \text{sgn}|\dot{q}_1^2|/W_{\Sigma} \delta_{\text{output}}$, the value of which is bigger than zero for nonzero velocity \dot{q}_1 .

The next rule tries to avoid such situations:

- (*Proportional excitement rule*). Let us define W_{Σ} as

$$W_{\Sigma} = \sum_i |w_i \cdot o_i|$$

where o_i is the output coming from the i th net; then the error transformation is governed by

$$\delta_{\text{input}} = \frac{\text{sgn}(w_i)|o_i|}{W_{\Sigma}} \cdot \delta_{\text{output}} \quad (5.86)$$

For comparison we define the final rule as:

- (*Null rule*). The error on a net is assumed to be constant and equal to zero.

The existence of many rules is a consequence of the fact that the number of errors on a net is bigger than the number of outputs.

Example 5.2.4. The following test was performed to judge the effectiveness of each rule. For conditions different from the nominal ones $op_1 = 1.5$ kg, $op_2 = 0.05$ m, and $op_3 = 0.3$ m fixed for the time of the experiment, and randomly generated points (q, \dot{q}, \ddot{q}) in configuration space, an error was introduced to the patterns and examined. The error was defined as follows:

$$\epsilon_i^s = |\delta_{ki}^{\text{net}} - \delta_i^{\text{real}}| \quad (5.87)$$

where δ_{ki}^{net} is the value of the error prompt by the k th error propagation rule for the i th neural network modeling the robot's coefficients, δ_i^{real} is the value of the error prompt for the ideal error propagation algorithm, and s denotes the s th experimental item.

The number of experiments is taken as $Num = 100$ and the performance index is defined as

$$\epsilon_i^{\max} = \max_{i=1}^{Num} \epsilon_i^s \quad (5.88)$$

Table 5.7. Mean Errors for the Coefficient

	M_{12}			
Rule	0	1	2	3
M_{12}	0.19522	2.44496	0.12890	0.11390

Results for a given coefficient are summarized in Table 5.7, which compares the classical backpropagation rule (rule 1), the common-sense rule (rule 2), the proportional excitement rule (rule 3), and the null rule (rule 0).

Now it is reasonable to couple symbolic computations with classical mathematical apparatus in order to get domain (robotic)-specific knowledge without performing tedious and very complicated computations. The specialized structure of neural nets can be treated with a known (after performing symbolic computations of the robot's dynamics) and small number of neurons, so the dynamics is easy to implement within a chip with relatively small capacity. The active mode of learning should be performed very carefully because the number of variables decreases significantly (in fact only object parameters remain) and the starting point for the passive mode of learning can be established properly. The most important step in active mode of learning is to detect gravity terms properly. This is because in the next submodes, the gravity terms are frozen [cf. Equations (5.78)–(5.82)] and errors in gravity identification propagate to other components of the robot's dynamics. In fact, the passive mode of learning in static conditions lead to the conclusion that learning based on tuning is more efficient than that based on backpropagation algorithms and has the following advantages:

- It bears more domain specific information (not only the structure of robot dynamics, but also how unknown variables, object parameters, enter into the weights of nets).
- It works well even when an error at a single point in state space is available (just opposite to the backpropagation algorithms).
- As a straightforward consequence of the previous remark one might suspect that an error cannot accumulate while the process of learning is being performed,
- The complexity of the tuning algorithm grows linearly with the number of varied parameters; when the antigradient is established the optimization is performed for one variable function.

The main disadvantage of this method is that a parameter-dependent function needs to be known for any weight depending on the variable parameters.

5.2.3. *Optimal Trajectory Planning Problem*

Because of the nonlinearity and highly coupled nature of the manipulator dynamics, the conventional optimal control approach is complicated and too time consuming for on-line implementation. Instead a two-stage optimization approach has been commonly used to tackle the problem. The first stage involves an off-line trajectory planning model, which yields a time history of the joint angles and joint velocities (as well as joint accelerations) to be followed by the robot's arm to carry out a prescribed task. The time trajectory of robot motion is planned with the objective of achieving minimum cost or minimum time. The second stage is the on-line path tracking problem. This section treats the problem of the first stage, that is, minimum-cost trajectory planning for manipulators with general minimum-cost objectives. Particular solutions for minimum criteria have been found (Shin and McKay, 1986; Shin and McKay, 1985).

Minimum-time task is especially easy to solve because of the existence of a local law for global optimal control (i.e., minimum/maximum acceleration rule (Shin and McKay, 1986)). For a general criterion there does not exist such a law. The trajectory planner receives as input some sort of geometric path description from which it calculates the time history of the desired positions and velocities.

One possible approach to the solution of the stated trajectory planning problem is to apply one of the standard tools of optimal control theory, Pontryagin's maximum principle. Since the trajectory planning problem frequently requires state or mixed state-control constraints, the maximum principle is not usually applicable even if appropriate equations can be formulated. Therefore we take a more intuitive approach to develop optimal planning methods. The early trajectory planning methods presented by Luh and Lin (1986) use a nonlinear programming approach to solve the planning problem. These methods suffer from the deficiency that the manipulator dynamics is not taken into account so that the robot's joint actuators are likely to be underutilized. In more recent publications [e.g., (Shin and McKay, 1986; 1985)] the robot's dynamics is taken into account. Shin and McKay (1985) adopt a dynamic programming approach to the trajectory planning problem. They consider a general form of the joint torque constraints and general minimum-cost objectives such as minimum energy loss. Dynamics programming has the advantage that it is a well-established method and also gives the control law for any point on the curve (path). On the other hand, if it is implemented in the most obvious and straightforward manner (on the discretized phase plane), it uses a large amount of computer memory and time for computation. The computation time also increases quickly as the density of the discretization and hence the solution accuracy are made finer.

For this reason other methods or modifications of existing ones are being sought in an attempt to achieve considerably increased speed. One modification, namely the method of optimal cost trajectory planning, is based on the graph-searching algorithm of this chapter. It is purpose of this chapter to extend the work done

in (Shin and McKay, 1985; Jacak et al., 1992) and present a trajectory planner based on a discrete robot model and heuristic search methods applied to the state graph of the model of the dynamics. Before discussing the proposed method of trajectory planning based on the graph-searching algorithm, I first describe briefly the optimal trajectory planning problem and its standard solutions.

5.2.3.1. Optimal trajectory of motion. For the trajectory planning problem we have to consider the effects of restricting the manipulator's motions to a preplanned collision-free geometrical path which specified by the (collision-free) path planner. The path planner usually yields a sequence of manipulator configurations described in joint space by the joint position vector. A smooth geometric path can be constructed by connecting intermediate configurations with means such as cubic splines. Then the smooth geometric path of the motion is given by a parametrized curve comprising a set of n functions of a single parameter s [Equation (5.58)]. We assume that the joint vector varies continuously with s . Since the parameter s along with the functions q_i completely describe the joint positions, s will be referred to as the "position" variable. The i th joint velocity then becomes

$$\dot{q}_i = \frac{dq_i(s)}{ds} \cdot \frac{ds}{dt} = \frac{dq_i(s)}{ds} \cdot \mu, \quad i = 1, \dots, n \text{ (DOF)} \quad (5.89)$$

where $\mu = \dot{s}$ is the pseudo-velocity of the manipulator. It is also assumed that the derivatives dq_i/ds and d^2q_i/ds^2 exist, and that the derivatives dq_i/ds are never all zero simultaneously. In order to present the dynamic model of the robot the Einstein summation convention is used and all indices run from 1 to n . Plugging this into the known dynamic equation of the robot's manipulator [Equation (5.61)] gives the following equations of motion along the geometric path:

$$\begin{aligned} \dot{s} &= \mu \\ F_i &= M_{ij}(s) \cdot \frac{dq_j}{ds} \cdot \dot{\mu} + M_{ij}(s) \cdot \frac{d^2q_j}{ds^2} \cdot \mu^2 + N_{ijk}(s) \cdot \frac{dq_j}{ds} \cdot \frac{dq_k}{ds} \cdot \mu^2 \\ &\quad + R_{ij} \cdot \frac{dq_j}{ds} \cdot \mu + G_i(s) \end{aligned} \quad (5.90)$$

$$i, j, k = 1, \dots, n$$

where F_i is the torque or force applied at the i th joint. M_{ij} is the $n \times n$ inertia matrix, N_{ij} is the Coriolis force array, R_{ij} is the matrix representing viscous friction, G_i is the gravitational force, and n is the number of robot joints.

Introducing shorthand notation (Shin and McKay, 1985) equations (5.90) can be expressed as follows

$$F_i = M_i \dot{\mu} + Q_i \mu^2 + R_i \mu + G_i \quad (5.91)$$

where the quantities are functions of s .

The goal of automation is to produce at as low a cost as possible. Such cost depends upon the cost of driving the robot, which varies with the robot motion. Let the functional C given by

$$C = \int_0^T L(q, \dot{q}, F) dt \quad (5.92)$$

(L is the objective function and T is the time of motion) describe the cost of the robot motion. Using the parametrized curve, the cost functional can be transformed into

$$C = \int_{s_0}^{s_{fin}} L'(s, \mu, F) ds \quad (5.93)$$

More exactly, the cost of motion along a track is assumed to be

$$C = \int_{s_0}^{s_{fin}} L'(s, \mu, F) ds = \int_0^{s_{fin}} \left(\frac{1}{\mu(s)} \lambda_1 + \lambda_2 \sum_{i=1}^n \|F_i\| \right) ds \quad (5.94)$$

where $\lambda_1 + \lambda_2 = 1$ for general minimal energy motion. For minimal time motion, $\lambda_1 = 1$ and $\lambda_2 = 0$.

Then the minimum-cost control problem can be stated as follows:

Trajectory planning problem. Given a curve in the robot's joint space, a model of the robot's dynamics, and the robot's actuator characteristics, find the set of signals to the actuators that will drive the robot from its initial configuration to the desired final one with minimum cost.

More formally, assume that the constraints of the input torques can be expressed in terms of the state of the robot, i.e.,

$$F \in E(q, \dot{q}) = E'(s, \mu) \quad (5.95)$$

The optimal trajectory planning problem is to find the control $F(s)$ which minimize the functional C of cost (5.93).

In such cases the standard tools of optimal control theory, such as the maximum principle, are not applicable. This problem is solved using a discretization of the parameter s into intervals $[s_k, s_{k+1}]$ where $k = 0, \dots, N$. Assuming that in each interval $[s_k, s_{k+1}]$ the pseudo-acceleration $\dot{\mu}_k = \text{const}$, one can express the pseudo-velocity μ in this interval as a function of s as follows (Shin and McKay, 1985)

$$\mu(s) = \sqrt{\frac{(s_{k+1} - s) \cdot \mu_k^2 + (s + s_k) \cdot \mu_{k+1}^2}{(s_{k+1} - s_k)}} \quad (5.96)$$

where μ_k denotes $\mu(s_k)$ and $\mu_{k+1} = \mu(s_{k+1})$. Given the formulas for the velocity the incremental cost of motion in the interval under consideration can be found using the formula

$$C_k = \int_{s_k}^{s_{k+1}} L'(s, \mu(s), F(s)) ds = C_k(s_k, \mu_k, s_{k+1}, \mu_{k+1}) \quad (5.97)$$

where

$$F(s) = Q\mu^2(s) + R\mu(s) + G + M \cdot \frac{\mu_{k+1}^2 - \mu_k^2}{2(s_{k+1} - s_k)} \quad \text{for } s \in [s_k, s_{k+1}] \quad (5.98)$$

With these formulas at hand it is possible to solve our optimization problem using the *dynamic programming algorithm* (Shin and McKay, 1985).

To use dynamic programming, a grid is set up so that the position parameter s is used as the state variable. Thus a column of the grid corresponds to a fixed value of s , while a row corresponds to a fixed μ value. One starts at the desired final state (the last column of the grid, with the row corresponding to the desired final μ value) and assigns that state zero cost. All other states with position s_{fin} are given a cost of infinity. Once costs have been computed, the dynamic programming algorithm can be applied with respect to the following recursive form:

$$I_k(\mu_k) = \min_{\mu_{k+1}} [C_k + I_{k+1}(\mu_{k+1})]$$

where $I_k(\mu_k)$ is the optimal cost from (s_k, μ_k) to $(s_{\text{fin}}, \mu_{\text{fin}})$.

The standard algorithm starts at the last column. For each point in the previous column one finds all the accessible points in the current column, determines the minimum cost to go from the previous to the current column, and the increments cost accordingly. The point (s, μ) is accessible if there exists a realizable torque $F \in E$ which achieves this point. It can be easily checked by the condition given in (Shin and McKay, 1985). Determining which points are accessible from one column to the next is simply a matter of checking to see if the slope of the curve connecting the two points gives a feasible value. The slope limits can be found from the constraints on the actuator torques. For each point in the previous grid points, the optimal choice of the next grid point is recorded. When the initial state is reached, the optimal trajectory is found by following the pointer chain which starts at the given initial state. The incremental cost is computed for the minimum-cost problem so that a running sum can be kept for the total cost.

5.2.4. Neural Network-Computed Dynamics for Time-Trajectory Planning

The application of the dynamical programming method leads to the analysis of all the nodes of the grid (s, μ) , starting from the node $(s_{\text{fin}}, \mu_{\text{fin}})$. It is easy to

observe that the pair (s_k, μ_k) describes the state of the reduced dynamical system (5.90). Assuming that at the k th step of the discretization the state of system (5.90) is (s_k, μ_k) ($k = 0, 1, \dots, N$), we can reach the next nodes of the grid (s_{k+1}, μ_j) by using the feasible input $F(s)$ on the interval $[s_k, s_{k+1}]$ such that the obtained pseudo-velocity μ_j is feasible at point s_{k+1} , i.e., there must exist $F(s)$ such that $\mu(s_{k+1}) = \mu_j$, where $\mu(s_{k+1})$ is described by formula (5.96). The feasibility of μ_j for the given s_k, μ_k, s_{k+1} is easy to prove by applying the rules proposed in (Shin and McKay, 1985).

Hence, we can establish that each state [node of the grid (s_k, μ_k)] can be expanded into the following set of states (nodes):

$$\text{Succ}(s_k, \mu_k) = \{(s_{k+1}, \mu_j) \mid \mu_j = \mu(s_{k+1}) \text{ and } F(s) \in E'(s, \mu) \text{ for } s \in [s_k, s_{k+1}]\} \quad (5.99)$$

where $\mu(s_{k+1})$ is given by formula (5.96).

The set $\text{Succ}(s_k, \mu_k)$ is called the set of successors of the node (s_k, μ_k) . Such a definition allows one to expand locally the state graph of the discretized system (5.90).

The optimization task (5.93) now can be reformulated into the task of finding the minimum-cost path on the state graph connecting the starting node (s_0, μ_0) with the final node $(s_{\text{fin}}, \mu_{\text{fin}})$. The dynamical programming method can create the full graph of states. To reduce the computational complexity of the problem we propose a method based on methods applied in AI. The method relies on sequentially expanding the state graph of the dynamical system (5.90). Expanding (s_0, μ_0) , successors of (s_0, μ_0) , etc., ad infinitum makes explicit the graph that is implicitly defined by the rules (5.99) of creating a successor set. A graph-search control strategy is a process of making explicit the portion of the implicit graph sufficient to reach the goal node $(s_{\text{fin}}, \mu_{\text{fin}})$. The way of expanding the graph will depend on the form of the cost function and the evaluation function.

As noted previously, the cost of transition from node (s_k, μ_k) to node (s_{k+1}, μ_j) is equal to $C_k(s_k, \mu_k, s_{k+1}, \mu_j)$ defined by formula (5.97). This means that there exists $F(s)$ for which $\mu(s_{k+1}) = \mu_j$ and the cost of an arc directed from node (s_k, μ_k) to node (s_{k+1}, μ_j) is equal to C_k . The cost of a path between two nodes is then the sum of the costs of all the arcs connecting the nodes along the path.

Let the function $w(s_k, \mu_k)$ give the actual cost of the path in the search tree from the start node (s_0, μ_0) to the node (s_k, μ_k) for all (s_k, μ_k) accessible from (s_0, μ_0) , and let the function $h(s_k, \mu_k)$ give the cost estimate of the path from the node (s_k, μ_k) to the goal node $(s_{\text{fin}}, \mu_{\text{fin}})$. We call h the heuristic function.

The evaluation function $e(s_k, \mu_k)$ at any node (s_k, μ_k) consists of the cost of the path from (s_0, μ_0) to (s_k, μ_k) and the estimated cost of the path from (s_k, μ_k) to the goal node.

Our problem can be solved using a more effective bidirectional search strategy as well. The other direction is evaluated by the same procedure with the starting

node $(s_{\text{fin}}, \mu_{\text{fin}})$. For calculating the optimal path we check all points belonging to both direction trees and choose the path on which the sum of the cost functions in both directions is minimal. The bidirectional process expands many fewer nodes than does the unidirectional one. The choice of evaluation function to order nodes in the list *OPEN* critically determines the search results.

5.2.4.1. *The evaluation function e .* As we saw previously, the evaluation function consists of two components, namely

$$e(v) = w(v) + h(v) \quad (5.100)$$

where $w(v)$ gives the cost of the path in the search tree from the start node $v_0 = (s_0, \mu_0)$ for the forward direction [$v_0 = (s_{\text{fin}}, \mu_{\text{fin}})$ for the backward direction] to the node $v_k = (s_k, \mu_k)$, and the heuristic function $h(v)$ gives the cost estimate of the path from the node v to the goal node $v_F = (s_{\text{fin}}, \mu_{\text{fin}})$ for the forward direction [$v_F = (s_0, \mu_0)$ for the backward direction]. The cost function $w(v)$ can be calculated with the help of the cost $C_k(v_k, v_{k+1})$ of transition from node $v_k = (s_k, \mu_k)$ to node $v_{k+1} = (s_{k+1}, \mu_{k+1})$, described by formula (5.97). The cost of path $w(v)$ between the start node and node v is then the sum of the costs of all the arcs connecting the nodes along path,

$$w(s_k, \mu_k) = \sum_{j=1}^k C_j(s_{j-1}, \mu_{j-1}, s_j, \mu_j)$$

The selection of the heuristic function $h(v)$ is critical in determining the heuristic power of the search algorithm.

The heuristic function h . The heuristic function h estimates the cost of movement from the present node $v_k = (s_k, \mu_k)$ to the goal node $v_{\text{fin}} = (s_{\text{fin}}, \mu_{\text{fin}})$ for a forward procedure [or $v_0 = (s_0, \mu_0)$ for a backward procedure]. The heuristic function h is a lower bound of the optimal cost of movement from v_k to v_{fin} (Nilsson, 1980).

An admissible heuristic function is the optimal cost of robot movement, without imposing limitations on its geometric shape, from the state $q_k = q(s_k)$ to the state $q_f = q(s_{\text{fin}})$, i.e.,

$$h(v_k) = \min \int_0^{T_k} L(q, \dot{q}, F) dt \quad (5.101)$$

where $q(0) = q_k$ and $q(T_k) = q_f$. Such a function meets the condition of being a lower bound.

Finding such a function, however, cannot be solved for the general case. Because of this we will show another method for estimating the cost of movement to the goal node which allows us to construct a nontrivial heuristic function. The idea is to use the knowledge about local minima gathered in subsequent steps of evaluating the cost of searching.

Suppose that we are in the node $v_k = (s_k, \mu_k)$ of the forward searching procedure. Let I_k^* denote the minimal current cost of transition from node (s_k, μ_k) to node (s_{k+1}, μ_j) if such a transition is calculated in the expansion process. Otherwise the current cost I_k^* is equal to zero.

By I_{kF}^*/I_{kB}^* we denote the current cost for the forward/backward searching direction. Denote by $C_{i,i+1}$ the current estimation of the cost of the path in the interval $[s_i, s_{i+1}]$; it is given by

$$C_{i,i+1} = \min(I_{kF}^*, I_{kB}^*) \text{ if } I_{kF}^* \neq 0 \text{ and } I_{kB}^* \neq 0 \quad (5.102)$$

Then the heuristic function $h(v_k)$ is defined as follows:

$$\begin{aligned} h(s_k \mu_k) &= \sum_{i=k}^{N-1} I_{iB}^* && \text{for forward direction} \\ h(s_k \mu_k) &= \sum_{i=0}^{k-1} I_{iF}^* && \text{for backward direction} \end{aligned} \quad (5.103)$$

where I_{iF}^* and I_{iB}^* denote the minimal current cost of transition from the s_k layer to the s_{k+1} layer in the forward (backward) search tree if such a transition is calculated during the expansion process. Otherwise it is set to zero.

Observe that such a function meets the condition of the lower bound of the actual optimal cost of searching restricted to current information about the search process. At the beginning, I_{kF}^* and I_{kB}^* are both equal to zero and they become nontrivial during the execution of the procedure.

The construction of the function h requires bidirectional searching.

Fact 5.2.2. *The heuristic function defined in Equation (5.103) is monotonic and lower-bound of real cost of achieving the goal node. While the properties are met the solution is optimal.*

□

□

The heuristic function based on learning allows for an essential reduction of the number of extended nodes slightly worsening the quality of the results. The number of extended nodes can be used as the evaluation criterion of the power of heuristic function, as can the relative deviation of the current value of the criterion function obtained by the proposed graph search from its optimal value obtained by dynamic programming. Tests lead to the conclusion that a heuristic function based on learning allows for an essential reduction of the number of extended nodes and leads to results which differ slightly from optimal. Also, different heuristic functions can be used to improve the power of the algorithm. Such results are presented in (Jacak et al., 1992).

The trajectory planner generates the sequence of optimal pseudo-velocity values, i.e.,

$$(\mu_0 = \mu_0^*, \mu_1^*, \dots, \mu_k^*, \dots, \mu_N^* = \mu_f)$$

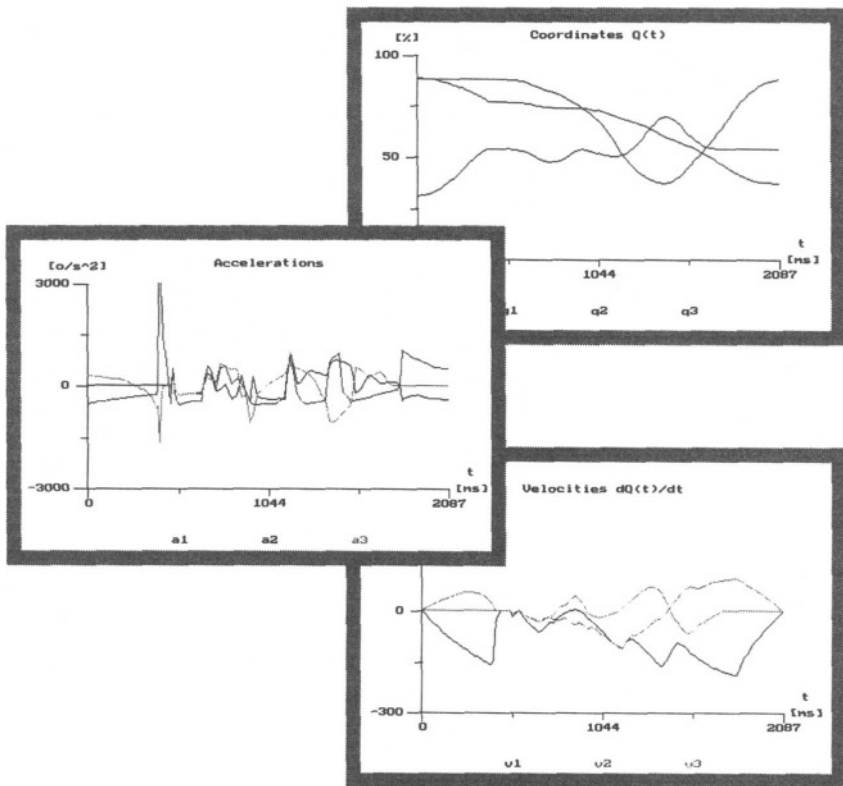


Figure 5.25. Minimum-time trajectory of the first movement along the optimal route \mathbf{p}^* .

for each \mathbf{s}_k for $k = 0, \dots, N$. Based on equation (5.96) it is easy to compute the sequence of optimal pairs (\mathbf{s}_k, t_k) for $k = 0, \dots, N$ which approximates the optimal function $\mathbf{s}^*(t)$. Plugging this function into (5.58) for $q(s)$ gives the optimal time trajectory $\mathbf{q}^*(t)$ of robot motion.

Hence, the trajectory planner yields an optimal trajectory, the time and energy needed for robot movement along each geometrical track. The motion planner is capable of automatically calculating the optimal trajectories with respect to desired quality criteria for all holding movements resulting from the production route as well as the optimal trajectory for all empty movements.

Example 5.2.5. The trajectory planner results shown in Figure 5.25 demonstrate the minimum-time trajectory of the first movement from the optimal route \mathbf{p}^* calculated by the process and path planner for the **Example_Cell**.

To generate automatically the cell-control program, the process planner determine the succession of technological operations. Then, for each robot r servicing

the process, a set of time trajectories of the robot's motions for all transfer operations of a given route is established,

$$TR(r) = \{q_{ij}^r : T_{ij} \rightarrow Q(r) | (i,j) \in \mathbf{Frame_Table}\} \quad (5.104)$$

and

$$T_{ij} = [0, \tau_{ij}] \subset \mathbf{Time}$$

where the trajectory $q^r(t)$ is a function mapping the time interval T_{ij} into the joint space $Q(r)$ of the robot r (Brady, 1986; Jacak and Rozenblit, 1992a). These functions realize the transfer of parts between machines of the cell and, for each part, they ensure that all the operations from *Task* are executed in a preplanned order. The set $\{TR(r) | r \in R\}$ is called the *geometric and dynamic control* of the robotic agents in the flexible workcell. The structure of the motion planner is shown in Figure 5.26.

The time trajectories of motions provide the basis for computing the times of each elementary action. For each motion command, we can change the geometry of movement or change the motion dynamics by selecting criteria for optimal trajectory planning. Variant interpretations obtained from the motion planning allow us to test and select the logical control law p which minimizes the makespan. To calculate the makespan for different sequences of operations and different selection rules, we use the DEVIS simulator, which is described in the Part II of this book.

Consequently, we can test and select the logical control of a cell with respect to different criteria. The intelligent robotic cell planning system is capable of calculating and automatically choosing the best logical control (production route) which minimize the global quality criterion in the form

$$Q(p) = \lambda_1 \cdot \mathbf{Length} + \lambda_2 \cdot \mathbf{Time} + \lambda_3 \cdot \mathbf{Energy} \quad (5.105)$$

Example 5.2.6. In the **Example_Cell** the best route p^* has the following dynamic parameters: length = 14.87m, time of motion = 9.76s, energy of motion = 1837J.

The geometrical route planner completes the description of the CAPP/CAM components of the intelligent autonomous robotic cell CARC.

5.3. Planning for Fine Motion and Grasping

When the workspace is too densely occupied by obstacles, the robot may not attain the goal configuration with the desired precision and orientation. In such

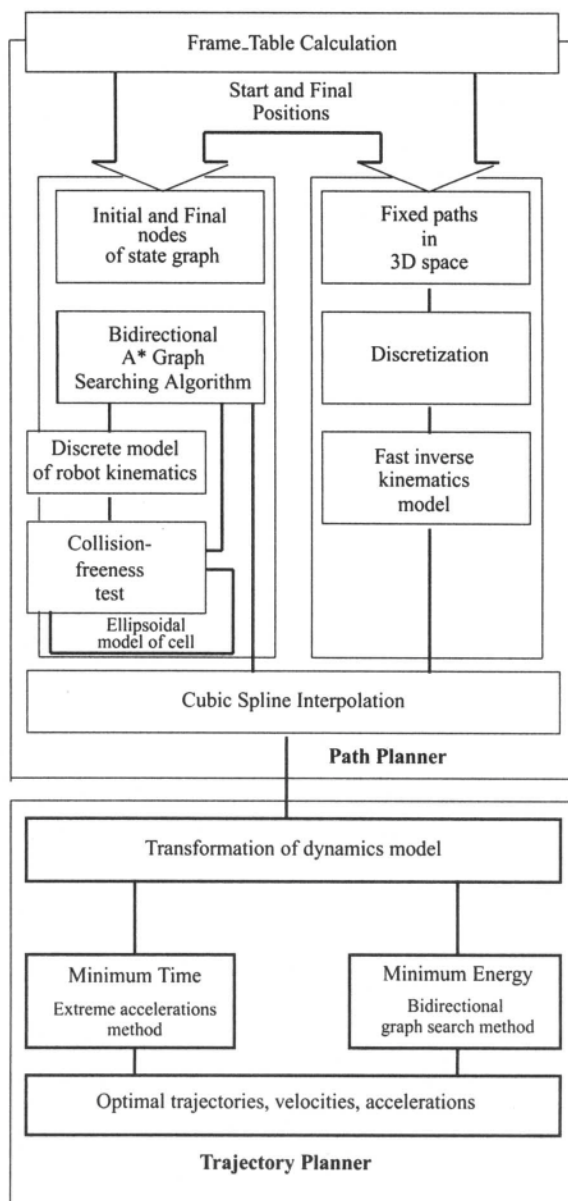


Figure 5.26. Structure of the motion planner.

cases some additional preplanning of the motion path in the goal region and on-line sensing must be explicitly incorporated into the motion plans. Such motion is called *fine motion*. Precise control of the effector end is crucial to accomplishing advanced robot tasks.

5.3.1. Fine Motion Planner

Given two regions IR and GR in the workspace called the *initial region* and the *goal region*, respectively, the fine motion planning problem is to generate a motion strategy whose execution is guaranteed to return success with a final configuration in GR whenever the initial configuration is in IR . The motion between the both regions, called *gross motion*, can be realized less precisely and can be planned using the previous presented planner.

5.3.1.1. Conditional geometric path planning. In both regions it is very important to create a geometric path of motion that satisfies the additional conditions concerning the geometrical relationships between the manipulated object OM and objects S_i ($i = 1, \dots, K$) from its environment. Such a type of path will be called a *conditional geometric path of motion*.

For determining the degree of additional conditions between the object OM and its environment we select a set of l test points from the object (or from the gripper). For the i th test point the motion conditions can be defined as

$$\text{Point-}i : f_i(\rho) = 0 \quad (5.106)$$

where the constraint function f_i is a positive function, i.e., $f_i(\cdot) \geq 0$, and

$$\rho_i = (\rho_i^j | j = 1, \dots, K)^T$$

is the vector of current distances between the i th test point and objects S_j ($j = 1, \dots, K$).

Conditional path planning can be transformed into an optimization problem with two constraints. One is to minimize the length of the path achieving the final position and the other is to satisfy the distance constraints along the path from the beginning of the goal (or initial region).

5.3.1.2. Neural computation-based conditional path planning. These two constraints can be implemented by a neural network representation of the penalty function associated with the i th test point in the following form:

$$\omega_i(p) = \frac{2\varphi_i(f_i(\rho_i)) - 1}{1 - \varphi_i(f_i(\rho_i))} \cdot \phi_i(p, d) \quad (5.107)$$

where $\varphi_i(f_i(\rho_i))$ is the neuron with sigmoid function representing the constraint f_i

$$\varphi_i(f_i(\rho_i)) = \frac{1}{1 + e^{-\beta \cdot f_i(\rho_i)}} \quad (5.108)$$

p is the current position of the i th test point, and $\phi_i(p, d)$ is the neuron representing the goal region GR (d is the range of the goal region and p_g is the center point of the goal region).

$$\phi_i(p, d) = \frac{1}{1 + e^{-\beta \cdot (d - \|p - p_g\|)}} \quad (5.109)$$

We present two methods to solve the conditional fine motion planning problem. The first is based on the artificial potential field approach, and the second on minimizing the energy of motion.

5.3.1.3. Potential field method. Let p be a point in the base coordinate frame E_o . The field of artificial forces $F(p)$ is produced by a differentiable potential function (Latombe, 1991; Lee and Bekey, 1991)

$$U : E_o \rightarrow R^+ \quad (5.110)$$

with forces

$$F(p) = -\underset{p}{\text{grad}} U(p) \quad (5.111)$$

U is constructed as the sum of two elementary potential functions:

$$U(p) = U_a(p) + U_r(p) \quad (5.112)$$

where U_a is the attractive potential associated with the goal position p_g and U_r is the repulsive potential associated with the conditional geometric path.

Attractive potential. We use the parabolic-well attractive field to compute the attractive potential field:

$$U_a(p) = \frac{1}{2} \beta \rho_{\text{goal}}^2(p) \quad (5.113)$$

where $\beta > 0$ and $\rho_{\text{goal}}^2(p) = [p - p_g]^T [p - p_g]$.

Repulsive potential. To model the repulsive potential we use the conditional geometric path potential

$$U_r(p) = \sum_i^l \frac{2\varphi_i(f_i(p)) - 1}{1 - \varphi_i(f_i(p))} \cdot \phi_i(p, d) = \sum_i^l \omega_i(p) \quad (5.114)$$

Using the above relationships which describe the value of the potential field at point p , we deduce the following equation of motion of point p .

$$\frac{d^2 p}{dt^2} = F_a(p) + \sum_i^l F_r^i(p) \quad (5.115)$$

where $F_a(p) = -\underset{p}{\text{grad}} U_a(p)$ and $F_r^i(p)$ are the attractive and repulsive forces for each test point, respectively.

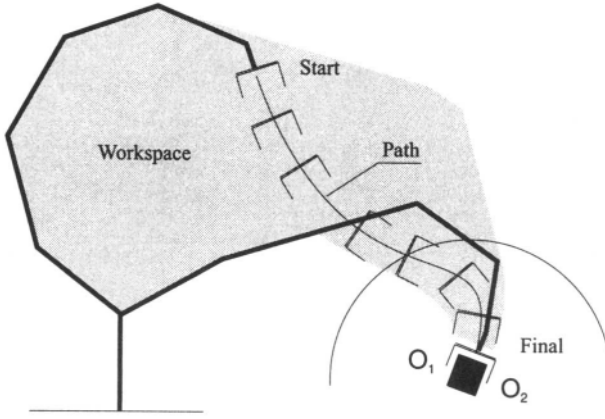


Figure 5.27. Fine motion of a six-DOF manipulator.

Example 5.3.1 (*Problem of keeping a fixed distance*). In experiments with fine motion planning the potential field method was used. The problem we seek to solve is specified as follows:

For a given initial configuration of a six-DOF manipulator on the plane and for a given placement of the object that has to be picked up, find the manipulator track that ensures a good (center) grasp.

Introducing the test point p in the center of the gripper (tool center point) we can define the constraint function f in the goal region as follows:

$$f(p) = (||p - o_1|| - ||p - o_2||)^2 \quad (5.116)$$

where o_1 and o_2 are two points situated symmetrically near the object OM (see Figure 5.27). Assuming the parabolic attractive potential field in the form

$$U_a(p) = \frac{1}{2} \beta \rho_{\text{goal}}^2(p) \quad (5.117)$$

where $\beta > 0$ and $\rho_{\text{goal}}(p) = ||p - p_g|| = ||t_n(q) - p_g||$, the potential field-based path planner is able to produce the desired path. This path can be treated as the input to the inverse kinematics module. In this way the manipulator configurations to achieve a given task are calculated. The result of such a simulation is shown in Figure 5.27.

5.3.1.4. Minimal energy method. The fine motion planning algorithm presented here is based on minimizing the energy of the path lengths and the constraint penalty. Let p represent the reference point from the object OM for which the path is to be planned. Then the energy of the path length is defined as the sum

of the squares of all the lengths of the line segments connecting the route points of motion $p(k)$ for $k = 1, \dots, N$.

The total energy E of the path is defined as

$$E = \sum_{k=1}^N (p(k) - p(k-1))^T (p(k) - p(k-1)) + \sum_{k=1}^N \sum_{i=1}^l \omega_i(p(k)) \quad (5.118)$$

where l is the number of test points. This implies that individual route points should move in directions that minimize the energy.

Based on the gradient of the energy function the path planning algorithm can be implemented as a coupled connectionist network (Lee and Bekey, 1991). It should be noted that the gradient of the total energy is easy to compute based on the output of neurons φ and ϕ .

The sequence of optimal route points represents the effector path in the goal (initial) region and is the input data for the robot movement track planner presented in the previous section.

5.3.2. Grasp Planner

In order to control properly the position and the attitude of a grasped object a suitable set of contact forces must always be applied by the manipulating structure in accord with the following basic specifications: the resultant wrench must be appropriate to the assigned object motion, and the set of applied contact forces must satisfy the appropriate physical constraint.

Because the contacts are not restricted to the end areas on the individual fingers of the gripper, therefore sensing capability on the whole surface is needed.

Distributed tactile elements suitable to give a detailed map of contact areas with relative pressure distribution are necessary in case of large or segmented contact areas on the same finger. If distributed tactile sensors are mounted all over the finger links, the robot can detect any touch between finger and object. Several works have discussed the shape recognition of an object using finger tactile sensors. The design of grasp planning systems has been greatly conditioned by the large amount of sensory information which needs to be acquired and elaborated in real time. This heavy computational burden led to the design of neural network-based grasp planners.

We assume that the fingers of the gripper are equipped with distributed tactile sensors (touch sensors) organized in matrix form.

The object is grasped in a fixed specified sequence, depending on the solid contour. An approximation of the contour of the object can be obtained by sequentially grasping and interpreting of the sensor signals.

For the approximation of the contour it is divided into a constant number, say K , of segments and for each i th segment k_i grip points are chosen. Some of these

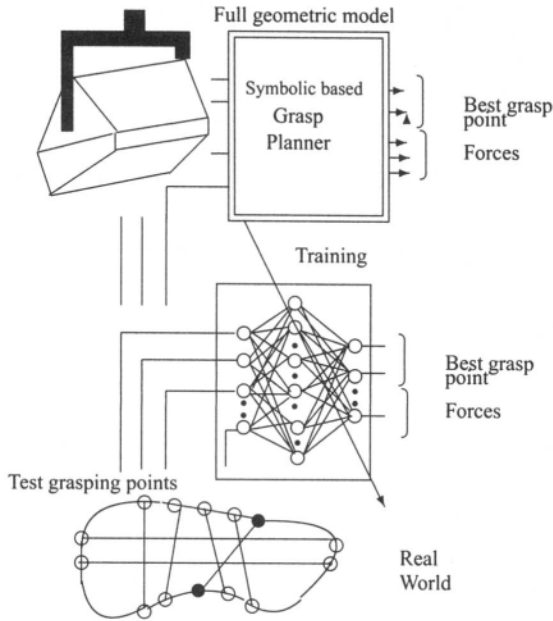


Figure 5.28. Symbolic computation-based training of a neural network for grasping.

points are used for contour approximation. The set of grip points is grouped in pairs.

The signals from the sensors create the input vector for the feedforward neural network which calculates the best pair of points and the grasping forces to be used for grip realization.

5.3.2.1. Symbolic computation-based grasp learning. The training of this network is performed with the simulation approach, using symbolic representation of the object to be grasped.

The object is represented by a geometric model built up through solid modeling techniques. Based on the contour segmentation the k_G pair of grip points is generated. For every pair of grip points we simulate the touch sensor signals for each finger, and so model the *input vector* of patterns.

For a given pair of grip points a mathematical analysis of grasping forces is performed and the feasibility of the forces is tested. The best pair according to the criterion of minimal grasp force is used as the pattern *output* of the neural network together with the generated grasp forces. The idea of such a grasp planner is shown in Figure 5.28.

The objects considered in the learning phase are sums of convex polyhedra. In addition, the physical characteristics of each object, i.e., mass, coefficient of

friction μ , and the stiffness R , are given.

We assume that the object is placed on a table and a two-finger parallel jaw is used to pick it up. Each j th link of a finger is equipped with an $l \times k$ touch sensor matrix TS_j with the values 0 or 1.

Before analyzing feasible grip points using mating surfaces, all the forbidden surfaces which the gripper cannot reach should be avoided. Since initial and final grasp configurations are known beforehand, this can be done by analyzing the geometrical constraints of the object and the environment with respect to the gripper constraints. The analysis is based on two criteria. Initially, feasible gripping points are found by analyzing the geometrical characteristics of the object to be picked up and the gripper's geometry. Then one finds gripping points such that the object is in equilibrium in the gripper using the given forces, and then one obtains the optimal gripping points by analyzing the rigidity of the object with respect to the gripper.

5.3.2.2. Optimal grasping forces. Most authors (Wolter et al., 1985; Lozano-Perez, 1981; Gatrell, 1989) find grip points by considering the rotational and slippage effects when the gripper forces are applied against the surface of the object. Though this approach seems to be an extension of the work of (Wolter et al., 1985), the analysis of the stability of the object in the gripper is considered using concepts described in (Kumar and Waldron, 1991). As in many applications, the gripper is made of metal and the gripping points are relatively point contacts, and by simply verifying the rigidity of the object we find effective optimal gripping points.

The object is considered as the sum of convex polyhedra. An object OM to be grasped and moved has an intrinsic coordinate system with respect to which points in it can be specified. The physical characteristics of each object are its mass, coefficient of friction, and rigidity.

The grasping activity is analyzed assuming a two-finger parallel jaw and that the object to be grasped stays on the workstation serviced by the robot. The solution uses a the top-down structure. The problem can be divided into two subproblems.

- A. Find feasible grip points using the geometrical characteristics of the object, gripper, and the workstation serviced by the robot.
- B. Find the best pair of grip points for each pair of surface points of the object by analyzing the stability and rigidity of object with respect to the gripper.

Selecting feasible gripping points. All possible gripping points can be analyzed using the geometrical characteristics of the object, gripper, and workstation serviced by the robot.

This analysis is carried out by first finding all possible gripping surfaces of the object. Before finding these surfaces, excluded surfaces should be avoided.

UNREACHABLE SURFACES: Let \mathcal{S}_a be the set of all of the surfaces of the object OM , i.e., $\mathcal{S}_a = \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$. All the surfaces which the gripper cannot reach are called *excluded surfaces* (Feddema and Ahamed, 1985). There are two types of excluded surfaces, unreachable and coexcluded surfaces. Unreachable surfaces are those already in contact with other surfaces from the environment at the initial position, and coexcluded surfaces are parallel for the unreachable surfaces for the proposed parallel jaw.

From the set \mathcal{S}_a we eliminate the subset \mathcal{S}_f of surfaces already in contact with other surfaces from the environment' and all surfaces nearly parallel to each of the surfaces of \mathcal{S}_f .

After analyzing excluded surfaces, all the grip points can be found using individual surfaces that belong to the set of feasible surfaces.

SELECTION OF MATING SURFACES OF THE GRIPPER: Based on the "grasp view" (approach view of the object, typically top view or side view) the contour of the object is established. In the next step we select a pair of surface elements which connect to the contour and are nearly parallel, facing away from and opposite to each other. Each such pair determines an orientation vector. This can be easily analyzed using the symbolic surface equation

$$Ax + By + Cz + D = 0 \quad (5.119)$$

The vectors A, B, C have unit length and point outward from the face.

Each pair of faces belonging to the contour passes through a series of filters which eliminate geometrically infeasible pairs.

To ensure that two faces are parallel within an angle θ_e , the angle between normal vectors must be within the range $[(\pi - \theta_e), (\pi + \theta_e)]$.

Selection of contact points of the robot gripper. After selecting a pair of surfaces, specific grip positions must be chosen on these surfaces. In order to ensure that a grip can be found in as many run-time situations as possible, a diverse set of grips is proposed. These vectors are selected using the raster method. This is done by selecting contact points from all pairs of surfaces. The selection of a pair of contact points is done starting from the middle point of the chosen face. In this way, various points are selected along all symmetries of the face and along the edges of the face. Each point is analyzed with appropriate point from the other surface which is nearly parallel to the chosen face. The selected points are grouped according to the fixed segmentation of the "grasp contour" of the object. Additionally, the simulated values of the sensors signals are stored.

For each surface two types of vectors are found: one parallel to the considered edge on a specific surface, the second perpendicular to the surface on which the position vectors for gripping will be selected. This pair of gripping points should lie on the plane formed between these two points and the resultant force vector in order to avoid a rotational effect in the initial configuration (Figure 5.29).

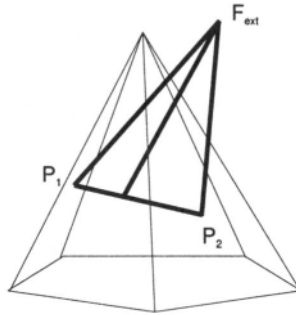


Figure 5.29. Grasping plane.

Calculation of optimal grip points. From the feasible grip points, semioptimal grip points can be obtained by analyzing the stability of the object with respect to the gripper, considering the forces distributed between them and the rigidity of the object. The best (optimal) pair of grip points for each pair of mating surfaces can be obtained by finding the maximum angle between the position vector of each grip point with respect to the resultant external force of the object.

STABILITY OF THE OBJECT IN THE GRIPPER: Gripper-object contacts are modeled as point contacts, which means that the gripper can apply any three force components, but no moments. A quasistatic approach to the problem has been adopted which considers the resultant of the inertial forces and moments on the object, and all external forces and moments excluding the grasping forces are always balanced by the grasping forces. The contact forces of these grip points can be divided into two types of forces: *equilibrating and interaction* forces:

$$\mathbf{F}_i = \mathbf{F}_{iE} + \mathbf{F}_{iI} \quad (5.120)$$

Equilibrating forces are forces required to maintain the object in equilibrium without squeezing it, while the interaction forces squeeze the object with zero resultant force. The interaction forces are defined by the geometry in the two-fingers grasp, as the component of the difference of the contact forces projected along the line joining the two contact points (Cutkosky, 1985). Thus the object-gripper system is required to maintain the object in equilibrium while the interaction forces ensure that the friction angle at each contact point is within allowable limits.

For further calculation, we denote by \mathbf{r}_i the position vector of i th grip point, by n the number of grip points (in this case $n = 2$), by \mathbf{F}_i the grasping force at the i th contact point, and by O the center of n contact points.

The equilibrium equation for the grasped object may be written as follows:

$$\mathbf{G} \cdot \mathbf{F} = \mathbf{w} \quad (5.121)$$

where w is the 6×1 external force vector consisting of the inertial forces and torques and the weight of the object, F is the unknown $3n \times 1$ contact force vector, and G is the $6 \times 3n$ coefficient matrix. G is analogous to the Jacobian matrix encountered in the kinematics of the serial chain manipulator. Each column vector is a zero-pitch screw through a point contact in screw coordinates. If Ax_i , Ay_i , and Az_i are zero-pitch screw axes parallel to the X , Y , and Z axes through the i th contact point, then

$$G = [Ax_1, Ay_1, Az_1, \dots, Ax_i, Ay_i, Az_i, \dots, Ax_n, Ay_n, Az_n] \quad (5.122)$$

EQUILIBRATING FORCES: It has been shown (Cutkosky, 1985; Kumar and Waldron, 1989) that the pseudo-inverse solution for the force system can be found through an equilibrating force calculation. In this case w always belongs to the column space of G . The pseudo-inverse, then, is a right inverse which yields a minimum norm solution,

$$F = G^+ \cdot w \quad (5.123)$$

Since the solution vector must belong to the row space of G (Noble, 1976), we have

$$F = G^T \cdot c \quad (5.124)$$

where c is a 6×1 constant vector.

If F_{iE} is the equilibrating force at the i th contact point and c_0 and c_1 are two 3×1 vectors such that $c = [c_0, c_1]^T$, then

$$F_{iE} = c_0 - c_1 \times r_i \quad (5.125)$$

where r_i is the position vector of the i th contact point.

The above equilibrium equation may be rewritten as

$$\sum F_{iE} = Q \quad \text{and} \quad \sum (r_i \times F_{iE}) = T \quad (5.126)$$

where Q and T are external force and moment components of the load vector w .

INTERACTION FORCES: The interaction forces are determined by the geometry. They are equal and opposite along the line (e) joining two grip points as shown in Figure 5.30.

The problem of determining the forces at grip points may be decomposed into two subproblems:

- I. Determination of the equilibrating forces F_{iE} required to maintain the equilibrium of the gripped body assuming that the finger interaction forces are absent.
- II. Determination of the interaction forces F_{iI} needed to produce F_i without violating the constraints.

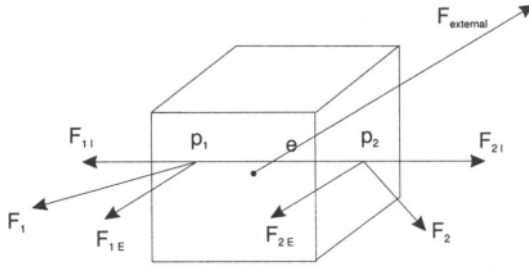


Figure 5.30. Interaction forces.

STABLE GRASP: If μ is the coefficient of friction and F_{\max} is the maximum allowable grasping force, the following two conditions should be satisfied for a stable grasp.

1. **Friction condition:**

$$\frac{F_i \cdot n_i}{\|F_i\|} \geq \cos(\tan^{-1}(\mu)) \quad \text{for } i = 1, 2 \quad (5.127)$$

2. **Maximum force condition:**

$$F_{\max} \geq F_i \cdot n_i > 0 \quad (5.128)$$

If n_1 and n_2 are unit normal vectors and ϕ_1 and ϕ_2 are the angles between the resultant contact force vector and the normal on each contact point respectively, the necessary and sufficient condition for producing a stable grasp is

$$\phi_i \leq \tan^{-1}(\mu) \quad \text{for } i = 1, 2 \quad (5.129)$$

where μ is the friction coefficient.

In order to obtain a stable grasp, the valid range of interaction force factor α must be chosen such that

$$F_i(\alpha) = F_{iE} + F_{iI} \quad (5.130)$$

satisfies the inequalities (5.127) and (5.129).

If F_{iI} is the interaction force, then it should satisfy the following equality:

$$F_{iI} = \alpha \cdot E \quad (5.131)$$

where E is a constant unit vector.

Hence, the force is a function of α , i.e.,

$$F_i(\alpha) = F_{iE} + \alpha \cdot E \quad (5.132)$$

3. Deforming condition:

Assuming that the gripper is sufficiently rigid in order to grasp any object, we evaluate the rigidity of the object. The rigidity of any material can be evaluated by

$$R = \frac{F \cdot L}{\delta s \cdot A_s} \quad (5.133)$$

where A_s is the cross-sectional area, L is the length over which the force is applied, and F is the force applied in the specified cross-sectional area δs .

Evaluating the rigidity condition gives the limit on the range of interaction force from the above stability condition.

$$F_i(\alpha) \leq \frac{R \delta s \cdot A_s}{L} \quad (5.134)$$

where R is the rigidity of the material used to construct the object.

The force F_{\max} depends on the hardness of the gripper and the material of the object. If the above conditions are satisfied, then the gripper can take the object at the contact points without deforming the object's shape. Determination of the gripping forces can be realized using a helicoidal velocity field. The interaction forces should be chosen in such a way that the grasp forces for a given pair of contact points $F_i = F_{Ii} + F_{Ei}$ are maximal inside a friction cone. This depends on the proper choice of the force factor α .

BEST INTERACTION FORCE FACTOR α : The best α is that for which the angle formed between the normal n and the resultant contact force vector F_i at the considered grip point is minimum. To find such an α , it is necessary to calculate the first derivative of the following criterion:

$$V(\alpha) = \frac{F_i \cdot n_i}{||F_i||} \quad \text{for } i = 1, 2 \quad (5.135)$$

Local extreme values of the derivative of this expression give the α whose F_i forms the minimum angle with the normal n .

If this value lies inside the range of α obtained from the stability condition, it is considered the best choice. Otherwise we must consider extreme α values in order to find the best α for the smaller angle formed between F_i and n . It is obvious that sometimes two different values of α are given for two-finger grasping.

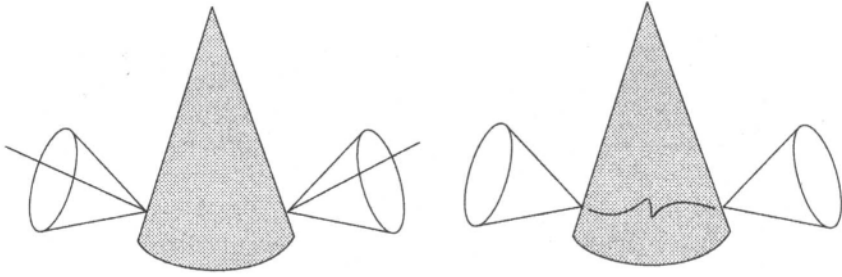


Figure 5.31. Contact forces for a cone. Left: Mass = 2 kg, friction coefficient = 0.5, rigidity = 0.7; best $\alpha = 15.765$. Resultant contact forces: $F_1 = 18.6 \text{ N}$, $F_2 = 18.6 \text{ N}$. Right: Mass = 2 kg, friction coefficient = 0.5, rigidity = 0.3; best α does not exist. Resultant contact forces: F_1, F_2 do not exist.

Example 5.3.2. The analysis of grasping forces for a cone-shaped object is considered without selecting the best choice for grasping, to show how the resultant contact force is increased along the edge when the distance between contact points is decreased. Based on the rigidity concept, the force is inversely proportional to this distance. It can be shown that no grasping exists when the distance between two contact points is too large. We analyze two cases:

- Case 1. A cone with mass = 2.0 kg, friction coefficient = 0.5, and rigidity = 0.7.
- Case 2. A cone with mass = 2.0 kg, friction coefficient = 0.5, and rigidity = 0.3. The forces are shown in Figure 5.31.

This example shows that, for the contact points in Figure 5.31, when rigidity decreases it is impossible to grasp the object because the maximum grasping force decreases and therefore the interaction force factor does not lie inside a valid range obtained from the equilibrium conditions. In other words, when the rigidity decreases and when the distance between the pair of grasping points increases, the object tends to be deformed.

BEST GRASPING FORCES: The best solution is found for the largest angle formed between the resultant contact force and the net external force vector acting on the object for each pair of grip points such that the stability condition is not violated and there is no collision with nearby objects during the movement of the object to its destination. In this form, we can obtain a pair of grip points for every pair of surfaces of any object.

Training pattern preparation. The method presented above finds the optimal solution for two-finger grasping and also finds the best choice for grasping at each pair of nearly parallel surfaces of the object. The analysis is performed for all chosen contact points, and for each pair of points the grasping forces F_i are

calculated. From all points we choose a pair for which the grasping forces are minimal. The best grasping point is calculated as

$$Point_k \rightarrow F_k = \min\{F_i | i = 1, \dots, k_G\} \quad (5.136)$$

Based on the above grasp planning algorithm, the pattern pair for neural network training is taken as the following set:

$$Patt = ((TS^j | j = 1, \dots, k_G), (Point_k, F_k)) \quad (5.137)$$

The symbolic calculation-based grasp planning process is performed for many fully modeled virtual objects. The best grasp points and model signals from finger sensors create the pattern set using to train the neural network.

CHAPTER 6

CAP/CAM Systems for Robotic Cell Design

There is currently a great deal of research and development effort aimed at integrating CAD/CAP/CAM systems and generating production process robot and machine programs using graphics simulation (Prasad, 1989; Bedworth et al., 1991; Faverjon, 1986; Speed, 1987; Ranky and Ho, 1985; Wloka, 1991).

In this chapter we give a brief description of the CAD/CAP/CAM systems **ICARS** (*intelligent control of autonomous robotic system*) and **HyRob** (*hybrid calculation-based robot modeler*), experimental process engineering environments, which provides many of the automatic planning and simulation tools for the synthesis of manufacturing processes and their control in CARC (Jacak and Rozenblit, 1992a; Jacak, in press).

ICARS and HyRob are complete structured software systems which, based on the specification of technological tasks, technical data, and cell models, allow the *development, design, programming, execution, and testing* of logical and geometric control of the machining process in a flexible robotic workcell as well as the fuzzy-neural control of individual robotic agents.

6.1. Structure of the CAP/CAM System ICARS

Possible robot and machine actions are modeled in ICARS in terms of different conceptual frameworks, namely logical, geometrical, kinematic, and dynamic. ICARS, which enables automatic generation of logical and geometric control of manufacturing cell, has a modular structure consisting of three basic modules:

1. *Process sequencing module (Taskplan)*
2. *Graphics modeling and simulation module (Grim)*
3. *Optimal and collision-free motion planning module (Groplan)*

Process planning in the **Taskplan** (*task planner*) module is carried out based on description of the technological operations of the underlying machining process, of the manufacturing cell and its resources such as devices, robots, stores, or fixtures, and of the precedence relation over the set of technological operations.

Task specification data are used to synthesize a geometrical model of the cell (*virtual workcell*) and simulate it in the 3D graphics simulation module **Grim** (*graphic representation of industrial-robot motions*).

The software of the **Grim** system enables the geometric modeling of each object of the cell and planning of the cell layout. Moreover, **Grim** can simulate different robots with direct/inverse kinematics and their dynamics and provide a “teach-in” method in an on-line mode. These models are imported from the HyRob system, which enables the design of any robot manipulator with neural-based kinematic and dynamic models obtained in an automatic way.

In this implementation the graphic simulation system is presented using the example of IRb ASEA and Adept robots. **Grim** allows 3D graphical display of the robot and its movement with free choice of the viewpoint, including zooming and the performance of the collision tests.

Such a robot simulation makes it possible to test the realization of the technological process and estimate the geometrical parameters of each transfer operation.

The process planner (**Taskplan**) then uses the geometrical data from **Grim** simulator for *logical control (production route)* planning. The basic problem here is the derivation of an ordered sequence of robot actions that can be used to perform the task. It is solved by finding an ordered and feasible sequence of technological operations which can be transformed directly into a sequence of robot actions. Such a fundamental action plan for a technological task determines the programs of robot manipulations which service the process. Every program is a sequence of motion and grasp actions. The interpretation of the fundamental plan (logical control) is carried out using an automatic motion planning approach in which the detailed paths and trajectories are specified using both gross and fine motions.

The main unit of the ICARS modules is the motion planner **Groplan** (*geometrical route planning*). The motion planner automatically creates variants of collision-free time trajectories of the manipulator movements which execute the individual robot actions. Such a planner is based on robot-dependent planning techniques and a discrete dynamical system formalism.

The **Groplan** unit automatically generates collision-free geometric paths of robot movements for the whole production route obtained from the **Taskplan** system. Moreover, **Groplan** allows one to optimize the dynamics of motion along paths with respect to time or energy criteria. Each of the planned movements is transferred automatically to the simulator **Grim**, which performs 2D and 3D animated computer simulation of cell actions. Automatically generated time trajectories of robot motions can be used as geometric control routines of a robotic cell when they are compiled with a postprocessor into a specified control code. The detailed ICARS system architecture is shown in Figure 6.1.

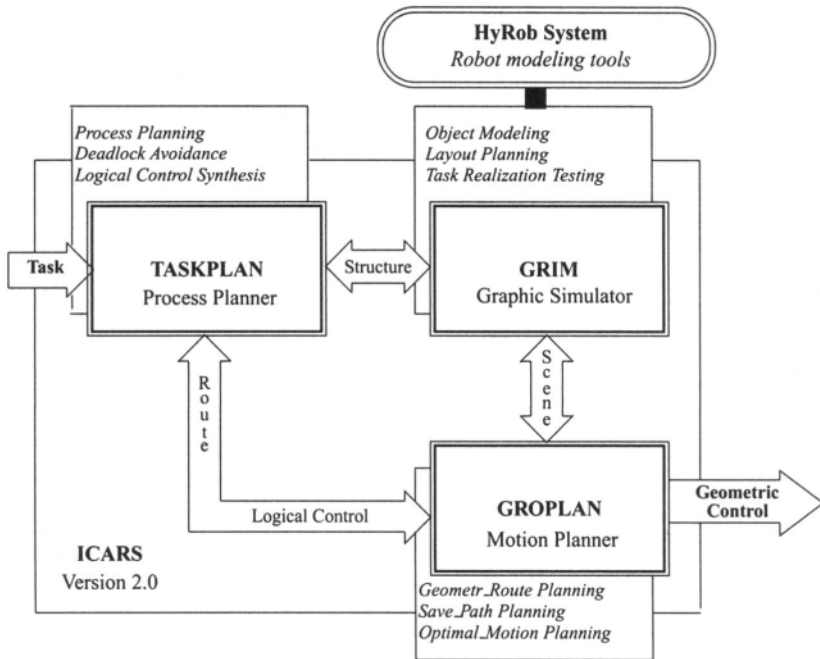


Figure 6.1. Structure of the CAP/CAM system ICARS.

6.2. Intelligent Robotic Cell Design with ICARS

6.2.1. Task Specification Process

To begin the design of an intelligent robotic workcell with ICARS, one specifies the technological task to be realized by the cell being designed. The technological task consists of a finite set of machining operations, an operations precedence relation, and operations allocation relation (Definition 4.1.1). Moreover, the workcell entity structure needs to be created.

The first module of ICARS (**Taskplan**) allows the description of the technological task and cell structure in standard form. **Taskplan** includes a special editor that enables the easy specification of each task and cell components. The logical names of operations (A, B, C,...) and cell devices [such as machines (d01, d02,...), stores (m01, m02,...), robots (r01, r02,...)] are automatically assigned by the **Taskplan** editor. The sample display of the **Taskplan** editor package shown in Figure 6.2 demonstrates the operations precedence relation.

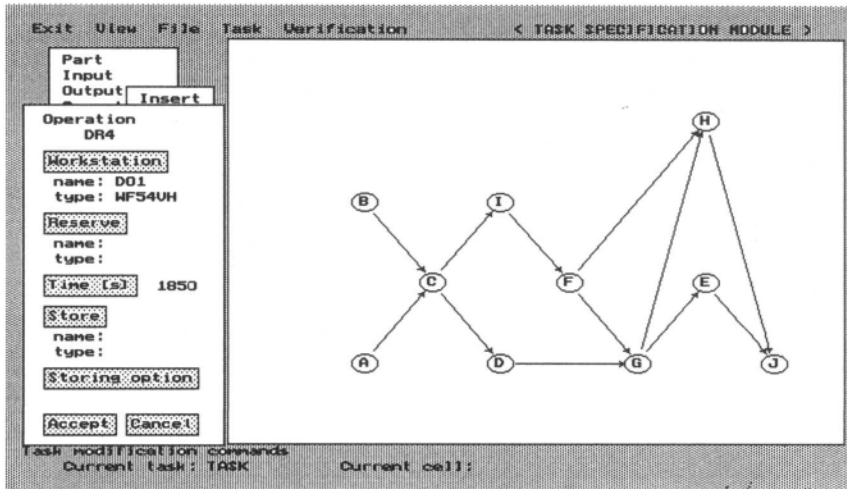


Figure 6.2. Example of an input screen of the Taskplan editor — the operations precedence relation.

6.2.2. Cell Geometry Modeling and Design

Task and cell specification data are used to synthesize a geometric model of the workcell (*virtual workcell*) in the graphics modeling module **Grim**. The software of the **Grim** module enables the geometric modeling of each object of the workcell and planning of the workcell layout.

6.2.2.1. Workcell object modeling. Complex objects such as technological devices, auxiliary devices, or static obstacles, are composed of solid primitives such as cuboids, pyramids, regular polyhedra, prisms, and cylinders. The main **Grim** menu allows the operator to load primitives from a catalog into the object's own base frame and rotate and transfer them for complex object composition. The geometric models of the objects can be stored in an open catalog.

6.2.2.2. Workcell layout modeling. The objects can be placed in a robot's workscene in any position and orientation. The transformations between the object frames and the base frame are performed automatically.

Grim allows the user to transfer the model of an object from the catalog to the workscene (edited on screen) and to locate it in the desired position and orientation with a user-friendly screen management system.

Grim presents a 3D graphic display of the robot and its environment with a free choice of the viewpoint, including zoom, and of a window system (up to four windows each with its own viewpoint and zoom). Figure 6.3 shows examples of models of the object and the workcell layout, created with **Grim**.

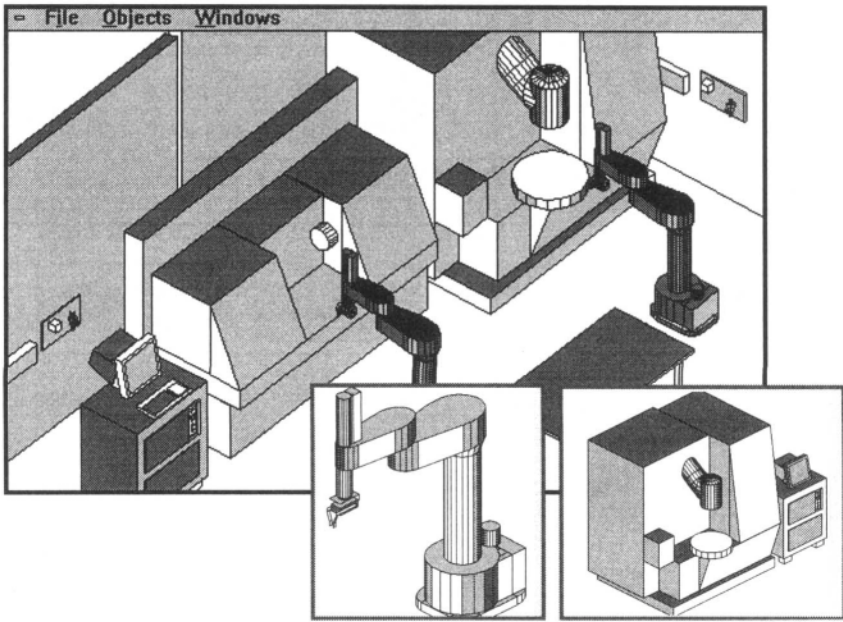


Figure 6.3. Model of a virtual object and of the cell layout.

6.2.3. Route Planning

Taskplan package of ICARS automatically sequences operations based on the task and cell descriptions, including allocation, specification of machining parameters (processing time), partial ordering of operations, and the logical structuring of a cell. **Taskplan** creates optimal sequences of machines (there can be more than one solution) called production routes along which the parts flow during the machining process. Figure 6.4 shows the result of the process planning, i.e., a sample output screen of **Taskplan**, and an example of the calculated optimal logical route.

6.2.4. Geometrical Route Planning and Programming

The ICARS module **Groplan** automatically generates geometric paths of robot movements for the whole production route obtained from the **Taskplan** system. Moreover, **Groplan** allows one to optimize the dynamics of motion along paths with respect to time or energy criteria. **Groplan** can simulate robots with direct/inverse kinematics and their dynamics and provide a teaching method in an on-line teach-in mode. The different robot models are imported from the HyRob system. The standard version of ICARS is equipped with geometric, kinematic,

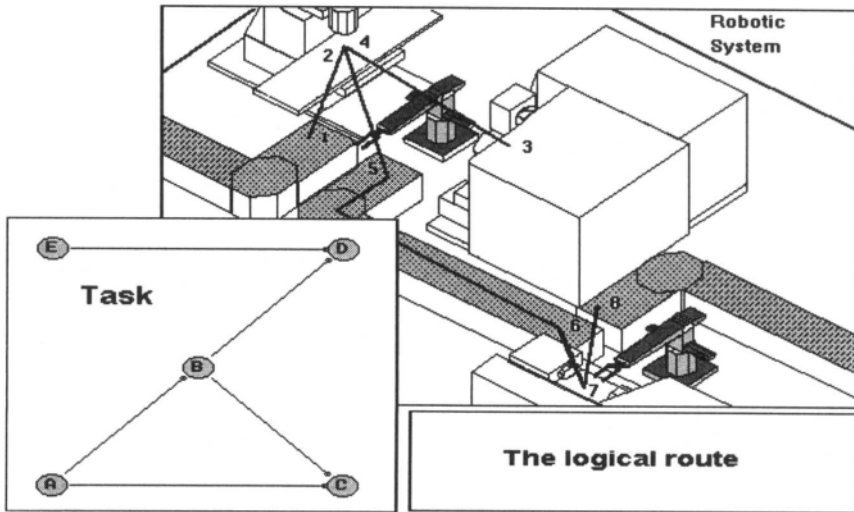


Figure 6.4. Output screen of Taskplan — the production route.

and dynamic models of IRb ASEA and Adept robots. The **Groplan** module consists of two packages: a path planner package and a trajectory package.

6.2.4.1. Path planner package. Based on the graphic model of the cell, the path planner allows the operator to establish precisely the position and orientation of the robot's effector end at each location of the machine's buffer or store. The operator can place the robot manipulator in the desired position on the screen and can move the robot's effector end in Cartesian base space as well as the robot's skeleton in the internal joint space. For each robot position a collision test is automatically performed.

The **Groplan** path planner facilitates motion programming and planning for any two given points along the following paths:

- straight line in Cartesian base frame
- circle in Cartesian base frame
- linear path in the joint coordinate frame
- linear path in the cylinder coordinate frame
- automatic collision-free path

The sequence of movements can be stored as an off-line programmed robot motion program.

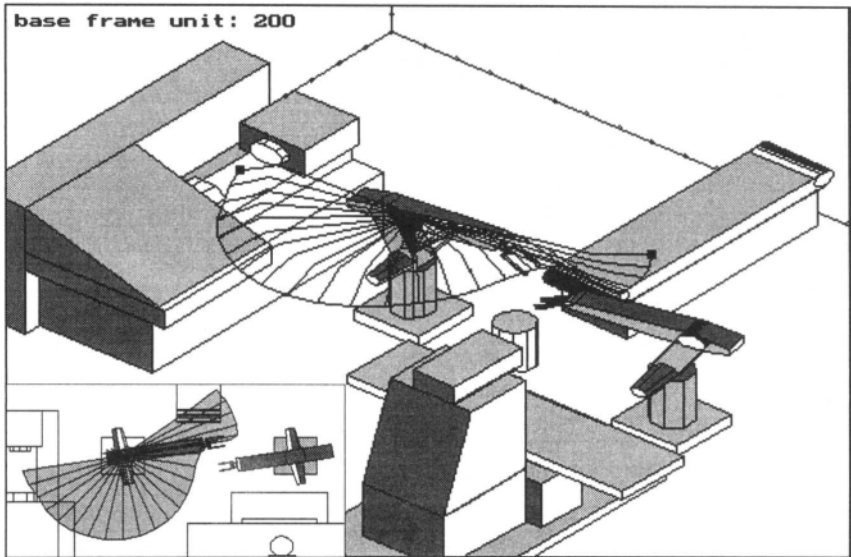


Figure 6.5. Collision-free track of a robot movement.

Groplan offers *automatic collision-free path planning* for all motions of a cell's robots with respect to a given production route, as well as for the one selected movement.

Each planned movement is transferred automatically to the graphics simulator, which allows the demonstration of each path of an effector's movement by computer animation in a 3D representation of the cell layout. Additionally, **Groplan** enables the display of geometric *tracks of the skeleton* or the whole *track of the robot's body* for all motions on the geometric route.

The package offers the possibility of computer animation for the whole geometric route or only for the selected movement. The sample display of the path planner package shown in Figure 6.5 demonstrates the collision-free geometric track of one selected robot movement.

6.2.4.2. Time-trajectory planning. The **Groplan** trajectory planner optimizes the motion dynamics with respect to *time* or *energy* criteria. Moreover, the dynamic parameters of motion along a given path, including the length of effector motion, the time of motion, and the global energy of motion, are calculated and displayed. The graphics editor of **Grim** allows the 2D display of such function graphs as the following:

- the optimal joint trajectories
- joint velocities

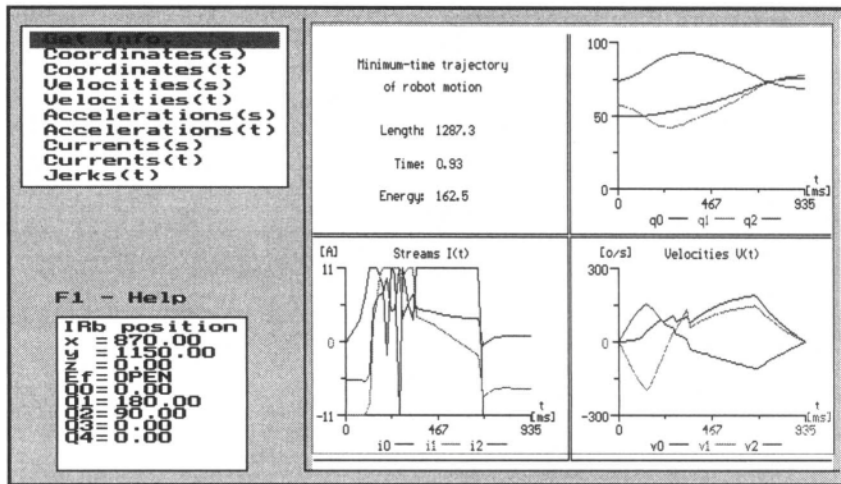


Figure 6.6. Multiwindows screen of the trajectory planner package.

- joint accelerations
- drive currents
- joint jerks

Figure 6.6 presents the sample display of an optimal trajectory. Automatically generated time trajectories of robot motions can be used as geometric control routines of a robotic workcell. Moreover, **Groplan** can calculate and automatically choose the best production route (logical control of the cell) which minimizes a global quality criterion.

6.3. Structure of the HyRob System and Robot Design Process

The structure of the HyRob system is shown in Figure 6.7. The kinematics and dynamics modelers are the main parts of the HyRob system. When starting a new project, the modules of the kinematics and dynamics modelers should be executed in the order presented in Figure 6.7. This is due to the fact that some modules require the output of other modules as their input, and cannot be executed before this input is available.

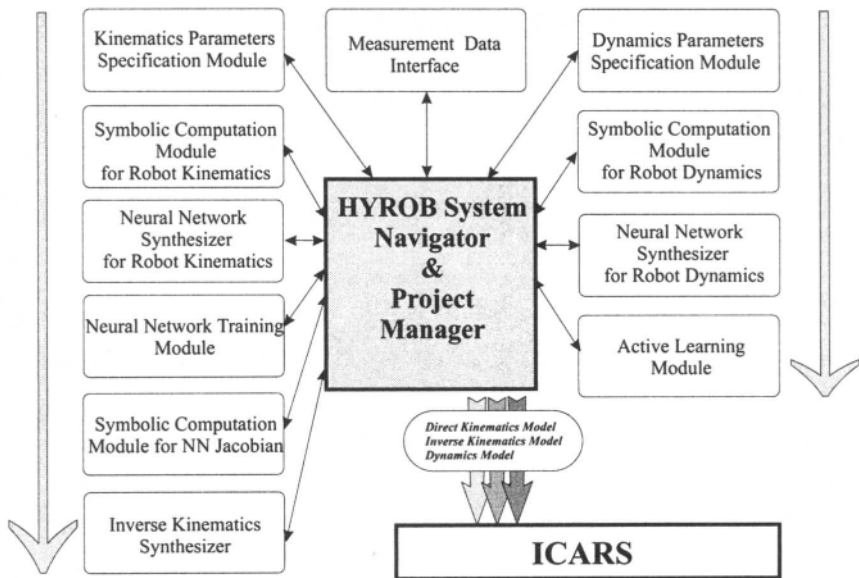


Figure 6.7. HyRob modules.

6.3.1. Robot Kinematic Data Specification Module

This is one of the top-level specification modules in the system. It queries the user for the Denavit–Hartenberg (and other) parameters of a robot to be used in the system and stores them in a special format. The first step is the specification of the Denavit–Hartenberg parameters d , s , and α as well as the range limits q_{\min} and q_{\max} for each joint (the value of α has to be given in radians). The Denavit–Hartenberg parameter file of the robot is created after all link information is entered by the user. This module is one of the centerpieces of the whole system. It has to be selected first when using the kinematics modeler for the first time because it is used to select or create the robot types that should be used in the ICARS project. The nominal data for each robot is not project-specific and therefore can be used in more than one project.

6.3.2. Symbolic Calculation-Based Kinematics Builder

With the information from the nominal kinematic parameters file, the module calls the computer algebra system Mathematica (possibly on a remote computer) and creates a file with the kinematic equations of the robot.

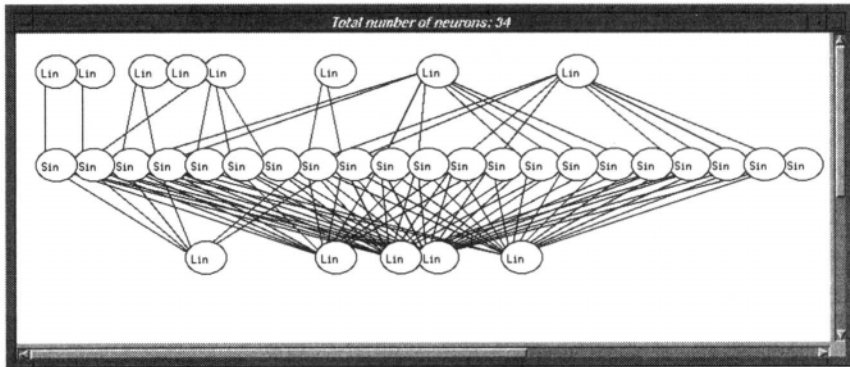


Figure 6.8. Example of a neural network synthesizer output.

6.3.3. *Neural Network Synthesizer for Robot Kinematics*

The module calls the computer algebra system Mathematica once more and transforms the kinematic equations of the robot into the form of a sinusoidal neural network. It produces the kinematic equations in a sum-of-sines format. The values of the all network weights are calculated. The resulting network is then displayed on the screen (see Figure 6.8).

The number of neurons in the network is displayed in the title bar of the window.

6.3.4. *Measurement Data Interface*

The measurement data interface is used to generate data on the pattern for the robot kinematics trainer. The user is then queried for the number of cycles. This value is needed in the training (calibration) module, an item in the kinematics modeler. The number of cycles determines how many times the kinematics neural network has to go through the list of measurement points in the calibration process. This module is used to calculate the differences between symbolic calculation based nominal kinematic parameters and the slightly different “real” values of these parameters obtained from a measurement system.

6.3.5. *Training Module for Direct Kinematics*

The calibration (training) process is started by selecting a “real” data file of robot kinematics. The symbolic calculation based nominal kinematics of the robot

is now trained (using a backpropagation method and delta rule) based on the information specified in the measurement data interface module.

Depending on the parameter values of learning items and learning cycles (specified in the measurement data interface module), the calibration process can take several minutes.

6.3.6. Calibrated Jacobian Neural Network Builder

After the calibration process, the calibrated neural network is transformed back to a symbolic expression format by the Mathematica function builder. The symbolic representation of the calibrated kinematics is then processed by Mathematica to calculate the Jacobian matrix.

This matrix is also transformed into a neural network format, resulting in a parallel neural implementation of the calibrated Jacobian matrix.

6.3.7. Inverse Kinematics Builder

The inverse kinematics builder is a stand alone module and can also be called from outside the HyRob system.

The neural-implemented robot direct kinematics and its Jacobian matrix can be loaded from the ICARS project database. After selecting a robot, a wireframe model of the robot is displayed on the screen. Next, the module calls the neural network synthesizer to create a coupled neural network for the inverse kinematics, which consists of neural networks for the direct kinematics and the Jacobian calculation together with a numerical part for calculation of the parameters by the gradient method.

After selecting the gradient method parameter values, the simulation can be run. The robot is animated and shown proceeding along the path specified by the sequence of route points in Cartesian space.

The modules of the HyRob system for synthesizing the neural implementation of the robot dynamics have similar properties.

6.3.8. Remark

The ICARS and HyRob systems have been developed in cooperation between the University of Linz (Austria) and the University of Wroclaw (Poland) and are available at the Research Institute for Symbolic Computation, University of Linz, Austria.

This page intentionally left blank

PART II

Event-Based Real-Time Control of Intelligent Robotic Systems Using Neural Networks and Fuzzy Logic

This page intentionally left blank

CHAPTER 7

The Execution Level of Robotic Agent Action

Process planning based on the description of task operations and their precedence relation creates a fundamental plan of cell action by decomposing the task into an ordered sequence of technological operations. In the next phase a real-time event-based controller of the computer-assisted robotic cell (CARC) is synthesized which generates via simulation a sequence of future events of a virtual cell in a given time window. These events are compared with the current states of the real cell and are used to predict motion commands for the robots and to monitor the process flow. The simulation model is modified any time the states of the real cell change and current real states are introduced into the model.

The simulation used to create a control system is event oriented and is based on complex DEVS (discrete event system) concept introduced by B. Zeigler (Figure 7.1). Such a model has the following structure:

$$Dev = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta) \quad (7.1)$$

where

- X is the set of external input events
- S is the set of the sequential states
- Y is the output value set
- δ_{int} is the internal transition specification function
- δ_{ext} is the external transition specification function
- λ is the output specification function
- ta is the time advance function function

with the following constraints:

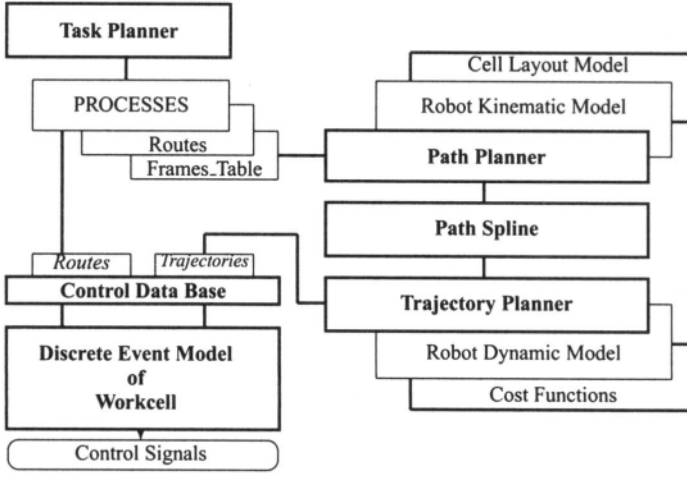


Figure 7.1. Structure of DEVS and its control database.

- a. The total state set of the system specified by *Dev* is

$$QE = \{(s, t) | s \in S, 0 \leq t \leq ta(s)\}$$

- b. δ_{int} is a mapping from S to S :

$$\delta_{int} : S \rightarrow S$$

- c. δ_{ext} is a function:

$$\delta_{ext} : QE \times X \rightarrow S$$

- d. ta is a mapping from S to the nonnegative reals with infinity:

$$ta : S \rightarrow \Re$$

- e. λ is a mapping from S to Y :

$$S \rightarrow Y$$

Further explanation of DEVS and its semantics is presented in (Zeigler, 1984; Jacak and Rozenblit, 1992b; Jacak and Rozenblit, in press).

Workcell components such as NC-machine tools, robot conveyors, etc., are modeled as elementary DEVS systems. Discrete changes of state of these systems are of interest. We separate the description of the model from simulation data. A

set of circumstances under which a model is experimented with is called a *control database* (Zeigler, 1984).

The variants of the fundamental plan of actions \wp obtained from the process planner should be tested by a simulator and then controlled in a real workcell. For the simulation, the system must have knowledge of how individual robot actions are carried out in the process.

The most important parameters are the time τ_d^o it takes to complete an operation o and the time the robot requires to service a workstation. The time τ_d^o depends on the type of machine on which the operation o is being executed. It is fixed, but can be changed by replacing the machine.

Similarly, the times of the PICKUP and PLACE operations are determined by the type of part and the machine on which the part is processed.

The times of the robot's interoperational moves (transfers) τ_r^{motion} depend on the geometry of the workscene and the cost function of the robot's motion. This function determines the dynamics of motion along the geometric tracks and the duration of the moves. These data must be accessible in order to simulate the entire system and are the database for control.

Based on the fundamental plan of actions \wp and its production route \mathbf{p} , the position table (**Frames_Table**) for all motions of each robot is created. The production route and **Frames_Table** determine the geometric parameters of the robot's movements. The path planner computes the shortest collision-free geometric track of motion. Then the optimal speed and acceleration of moves along the precomputed track are calculated. This task is solved by the trajectory planner. The trajectory planner receives the geometric tracks as input and determines the time history of position, velocity, acceleration, and input torques, which are then input to the trajectory tracker.

Hence we obtain the optimal trajectory and the times of the manipulator transfer moves. The time trajectories of the motions are the basis for computing the *times of each fundamental action*. In this way we create the knowledge base of the discrete event-based simulator. The structure of the control database is shown in Figure 7.1.

7.1. Event-Based Modeling and Control of Workstation

Each workstation $d \in D$ is modeled by two coupled atomic discrete event systems (DEVS) called the active and the passive models of device (Jacak and Rozenblit, 1994; Jacak and Rozenblit, 1993)

$$DEVS_d = (Dev_d^A, Dev_d^P) \quad (7.2)$$

where Dev_d^A is the active model and Dev_d^P is the passive model. The active atomic DEVS performs the simulation process and the passive atomic DEVS represents

the register of real states of the workstation as obtained from sensor signals, and acts as a synchronizer between the real and simulated processes.

This complex model is a modified version of the events system proposed by B. Zeigler and has the following structure.

7.1.1. The Active Model of a Workstation

We have

$$Dev_d^A = (X_V(d), S_V(d), \delta_{int}^d, \delta_{ext}^d, ta^d) \quad (7.3)$$

where

- $X_V(d)$ is the set of external input virtual event types
- $S_V(d)$ is the set of sequential virtual states
- δ_{int}^d is the set of internal transition specification functions
- δ_{ext}^d is the external transition specification function
- ta^d is the time advance function

with the following constraints:

- a. The total virtual event set of the system specified by Dev_d^A is

$$E_V(d) = \{(s, t) | s \in S_V(d), 0 \leq t \leq ta^d(s)\}$$

- b. δ_{int} is the set of parametrized internal state-transition functions:

$$\begin{aligned} \delta_{int} &= \{\delta_{int}^\mu | \mu \in U\} \\ \delta_{int}^\mu &: S_V \rightarrow S_V \end{aligned}$$

- c. δ_{ext} is the external state-transition function:

$$\delta_{ext} : E_V \times X_V \rightarrow S_V$$

- d. ta is a mapping from S_V to the nonnegative reals with infinity:

$$ta : S_V \rightarrow R$$

We describe now each component of an active workstation model. Each workstation $d \in D$ can have a buffer. The capacity of this buffer is denoted by $C(d)$. If a workstation has no buffer, then $C(d) = 1$.

Let $NC(d)$ (NC program register) denote the set of operations performed on the workstation d , i.e., $NC(d) = \{o \in O | (o, d) \in \alpha\}$.

7.1.1.1. *The virtual state set of a workstation.* The state set of d is defined by

$$S_V(d) = SD_d \times SB_d \quad (7.4)$$

where SD_d denotes the state set of the machine and SB_d denotes the state set of its buffer. The state set SD_d is defined as

$$SD_d = S_{\text{ready}} \cup S_{\text{busy}} \cup S_{\text{done}} \quad (7.5)$$

where

- $S_{\text{ready}} = \{\text{ready}\}$ signifies that **the machine is free**
- $S_{\text{busy}} = \{(\text{busy}, a) | a \in \text{NC}(d)\}$ and (busy, a) signifies that **the machine is busy processing the a -operation**
- $S_{\text{done}} = \{(\text{done}, a) | a \in \text{NC}(d)\}$ and (done, a) signifies that **the machine has completed the a -operation and is not free**

The state set of workstation's buffer SB_d is specified as

$$SB_d = (O \times \{0, 1, \#\})^{C(d)} \quad (7.6)$$

Let $C(d) = K$ and $b_d = (b_i | i = 1, \dots, K) \in SB_d$; then

$$b_i = \begin{cases} (0, 0) \leftrightarrow & \text{ith position of the buffer is free} \\ (0, \#) \leftrightarrow & \text{ith position of the buffer is reserved for a part being} \\ & \text{currently processed} \\ (a, 0) \leftrightarrow & \text{ith position of the buffer is occupied by a part before} \\ & \text{operation } a \\ (a, 1) \leftrightarrow & \text{ith position of the buffer is occupied by a part after} \\ & \text{operation } a \end{cases}$$

The state of the workstation's buffer is described by a vector whose coordinates specify the current state of each position in the buffer. We assume that the i th position denotes the location at which a part is placed in the buffer.

7.1.1.2. *Event-based workstation controller.* Given the virtual state set, we now define the internal state-transition functions δ_{int} . These functions represent the model of the workstation controller. The internal state transition function describes the control mechanism which determines the work of the device. Each of these functions is parametrized by the external parameter u which is loaded into the workstation from a higher level of control, namely from the workcell organizer. The parameter $u \in U$ is the operation's choice function, and represents the priority strategy of the workstation.

The strategy u can be interpreted as a function which determines the current realization priority for each operation from *Task*.

For the given operation o such a priority is represented by the pair $(o, \mu_u(o))$, where $\mu_u(o) \in [0, 1]$ is the membership function of the fuzzy representation of the priority (see Chapter 9). This function is calculated by the fuzzy decision system based organization level.

Then the operation's choice function is given by

$$u : 2^{NC(d)} \rightarrow NC(d) \quad (7.7)$$

where

$$u(A) = o \in A \text{ such that } \mu_u(o) = \max\{\mu_u(x) | x \in A\} \text{ and } u(\emptyset) = \emptyset$$

The strategy u defines the priority rule under which the operations will be chosen to be processed from the device's buffer.

Let $s(d) = (s_d, (b_1, b_2, \dots, b_K)) = (s_d, b) \in S_V(d)$, and let

$$\text{Wait}(d) = \{o | (\exists b_j)((o, 0) = b_j)\} \quad (7.8)$$

be the set of operations awaiting processing.

Then the internal transition function

$$\delta_{\text{int}}^u : S_V(d) \rightarrow S_V(d) \quad (7.9)$$

is specified as follows:

$$\delta_{\text{int}}^u(s(d)) = \begin{cases} ((\text{done}, a), (b_1, b_2, \dots, b_K)) & \leftrightarrow s_d = (\text{busy}, a) \\ ((\text{busy}, a), (b_1, \dots, b_{i-1}, (0, \#), b_{i+1}, \dots, b_K)) & \leftrightarrow s_d = \text{ready} \wedge \\ & u(\text{Wait}(d)) = a \wedge \\ & (a, 0) = b_i \\ (\text{ready}, (b_1, b_2, \dots, b_K)) & \leftrightarrow s_d = \text{ready} \wedge \\ & \text{Wait}(d) = \emptyset \\ (\text{ready}, (b_1, \dots, b_{i-1}, (a, 1), b_{i+1}, \dots, b_K)) & \leftrightarrow s_d = (\text{done}, a) \wedge \\ & b_i = (0, \#) \end{cases}$$

Fact 7.1.1. *It is clear that if $s_d = (\text{done}, a) \vee s_d = (\text{busy}, a)$, then $(\exists! i)(b_i = (0, \#))$ because the workstation cannot process two parts simultaneously.*

□

□

The internal function is used to perform the control process of the workstation.

7.1.1.3. Modeling of workstation–robotic agent interaction. The interaction between robot and workstation is modeled by the external state transition function. The communication between the workstation and robot is realized by exchanging external events. The robot generates events, called the set of the workstation's external events, which act on the workstation and change its state.

The set of external virtual events for the workstation model Dev_d^A is defined as follows:

$$X(d) = \{e_d^1(a, i), e_d^2(a, i), e^0 \mid a \in NC(d) \wedge i = 1, \dots, K\} \quad (7.10)$$

where

- $e_d^1(a, i) = \text{PLACE}(a, 0)$ ON *ith position* signifies that before the a operation a part is placed on the i th position of the (d -workstation's buffer
- $e_d^2(a, i) = \text{PICKUP}(a, 1)$ AT *ith position* signifies that after a operation a part is removed from the i th position of the d -workstation's buffer
- $e^0 = \text{DO NOTHING}$

The robot's influence on the workstation is realized by the exchange of events and is modeled by the external transition function. The external transition function δ_{ext}^d for each workstation d is defined in the following way. Let $s(d) = (s_d, (b_i \mid i = 1, \dots, K)) \in S_V(d)$; then

- $\delta_{\text{ext}}((s(d), t), e^0) = s(d)$
- $\delta_{\text{ext}}((s(d), t), e_d^1(a, i)) = (s_d, (b_1, \dots, b_{i-1}, (a, 0), b_{i+1}, \dots, b_K)) \leftrightarrow b_i = (0, 0)$
- $\delta_{\text{ext}}((s(d), t), e_d^2(a, i)) = (s_d, (b_1, \dots, b_{i-1}, (0, 0), b_{i+1}, \dots, b_K)) \leftrightarrow b_i = (a, 1)$
- $\delta_{\text{ext}}((\cdot, \cdot), \cdot) = \text{failure for all other states}$

7.1.1.4. Time advance model. The time advance function for Dev_d^A determines the time needed to process a part on the d th workstation. It is defined in the following manner. Let $s(d) = (s_d, (b_i \mid i = 1, \dots, K))$; then

$$\text{ta}^d(s(d)) = \begin{cases} \tau_{\text{process}}(a) & \leftrightarrow s_d = (\text{busy}, a) \\ \tau_{\text{load}} + \tau_{\text{setup}}(a) & \leftrightarrow s_d = \text{ready} \wedge u(\text{Wait}(d)) = \{a\} \\ \tau_{\text{unload}} & \leftrightarrow s_d = (\text{done}, a) \\ \infty & \text{otherwise} \end{cases}$$

$\tau_{\text{process}}(a)$ denotes the tooling/assembly time of operation a for the workstation d . The terms $\tau_{\text{load}} + \tau_{\text{setup}}(a)$ and τ_{unload} denote the loading and setup, and unloading times for d , respectively.

Let at time t the workstation be in the active state s , which began at time moment t_o and such that $\text{ta}(s) \neq \infty$. This means that during the interval of time $[t_o, t_o + \text{ta}(s)]$ the state s is active. After this time the workstation transfers its state from s into $\delta_{\text{int}}^\mu(s)$, which will be active in the next time interval $[t_o + \text{ta}(s), t_o + \text{ta}(s) + \text{ta}(\delta_{\text{int}}^\mu(s))]$.

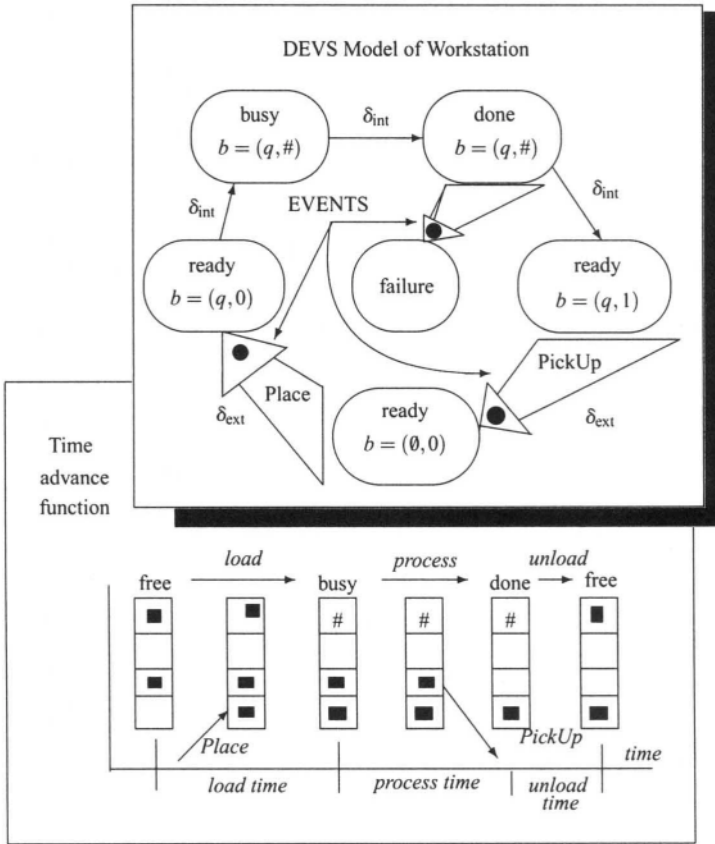


Figure 7.2. Active DEVS model of a workstation.

Fact 7.1.2. It is easy to observe that external events are not able to change the advance time for each active state.

Let $x = (e, t)$ be an external event different from e^0 which occurs at time $t \in [t_o, t_o + ta(s)]$. The workstation changes its state to state $s' = \delta_{ext}((s, t), e)$ and the advance time of the new state s' is equal to $ta(s') = t_o + ta(s) - t$.

□

□

The above is a complete active model of a cell's workstation.

Example 7.1.1. An example of a discrete event model of a workcell which has only one place in its buffer is shown in Figure 7.2.

7.1.2. The Passive Model of a Workstation

The passive model of a workstation is represented by a finite-state machine (FSM):

$$Dev_d^P = (X_R(d), S_R(d), \varphi_{\text{ext}}^d, \lambda^d) \quad (7.11)$$

where

- $X_R(d)$ is the set of the external input real events
- $S_R(d)$ is the set of the sequential real states
- φ_{ext}^d is the external real-state transition specification function
- λ^d is the updating function

with the following constraints:

- a. φ_{ext}^d is the real-state transition function (one-step transition function):

$$\varphi_{\text{ext}}^d : S_R(d) \times X_R(d) \rightarrow S_R(d)$$

- b. λ^d is the updating function (output function):

$$\lambda : UP \times S_R(d) \times S_V(d) \rightarrow S_V(d)$$

where $UP = \{0,1\}$ and 0 denotes that the updating process is stopped and 1 denotes that updating should be performed.

The interaction between external sensors and the workstation is modeled by the state transitionfunction φ_{ext} . The external sensor system generates real events which can be used to synchronize the simulated technological process with a real technological process.

7.1.2.1. Registration process. The set of real events for the workstation model Dev_d^P is defined as follows:

$$X_R(d) = \{re_d^1(a,i), re_d^2(a,i), re_d^3(a,i), re_d^4(a,i) | a \in NC(d) \wedge i = 1, \dots, K\} \quad (7.12)$$

where

- $re_d^1(a,i)$ signifies that before the a operation a part is placed on the i th position of the d -workstation's buffer
- $re_d^2(a,i)$ signifies that after the a operation a part is removed from the i th position of the d -workstation's buffer

- $re_d^3(a, i)$ signifies that before the a operation a part is loaded into the machine and processing is begun
- $re_d^4(a, i)$ signifies that the machine has completed the a operation, the machine is unloaded, and a part is placed on the i th position of the d -workstation's buffer

We assume that the state set $S_R(d)$ of the passive model is the same as the state set of the active model, i.e., $S_R(d) = S_V(d)$.

Then the state transition function φ_{ext}^d for the workstation d can be defined as follows: Let $st(d) = (s_d, (b_i | i = 1, \dots, K)) \in S_R(d)$ then

- $\varphi_{\text{ext}}(st(d), re_d^1(a, i)) = (s_d, (b_1, \dots, b_{i-1}, (a, 0), b_{i+1}, \dots, b_K))$
- $\varphi_{\text{ext}}(st(d), re_d^2(a, i)) = (s_d, (b_1, \dots, b_{i-1}, (0, 0), b_{i+1}, \dots, b_K))$
- $\varphi_{\text{ext}}(st(d), re_d^3(a, i)) = ((\text{busy}, a), (b_1, \dots, b_{i-1}, (0, \#), b_{i+1}, \dots, b_K))$
- $\varphi_{\text{ext}}(st(d), re_d^4(a, i)) = (\text{ready}, (b_1, \dots, b_{i-1}, (a, 1), b_{i+1}, \dots, b_K))$

Such a state transition function represents the real-state registration mechanism of the workstation.

7.1.2.2. Synchronization. The updating signal set is $UP = \{0, 1\}$, where 0 denotes that the updating process is stopped and 1 denotes that updating should be performed. The output function λ is a function which forces the real state in the active model of workstation when updating is needed, i.e.,

$$\lambda(s_{\text{real}}, s, 1) = s_{\text{real}} \in S_V(d) \quad \text{and} \quad \lambda(s_{\text{real}}, s, 0) = s \in S_V(d) \quad (7.13)$$

The above is a complete model of a cell's workstation.

7.2. Discrete Event-Based Model of Production Store

We can define a model of a production store in a similar manner. The state set of a store m is specified as a vector of states of each store position, i.e.,

$$S_V(m) = (O \cup \{0\})^{C(m)} \quad (7.14)$$

where if $s_m = (s_i | i = 1, \dots, C(m)) \in S_V(m)$, then

$$s_i = \begin{cases} 0 \leftrightarrow & i\text{th position of store is free} \\ a \leftrightarrow & i\text{th position of store is occupied by a part after operation } a \end{cases}$$

The set of external events for m is similar to the set of external events of a workstation.

The internal transition δ_{int}^m is an identity function which can be omitted. The external transition function δ_{ext}^m for each store m can be defined in the same manner as the function δ_{ext}^d . The time advance function for Dev_m is equal to ∞ .

7.3. Event-Based Model and Control of a Robotic Agent

Each robot $r \in R$ is modeled by two coupled systems, the active and passive models of the device,

$$REVS_r = (Rob_r^A, Reg_r^P) \quad (7.15)$$

where Rob_r^A is the active discrete event model and Reg_r^P is the passive model. The active atomic DEVS performs the simulation process and the passive model represents the register of real states of the robot and acts as a synchronizer between the real and simulated processes.

7.3.1. Event-Based Model of a Robotic Agent

To create an active model of a robot, we use a discrete event atomic system which generates external events for other devices from the set D . Such an event generator receives as external input events the outputs of the cell state recognizer (acceptor) and the cell controller (selector) (see next section) in order to determine its own state.

The model of a robot r is defined by the following DEVS:

$$Rob_r = (S_V(r), X(r), \delta_{int}^r, \delta_{ext}^r, ta^r, \{Z_{rd}\}) \quad (7.16)$$

The DEVS model of each robot contains the state set

$$S_V(r) = GOAL \times POSE \times HS \quad (7.17)$$

where:

- $GOAL = O \times ((D \cup M) \times N)^2$ is the set of possible goals of the robot's actions, i.e., $(a, (w, j), (v, i)) \in GOAL$ denotes the initial (v, i) and final (w, j) effector frames for the robot motion that transfers the part before operation a , and $(w, j), (v, i)$ have the same meaning for $POSE$. We assume that the empty set $\emptyset \in GOAL$.
- $POSE = (D \cup M) \times N$ is the set of effector frame numbers of the robot's effector end in the base Cartesian space (effector frame = position and orientation), and $(v, i) \in POSE$ denotes the effector frame which determines the grasp on the part at the i th position of the v -workstation's buffer.
- HS is the set of states of the effector, i.e., $HS = \{Empty, Holding\}$.

The set of external events for the model of the robot R_r is defined as follows:

$$X(r) = \{e_r(a, (v, i), (w, j)) | a \in O \wedge \{v, w\} \subset D \cup M\} \cup \{e^0\} \quad (7.18)$$

where:

$$e_r(a, (v, i), (w, j)) = \text{PICK\&PLACE part } (a, 0) \text{ FROM } (v, i) \text{ TO } (w, j)$$

signifies that a part before the a operation should be picked from the i position on the v -device's buffer and placed at the j position of the w -device's buffer; and

$$e^0 = \text{DO NOTHING}$$

Given the state and external event set, we now define the transition functions.

Let $s(r) = (s_1, s_2, s_3)$. Then the internal transition function

$$\delta_{\text{int}}^r : S_V(r) \rightarrow S_V(r) \quad (7.19)$$

is specified as follows:

$$\delta_{\text{int}}^r(s(r)) = \begin{cases} (s_1, s_2, s_3) \leftrightarrow \\ s_1 = \emptyset \\ ((a, (w, j), (v, i)), (v, i), \text{Empty}) \leftrightarrow \\ s_1 = (a, (w, j), (v, i)), s_2 \neq (v, i), s_3 = \text{Empty} \\ ((a, (w, j), (v, i)), (v, i), \text{Holding}) \leftrightarrow \\ s_1 = (a, (w, j), (v, i)), s_2 = (v, i), s_3 = \text{Empty} \\ ((a, (w, j), (v, i)), (w, j), \text{Holding}) \leftrightarrow \\ s_1 = (a, (w, j), (v, i)), s_2 = (v, i), s_3 = \text{Holding} \\ ((a, (w, j), (v, i)), (w, j), \text{Empty}) \leftrightarrow \\ s_1 = (a, (w, j), (v, i)), s_2 = (w, j), s_3 = \text{Holding} \\ (\emptyset, (w, j), \text{Empty}) \leftrightarrow \\ s_1 = (a, (w, j), (v, i)), s_2 = (w, j), s_3 = \text{Empty} \end{cases}$$

The external transition function δ_{ext}^r for each robot r is defined as follows. Let $s(r) = (s_1, s_2, s_3) \in S_V(r)$; then

$$\delta_{\text{ext}}^r((s(r), t), e^0) = s(r)$$

$$\delta_{\text{ext}}^r((s(r), t), e_r(a, (w, j), (v, i))) = ((a, (w, j), (v, i)), s_2, s_3) \leftrightarrow \begin{matrix} s_1 = \emptyset \wedge \\ s_3 = \text{Empty} \end{matrix}$$

$$\delta_{\text{ext}}^r((., .), .) = \text{failure for all other states}$$

The internal and external state transition functions are shown in Figure 7.3. The time advance functions determine the following times for the *Empty* and *Holding* states:

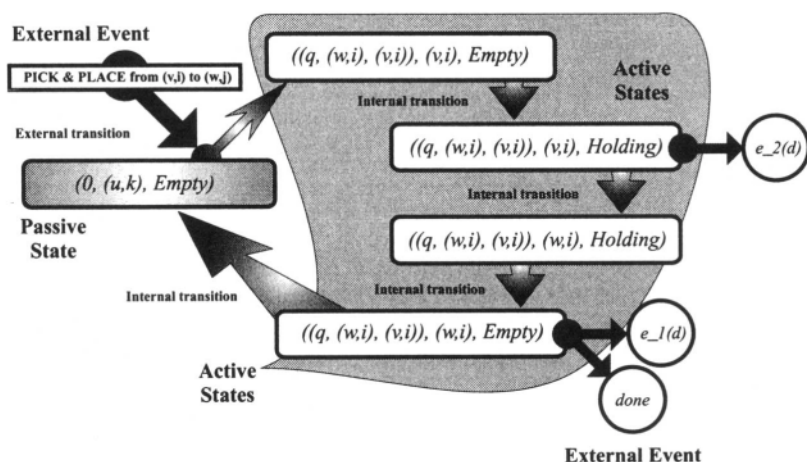


Figure 7.3. State transition and generated events for the model of the robot.

- The time of motion to position i : τ_{motion}^r
- The time of the pickup operation: τ_{pick}^r
- The motion time from position i to position j : τ_{motion}^r
- The time of the placement operation at the w th workstation: τ_{place}^r

The last component of Rob_r is the set of functions which generate external events for workstations or stores and the cell controller:

$$\{Z_{r,d}\} = \{Z_{r,d} : S_V(r) \rightarrow X_V(d) | d \in \text{Device}(r)\} \cup \{Z_{r,Acc}\} \quad (7.20)$$

where $\text{Device}(r) = \{\alpha(o_i) | ((o_i, o_{i+1}), r) \in \beta \wedge o_i \in \wp\}$ is the set of devices serviced by robot r in process \wp .

The function $Z_{r,d}$ generates external events for the model of the workstation d . More specifically, this function is defined as follows: For $s(r) = ((a, (w, j), (v, i)), (v, i), \text{Holding})$

$$Z_{r,d}(s(r)) = \begin{cases} e^0 & \text{if } d \neq v \\ e_d^2(a, i) & \text{if } d = v \end{cases} \quad (7.21)$$

and for $s(r) = ((a, (w, j), (v, i)), (w, j), \text{Empty})$

$$Z_{r,d}(s(r)) = \begin{cases} e^0 & \text{if } d \neq w \\ e_d^1(a, j) & \text{if } d = w \end{cases} \quad (7.22)$$

and

$$Z_{r,Acc}(s(r)) = \text{done}(r) \quad (7.23)$$

The model of the robot also generates external events (i.e., PICKUP and PLACE) for machines Dev_d , which trigger their corresponding simulators.

7.3.2. Passive State Register of the Robot Model

The passive model of a robot represents the measurement system of the robot's control. Each robot has own internal sensor system which can measure at every time point the real state of the robot,

$$Reg_r = (S_R(r), Int_Sens(r), \varphi_r, \lambda_r) \quad (7.24)$$

where:

- $Int_Sens(r)$ is the set of the internal sensor signals from the robot control system
- $S_R(d)$ is the set of the sequential real states
- φ_r is the input function,
- λ_r is the updating function

with the following constraints:

- a. φ_r is the input function:

$$\varphi_r : Int_Sens(r) \rightarrow S_R(r)$$

- b. λ_r is the updating function (output function):

$$\lambda_r : UP \times S_R(r) \times S_V(r) \rightarrow S_V(r)$$

where $UP = \{0,1\}$, where 0 denotes that the updating process is stopped and 1 denotes that updating should be performed.

The interaction between internal sensors and the robot model is described by the inputfunction φ_r . The internal sensor system generates signals which can be used to synchronize the simulated technological process with the real technological process.

We assume that the state set $S_R(r)$ of the passive model is the same as the state set of the active model, i.e., $S_R(r) = S_V(r)$. Then the input function φ_r for the robot r can be defined as follows:

$$\varphi_r(q_r) = s_R(r) \in S_R(r) \quad (7.25)$$

where $q_r \in Q_r$ is the joint configuration of robot r .

The output function λ forces the real state in the active model of the robot when updating is needed, i.e.,

$$\lambda(s_{\text{real}}, s, 1) = s_{\text{real}} \in S_V(r) \quad \text{and} \quad \lambda(s_{\text{real}}, s, 0) = s \in S_V(r) \quad (7.26)$$

The virtual workstations and robots (DEVS models) together with the real devices and robots represent the execution level of the CARC control system. The structure of the CARC execution level is illustrated in Figure 7.4.

7.4. Neural and Fuzzy Computation-Based Intelligent Robotic Agents

The execution of the transfer operations is realized by an event-based intelligent controller for each robot. The robot controller receives as input the output signals from the cell controller in order to determine its motion task. Additionally, the robot controller obtains the preplanned collision-free path realizing the transfer task from the coordination level.

The principal task of the intelligent local controller of a robotic agent is the execution of the given path of movement so that the robot action does not result in a collision with currently active dynamic objects (such as other robotic agents) in the cell. While the robot is tracking a preplanned path, some extraneous dynamic objects may enter the work space. These dynamic obstacles are detected by sensors mounted on the links of the manipulator. The sensor readings are analyzed to check for collision (see Figure 7.5). The problem of finding the path of a robot equipped with sensors has been dealt with by Lumelsky and Sun (1989). In this approach the manipulator moves along a specified path and traces the outer boundary of the obstacle when one of the links touches an obstacle. In our case the objective of sensory manipulation is preventive collision avoidance.

Recall that we prepared collision-free trajectories of motion for individual robotic agents during the off-line planning phase and the robot controller has local information about static obstacles in the robot's environment. For preventive collision avoidance we install ultrasonic sensors and a sensitive skin on each manipulator link and then use a neural network to estimate the proximity of objects to the link in question. The resulting distances are compared with the local virtual model of the robot's environment to recognize the new obstacles. The perception process realized by the neural network can be defined as the process of deriving specific items of information about meaningful structures in the environment from the sensor information.

The structure of an intelligent reactive controller is presented in Figure 7.6. The system consists of four main parts:

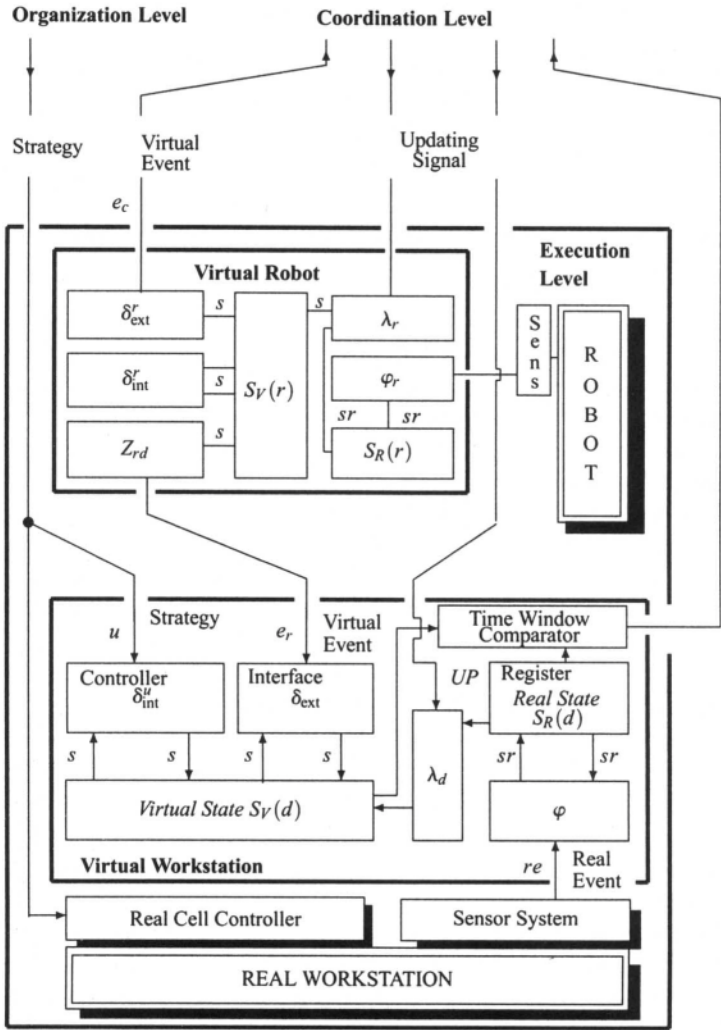


Figure 7.4. Structure of the execution level of a robotic workcell.

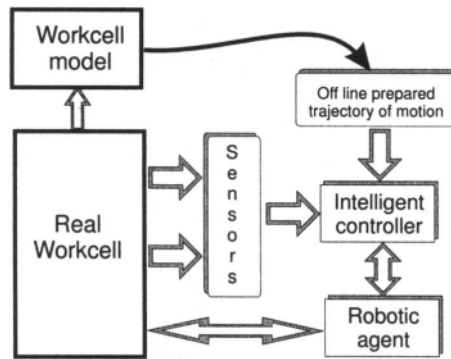


Figure 7.5. Intelligent robotic agent.

- The decision-making system
- The intelligent reactive one-step geometric path planner based on a multisensor local world model
- The intelligent one-step trajectory planner with neural-network based dynamics of the robot
- The intelligent reactive dynamic controller

The off-line trajectory planner is used every time a new goal state of the manipulator is set to the motion controller input. Then this planner computes the trajectory that allows one to achieve the manipulator goal state. It is clear that this trajectory does not ensure dynamical obstacle avoidance since we have no *a priori* information about these objects.

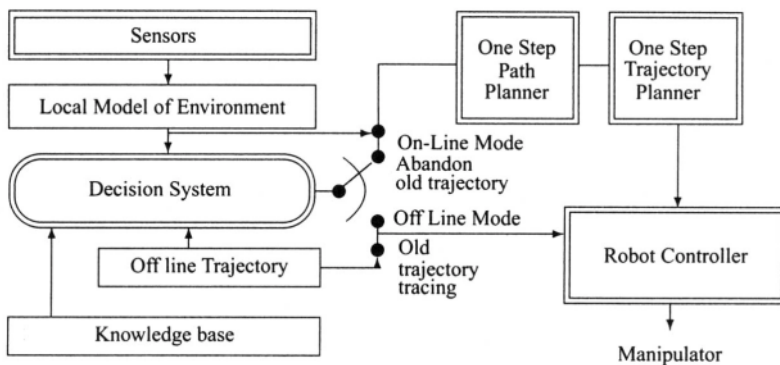


Figure 7.6. Intelligent controller of a robotic agent.

In the case when objects disturbing the off-line trajectory execution are reported via system sensors, the intelligent reactive one-step path planner is used to find new manipulator path segments. At the same time the intelligent one-step trajectory planner is used to calculate a time-trajectory segment corresponding to each path segment found by the path planner. To drive the robot manipulator along desired trajectories, the intelligent reactive dynamic controller is applied.

The decision-making system decides what the current mode of the motion controller work should be. Generally there are two possible main kinds of system work modes. The first is *Tracing*, in which the manipulator traces the trajectory preplanned by the off-line trajectory planner. In this kind of work mode the intelligent reactive controller is the only module used on-line to control the manipulator. In the second kind of work mode, the *Detour* mode, more system units are involved. When the system is switched to this mode some on-line trajectory planning is necessary.

7.4.1. Intelligent and Reactive Behavior of a Robotic Agent in the Presence of Uncertainties

The intelligent reactive controller allows one to drive a robot arm in the presence of environmental uncertainties. Based on signals coming from external sensors as well as its internal signals, the system produces the robot's input torques which make the robot perform a prescribed task.

7.4.1.1. Execution steps of the intelligent reactive controller of motion. When the intelligent reactive controller obtains the goal state from a higher layer, the off-line trajectory planner (see Chapter 5) uses the knowledge base to plan a trajectory connecting the current state of the robot with the goal state. When the trajectory is preplanned, the decision-making system calls the neural-based intelligent reactive controller to move the robot along this trajectory. The situation changes when the decision system reports (via the sensor system) an object in the scene which disturbs the trajectory execution. Then it calculates the desired direction of robot movement prohibiting the collision and passes it to the input of the intelligent reactive one-step path planner.

The path planner calculates a path segment lying along the desired direction of movement and passes it to the intelligent one-step trajectory planner, which finds the trajectory segment allowing the realization of the path segment just planned. Then the decision making system forces the controller to perform the trajectory planned by the one-step trajectory planner. The data flow corresponding to the above computations is shown in Figure 7.7.

We point out again that the most time consuming calculations in this case, i.e., the computation of the inverse kinematics, can be performed with the use of neural implementation.

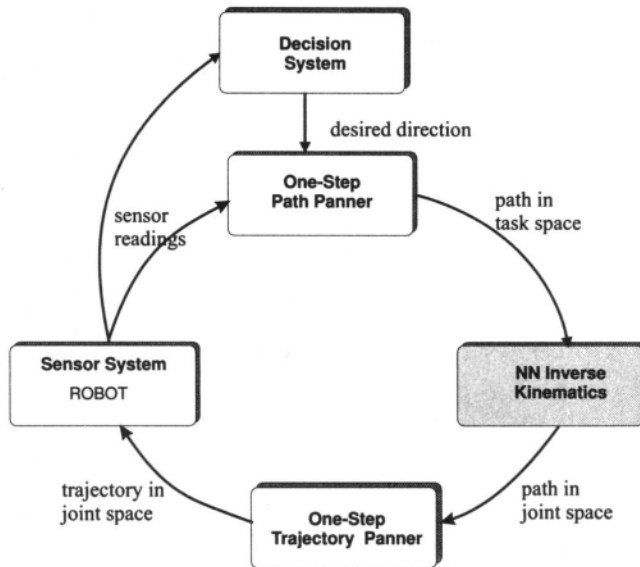


Figure 7.7. Data flow in the on-line trajectory planning stage.

The one-step path planner, in each step of its work, calculates only one point from the path segment along the given direction, which is used as input to the one-step trajectory planner. This planner calculates the trajectory segment leading to this point according to an established algorithm. During these steps the one-step planner and the one-step trajectory planner check whether the point is reachable without collision and without violating limiting constraints on the joint.

The on-line calculation steps described so far form the first cyclic loop in the system. This loop can be called the on-line trajectory planning loop since it finds the consecutive trajectory segments. The second loop in the system appears in the path execution stage. Figure 7.8 depicts the data flow in this stage. Here, when the one-step path planner sent the new desired position to the one-step trajectory planner, this generates the sequence of the new states in the joint space. Simultaneously the robot controller is called to drive the robot manipulator according to the desired state. This state continues until the desired position is achieved.

When the danger of collision disappears, the decision-making system forces the robot to return to the preplanned trajectory and then to execute the rest of it.

Below we describe the calculations performed by the motion controller components during one step of the system process.

7.4.1.2. Path segment planning step. When the motion controller is in one of the on-line modes, the intelligent reactive one-step path planner is used to find

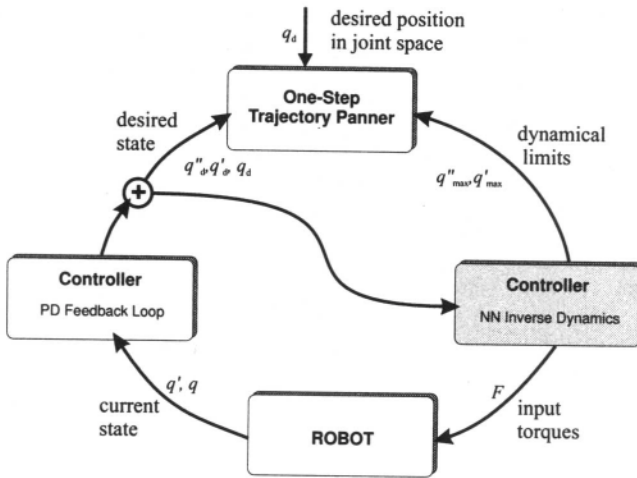


Figure 7.8. Data flow in the on-line trajectory execution stage.

segments of the path along which the manipulator should move. This task relies every time on finding one path point to which the robot manipulator should move. To this aim, the planner performs the following calculations in every iteration step of its work:

- It sets an arbitrary point in the workspace lying the direction of the desired movement.
- It uses the neural implementation of the inverse kinematics based on generalized sensor readings to calculate a collision-free position of the manipulator in the joint space.

Workspace point selection. The task to be performed now is to set an arbitrary point in the workspace lying in the direction of the desired movement. The manipulator end effector should travel to this point in the current step of activity. This point is generated by the control unit of the module. The control unit does this by translating the current manipulator end-effector position in the workspace by a unit vector in the desired movement direction multiplied by a scaling factor.

Joint-space position calculation. The point set by the control unit, the generalized sensor readings, and the current manipulator position in the joint space are used as inputs to the inverse kinematics module. This module finds the manipulator position in the joint space at which the manipulator end effector achieves the given point. The calculations are done iteratively. They start from the current manipulator position (which is treated as a temporary position) and follow by updating the temporary position in each step. The updating terms are calculated as a scaled gradient of the performance function to be minimized.

In each iteration of the algorithm for the inverse kinematics calculation the forward kinematics, the Jacobian matrix, and the terms responsible for the admissibility of the calculated positions (only collision-tree positions within the joint ranges) are calculated.

7.4.1.3. Trajectory-segment planning step. Now the task is to find functions of time that describe changes of the joint's acceleration, velocity, and position such that they start from the manipulator's current state and go to the desired position without violating the acceleration and velocity constraints. To do this, the planner performs the following calculations in every iteration step of its work:

- It uses the neural network dynamic model with zero payload to calculate limits on joint velocities and accelerations.
- It computes the period of time in which the current move must be performed.
- It computes the current trajectory segment.

Computation of limits on the joint motions. To calculate the maximal velocities and maximal and minimal accelerations which cannot be violated along the path segment considered in the present iteration, we assume that the move to the next point is to be performed along a straight line in the joint space. Since the present task is only to estimate the limit values, and since for typical situations consecutive points on the joint path lie close to one another, the limits obtained with this assumption are very similar to the real ones. With this assumption we parametrize the joint trajectory by an unknown, strictly monotonic function $s(t)$ where t is time, substitute this trajectory into the dynamic equations of the robot, set the components of input torques to their limiting values, and solve the equations, obtaining the limiting values of \ddot{s} and \dot{s} . This leads in a straightforward way to the limiting values of the joint velocities and accelerations in the neighborhood of the current path segment.

Computation of duration of movement. Having computed limits on joint velocities and accelerations, we are in a position to find the time period within which the manipulator is to perform the desired movement. First, based on the value of the desired end-effector velocity, we compute the lower bound of the movement realization time $t_{d \min}$. Then for each joint we check if it is possible within time $t_{d \min}$ to realize the movement from the initial state to the desired position without violating the velocity and acceleration limits. If we can, the time $t_{d \min}$ is the time we are looking for, if not, then for the joint for which the condition was not satisfied we compute a new value of the time t_d such that the move is now realizable and check whether it is valid for the rest of the joints. We repeat the procedure until it is possible to perform the movement of each joint within the calculated time.

Trajectory segment computation. The previous steps of the calculation yield the final time t_d within which the movement must be realized, and joint velocity and acceleration limits in the form

$$\begin{aligned} -\dot{q}_{\max} &\leq \dot{q} \leq \dot{q}_{\max} \\ \ddot{q}_{\min} &\leq \ddot{q} \leq \ddot{q}_{\max} \end{aligned} \quad (7.27)$$

Now we are in a position to calculate the current trajectory segment. The parameters are established by a fuzzy logic-based method. The calculations are performed separately for each manipulator joint. The calculated trajectory is in a form ready to be executed by the controller.

7.4.1.4. Trajectory execution step. Both kinds of trajectories, one obtained from the off-line trajectory planner and one from the intelligent one-step trajectory planner are executed by the intelligent reactive dynamic controller. The main computations performed by the dynamic controller are calculation of the sequence of manipulator input torques which will drive the manipulator along the desired trajectories. The most time-consuming calculations, i.e., the computation of the robot inverse dynamics, are performed in this case by neural networks. Additionally, taking advantage of the possibility of neural network learning, some adaptation of the controller is performed (Jacak et al., 1995e; Jacak et al., 1995g). This adaptation allows the control system to react to changes of the robot dynamic parameters (for example, different payloads in the gripper).

7.4.2. Multisensor Image Processing-Based World-Modeling and Decision-Making System

Instead of assuming only a sensitive skin on each manipulator link which responds only when the obstacle actually touches the manipulator, we install additional r rings of ultrasonic range finders on the links and use the neural network to estimate the proximity of obstacles to the link in question.

The sensing process is defined as that by which measurements of sensor readings are obtained. The perception process realized by the neural network is defined as the process of deriving from the sense data specific items of information about meaningful structures in the environment.

The inputs of the neural network are the signals \mathbf{o}_{rj} from the j th ultrasonic sensors of r th rings mounted on the manipulator links. The outputs of the neural network are vectors \mathbf{d}_i ($i = 1, \dots, n$) representing the minimal distances to objects in the robot's environment for each joint of the manipulator.

The sensing process can be interpreted as a mapping of the state of the ultrasonic sensors and the touch sensors into a set of multisensor images of much lower dimension. The neural net performing the sensing process is trained based on a

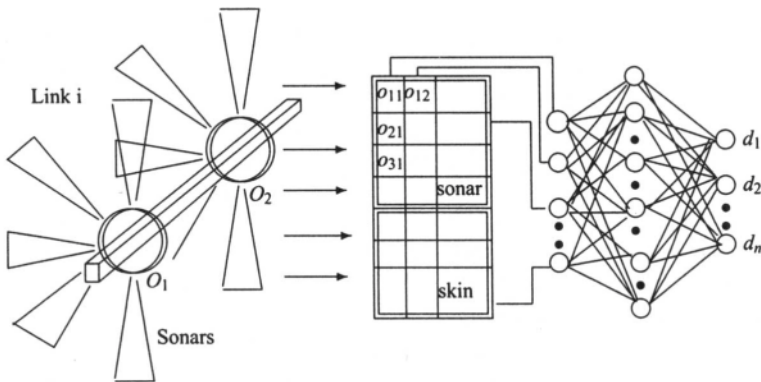


Figure 7.9. Sensor system.

geometrical model of the workcell during the off-line planning of movements for the individual robots. The sensor system is shown in Figure 7.9.

7.4.2.1. Physical principles of sonar ranging. Most conventional ranging systems employ a single acoustic transducer that acts as both a transmitter and receiver. After the transmitted pulse encounters an object, an echo may be detected by the same transducer acting as a receiver. The waveform of a typical echo observed as the output of the detection circuit has an oscillatory characteristic with decreasing amplitude (see Figure 7.10). A threshold level, denoted by τ , is included to suppress erroneous readings generated by electronic or acoustic noise.

A range measurement z_o is obtained from the round-trip time of flight by

$$z_o = ct_o/2$$

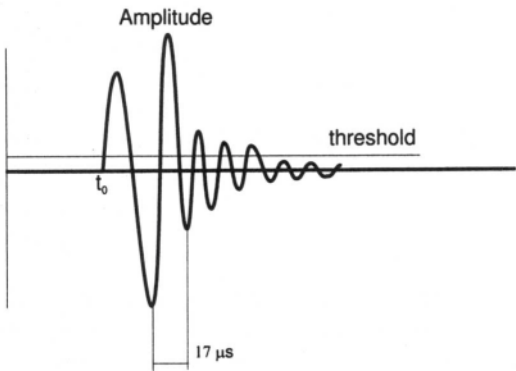


Figure 7.10. Typical echo waveform.

where c is the speed of sound in air and t_o is the time when the echo waveform first exceeds the threshold. The minimum range is determined by the time required for the transducer to switch from being a transmitter to a receiver, and allowing for the large transmitted voltage transients to decay below the threshold value. The maximum range is usually determined by the range in which the amplitude of the echo from a strong reflector falls below the threshold value. The maximal amplitude A of the waveform plays a crucial role in the detection process. In some cases it can be lower than the threshold τ , which can lead to detection errors. It was observed (Kuc and Viard, 1991) that the amplitude A can be described as a function of bearing (the angle with respect to the transducer orientation) and range. Other factors that influence the sensing process include, for example, the resonant frequency of the transducer, beam spreading and absorption, and reflection.

7.4.2.2. Sensor data combination. When a manipulator is equipped with many sensors and these sensors are mounted on different manipulator links the problem arises of how to combine the sensor readings to obtain useful information (Pau, 1988; Aggarwal and Wang, 1991). “Useful information” includes, for example, the distance from the nearest obstacle to the manipulator body, or “the safe” direction of manipulator movements. or the nature of a visible obstacle (dynamic or static).

Here, we combine the sensor readings in such a way that a manipulator can perform a given task. To this aim we apply an inverse kinematics algorithm that uses sensor combination methods as described below.

Projection method. The inverse kinematics algorithm (see Section 5.1) requires two inputs from the sensor data combination algorithm: a vector from a particular manipulator joint to the nearest obstacle (or equivalently, in the opposite direction, an escape vector), and the distance to it. To find such data we make the following assumptions:

- The manipulator model is a skeleton model
- Sensors are mounted directly onto manipulator links
- The positions and orientations of the sensing vectors are known
- The output o_i from the i th sensor is equal to o_i^{\max} if there is no obstacle within the sensing range $[o_i^{\min}, o_i^{\max}]$ and is equal to the distance to the visible obstacle otherwise
- Combination is performed only for sensors belonging to the same link
- Each link has its start position b (joint position) and the position of the link end is e

With these assumptions, let us consider the case of a manipulator link and the set of sensors mounted on it. Such a link (see Figure 7.11) can be interpreted as

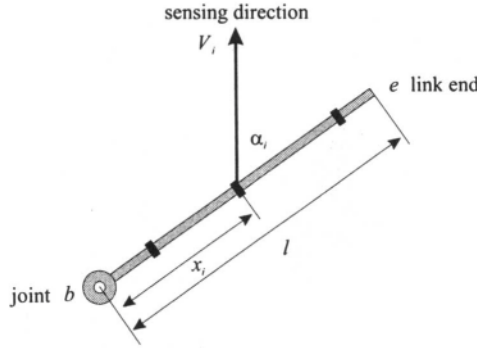


Figure 7.11. Manipulator link with sensors.

part of a line with sensors represented by vectors \mathbf{v}_i (we use a 2D representation for clearer explanation of the method).

The data combination algorithm combines two operations, namely vector projection and minimum finding. It can be expressed in the form

$$w = \min_i \left\{ v_i \cdot \cos \alpha_i \cdot o_i \cdot \frac{l}{x_i} \right\} \quad (7.28)$$

where \mathbf{o}_i is the i th sensor output, \mathbf{v}_i is the unit vector in the sensing direction (in the global coordinate frame), α_i is the angle between \mathbf{v}_i and the link, l is the link length, and \mathbf{x}_i the position of the i th sensor on the link. Figure 7.12 shows a simple example of sensor data combination is shown. The resulting vector w lies on the projection line k which is perpendicular to the link.

Linear approximation method. In the previous subsection we made some assumptions about the manipulator and sensor models. We make these assumptions here as well. Now we also assume that when the sensor j “sees” an obstacle, this obstacle is identified with a point \mathbf{p}_j . In the situation when no obstacle is visible, \mathbf{p}_j is identical with the mounting point of sensor j (which, in fact, lies on the link).

Knowing the link’s position, the vector \mathbf{v}_i , and the distance \mathbf{o}_i , we are able to calculate the position of this point in the global coordinate frame.

Let the point \mathbf{p}'_j be a “brother” of the point \mathbf{p}_j created in the following way (see Figure 7.13):

$$\mathbf{p}'_j = \mathbf{p}_j - 2 \cdot \mathbf{o}_j^{\max} \cdot \mathbf{v}'_j \quad (7.29)$$

if the obstacle is visible, and

$$\mathbf{p}'_j = \mathbf{p}_j$$

otherwise. \mathbf{v}'_j is the unit vector perpendicular to the link, created from the vector \mathbf{v}_j . In the next step, using well-known linear approximation methods (based on

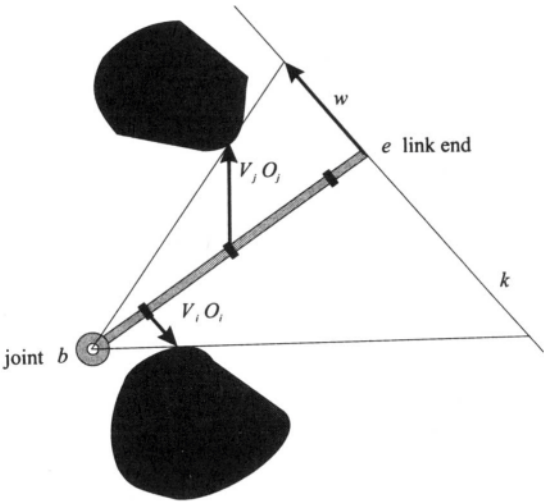


Figure 7.12. Example of sensor data combination based on the projection method.

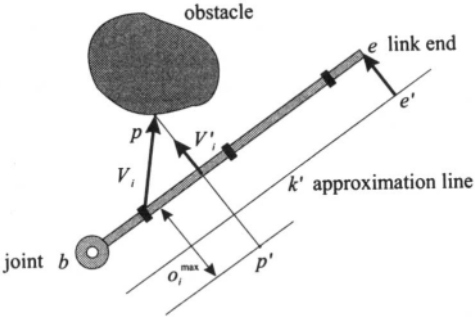


Figure 7.13. Example of sensor data combination based on the linear approximation method.

square error minimization), we can find a line k' approximating the set of points $\{p_j, p'_j, b, e\}$.

Now the vector representing the distance of the link to the nearest obstacle may be calculated from the normal projection e' of the link end e onto the line k' according to the following equation (see Figure 7.13)

$$w = e - e' \quad (7.30)$$

Application of these methods to each manipulator link produces a set of vectors D . Each particular vector from this set can be interpreted as the distance d_i to the obstacle and the direction to it for the appropriate manipulator joint. So it can be used in the decision system and one-step path-planning module.

Recall that in the first phase of workcell control development, i.e., during the off-line motion planning, a local model of the environment of the movement path is prepared. In general, the preplanned geometric and dynamic trajectory of motion of a robot r together with a local model of the world can be expressed as

$$TR(r)(\text{move from [A] to [B]}) = (q_r(s), s_r(t), D_r(s)) \quad (7.31)$$

where $q_r(s)$ is the parametrized track created by the path planner of the motion of robot r in the joint space, $s_r(t)$ is the time parameterization of motion along the track, as calculated by the off-line trajectory planner, and $D_r(s) = [D_1(s), D_2(s), \dots, D_n(s)]$ is the sensor-image compatible local model of the path environment. $D_i(s)$ represents the minimal distance vector from the i th joint to a static obstacle at point s of the trajectory.

7.4.2.3. Decision-making system. During the motion the local model is compared with the output of a neural network-based sensor-image processor and the following external events for a DEVS-based decision making system are created:

$$\{Path_free, New_obstacle, Danger_increasing, Danger_decreasing\}$$

The decision-making system is represented by an event-oriented DEVS whose transition function is presented in Figure 7.14.

Tracking denotes the state when the robot manipulator is tracking its currently assigned individual collision-free path. *Pause* is the state when the manipulator is stationary. *Detour* denotes the state when the manipulator is trying to circumvent an obstacle. *Advance* denotes the state when the obstacle has been circumvented and the manipulator is approaching its immediate goal. *Abandon* denotes the abandonment of the assigned path and *Deadlock* is the state when the environment and the manipulator seem to be static.

If the dynamic obstacle comes dangerously near to the link, the manipulator is stopped or is reconfigured to avoid touching the obstacle. Under extreme cases, it might happen that the obstacle cannot be avoided if the end effector is to pursue its preplanned path. The manipulator must now be moved to give way to the obstacle.

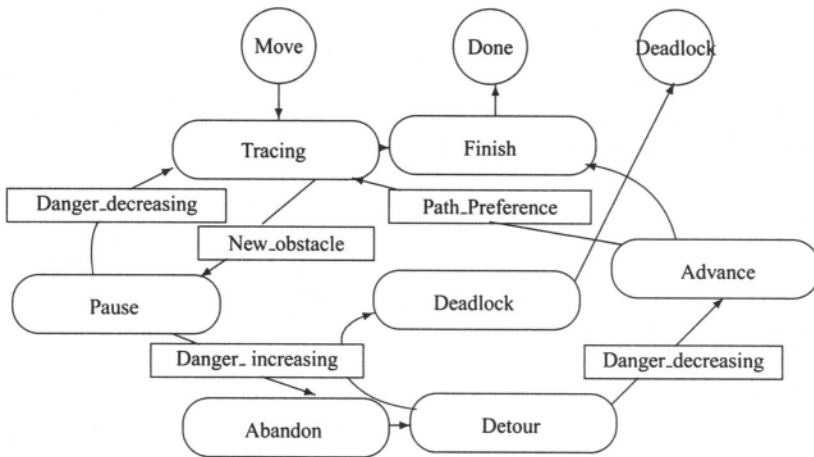


Figure 7.14. Event-based decision system for motion planning.

Then the manipulator is made to go through a sequence of actions to bring back the end effector to its original path. To realize such a task the local controller of the robot uses the above event-based decisions as the strategy to change between manipulator motion along the preplanned trajectory and neural calculation-based on-line planning of the motion to adopt at each step.

In extreme cases the robot must abandon the preplanned trajectory and find the next collision-free position. This position should be as near as possible to the preplanned path.

7.4.3. Neural Computation-Based On-Line Geometrical Path Planner in the Presence of Unknown Objects in the Environment

The one-step path planning and reactive control approach is based on the computation of robot kinematics and dynamics, and is computationally expensive. Therefore it is attractive to develop a neural network which automatically generates a safe configuration of the robot. Neural networks can be used to reduce the computational complexity of the model of robot kinematics. Additionally, neural networks can give robots the ability to learn and self-calibrate. Some neural network models used to attack this problem include the backpropagation network (Guo and Cherkassky, 1989; Martinez and Schulten, 1990), the Hopfield network (Kung and Hwang, 1989; Wu et al., 1993), the context-sensitive network (Lee and Bekey, 1991; Kawato et al., 1987; Yeung and Bekey, 1989), and the two-layer counterpropagation network (Hecht-Nielsen, 1987). Experiments show that

the forward kinematics models based on these neural network types have many disadvantages. For all of them the training time is too long for on-line relearning and these approaches can only provide approximations of the robot positions in the base frame. For this reason we use a multilayer feedforward neural network with hidden units having sinusoidal activation functions (Lee and Bekey, 1991; Lee and Kil, 1989; Lee and Kil, 1990) (see Chapter 5). To reduce the training time we apply hybrid techniques which automatically create the full network topology and values of the neural weights based on symbolic computation of components of the forward kinematics model.

Each component $t_i(q)$ of the direct kinematics can be represented in the form

$$t_i^s(q) = \sum_{j=1}^l l_j^s \sin(\mathbf{w}_j^s T q) \quad \text{for } i = 1, \dots, n \quad \text{and } s = x, y, z \quad (7.32)$$

where $t_i^s(q)$ defines the (i, m) output of the neural network representing the s th Cartesian variable of the i th joint position, q is the joint state vector, and $\mathbf{w}_j^s = (w_{j1}^s, \dots, w_{jn}^s)^T$ represents the weight vector of the j th sinusoidal function.

Such a form, obtained from symbolic network generation, represents the input–output function of a neural network with sinusoidal hidden units (see Chapter 5).

For the above reasons the model of manipulator kinematics must generate a sequence of robot configurations which realizes the motion along the desired effector displacement and avoids obstacles.

In Section 7.4.2 we described some methods for combining signals from sensors mounted onto robot links. Using these methods we can obtain the generalized distance and the vector from a particular robot joint to the nearest obstacle. We are going to use this information now and combine it with the inverse kinematics computation (methods for the inverse kinematics computation without obstacle avoidance are described in Chapter 5).

The solution of the inverse kinematics problem can be obtained by attaching a feedback network around a forward network to form a recurrent loop such that, given a desired Cartesian location from the path ℓ_s of a feedforward network, the feedback network iteratively generates joint angle correction terms to move the output of the forward network toward the given location. To avoid obstacles, we can add additional conditions combining the information from the local model of the robot's environment (the values of the neural network outputs d_i) with the current position of each robot joint.

Let

$$e_n(q(k)) = [\ell_s(k+1) - t_n(q(k))] \quad (7.33)$$

denote the location-error between the current position of the effector end (in step k) calculated by the forward kinematics and the desired next Cartesian location $\ell_s(k+1)$ (in step $k+1$) on the path.

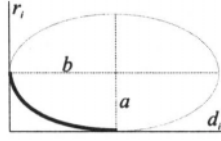


Figure 7.15. The ellipse function.

Obstacle avoidance can be achieved by introducing additional errors for each joint, i.e., errors between virtual points p_i and the joint positions $t_i(q(k))$:

$$e_i = [p_i - t_i(q(k))], \quad i = 1, 2, \dots, n-1$$

7.4.3.1. Virtual points. The virtual points p_i are the positions of the i th link that achieve collision avoidance and are given by

$$p_i = t_i(q(k)) - \frac{d_i}{\|d_i\|} \cdot r_i(d_i) \quad (7.34)$$

where d_i is the vector to the nearest obstacle (the generalized output from sensors mounted onto i th link), and $r_i(d_i) = a(1 - \sqrt{1 - (\|d_i\|/b - 1)^2})$ is an ellipse function parametrized by a, b (radii of the ellipse; see Figure 7.15).

Using formula (7.34), the virtual points are placed on the opposite side of the joint with respect to the obstacle (see Figure 7.16). Note that the parameter b can be interpreted as an emergence threshold. The terms $e_i, i = 1, \dots, n-1$, are defined to avoid collision. But the points p_i cannot be placed too far from the corresponding joints because this would yield a discrepancy from the goals. Therefore the parameters a and b have to be set carefully. The influence of a virtual point on the robot skeleton is shown in Figure 7.17.

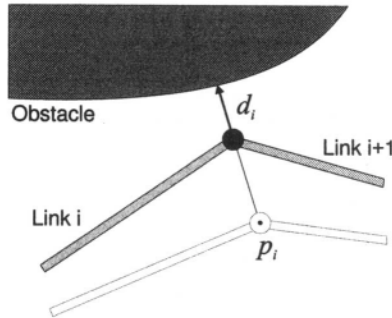


Figure 7.16. A virtual point.

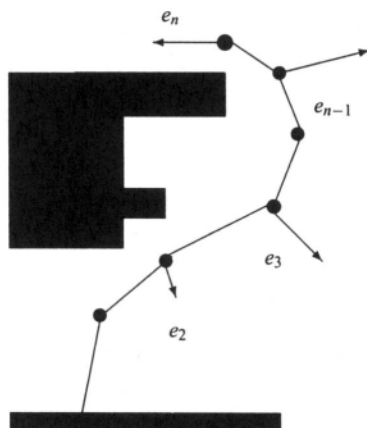


Figure 7.17. Virtual points.

7.4.3.2. *Inverse kinematics with on-line obstacle avoidance.* To solve the inverse kinematic problem in our particular case we transform it into an optimization problem as follows:

For a given end-effector position ℓ_s find a joint configuration q^* that minimizes the performance criterion

$$v(q) = \frac{1}{2} \sum_{i=1}^n \lambda_i e_i(q)^T e_i(q) \quad (7.35)$$

and for $i = 1, \dots, n$,

$$q_i^{\min} \leq q_i^* \leq q_i^{\max} \quad \text{and} \quad \sum_{i=1}^n \lambda_i = 1$$

where: $\lambda_i \geq 0, i = 1, \dots, n$ is a coefficient set, and $e_n = [\ell_s - t_n(q)]$ is the position error.

The above problem can be transformed into a problem without constraints by introducing the penalty function ω to obtain the limits on the joints. To this purpose we use additional errors for each degree of freedom, i.e.,

$$\epsilon_i^m = q_i^{\min} - q_i \quad \text{and} \quad \epsilon_i^M = q_i - q_i^{\max} \quad \text{for} \quad i = 0, \dots, n-1 \quad (7.36)$$

The penalty function is defined as (see Chapter 5)

$$\omega(q) = \frac{1}{2} \epsilon_m^T \Phi_m \epsilon_m + \frac{1}{2} \epsilon_M^T \Phi_M \epsilon_M \quad (7.37)$$

where $\epsilon_m = (\epsilon_i^m | i = 1, \dots, n)^T$, and Φ_m is a diagonal matrix $\Phi_m = \text{diag}[\varphi_i^m]$, and

$$\varphi_i^m = \frac{1}{1 + e^{-\beta \cdot \epsilon_i^m}} \quad (7.38)$$

Then the modified performance function is given by

$$v'(q) = v(q) + \omega = v(q) + \frac{1}{2} \epsilon_m^T \Phi_m \epsilon_m + \frac{1}{2} \epsilon_M^T \Phi_M \epsilon_M \quad (7.39)$$

By using the sigmoid function φ in the penalty function ω with large value of β we activate the error ϵ_i^m or ϵ_i^M only if $q_i < q_i^{\min}$ or $q_i > q_i^{\max}$, respectively.

Method based on a Lyapunov function. Let the performance criterion v' be such a function candidate. It is easy to see that v' is a locally positive definite function and if

$$\min_q v'(q) = v'(q^*) = 0$$

then q^* is the solution of the inverse kinematic problem.

The modified function v' can be interpreted as the Lyapunov function of the dynamics system. The time derivative of v' is given by

$$\dot{v}' = \text{grad } v'^T \cdot \dot{q} \quad (7.40)$$

where the gradient of the function v' has the following form:

$$\text{grad}_q v'(q) = - \sum_1^n \lambda_i J_i(q)^T e_i - \Phi_m \epsilon_m - D(\epsilon_m) \Phi'_m \epsilon_m + \Phi_M \epsilon_M + D(\epsilon_M) \Phi'_M \epsilon_M \quad (7.41)$$

where $J_i = [\partial t_i(q) / \partial q]$ is the Jacobian matrix of the kinematics t_i , $D(\epsilon_m) = \text{diag}[D_i]$ is the $n \times n$ diagonal matrix with $D_i = \epsilon_i^m$, and $\Phi'_m = \text{diag}[\varphi_i^m]$ is the diagonal matrix of the first derivative of φ_i^m such that $\varphi_i^m = \beta \varphi_i^m (1 - \varphi_i^m)$.

From (7.40) we can formulate an update rule to determine \dot{q} in such a way as to guarantee the convergence to a solution in the sense of Lyapunov. With the choice

$$\dot{q} = \eta \cdot \frac{\sum_1^n \lambda_i e_i(q)^T e_i(q) + \epsilon_m^T \Phi_m \epsilon_m + \epsilon_M^T \Phi_M \epsilon_M}{\text{grad } v'(q)^T \cdot \text{grad } v'(q)} \text{grad } v'(q) \quad (7.42)$$

v' becomes nonpositive throughout the convergence process. Updating q with \dot{q} iteratively the errors e_i converge to 0, i.e., the inverse kinematics is calculated.

Note that the solution of the inverse kinematics problem presented above uses only direct kinematics models, namely $t_i(q)$ and Jacobians $J_i(q)$, and both of these components can be implemented as neural networks. But the update rule (7.42)

cannot be implemented in a neural algorithm because it needs some additional terms to assure convergence stability and to increase the convergence rate.

Substituting (7.42) into (7.40), we obtain the following differential equation:

$$\dot{v}' = -v' \quad (7.43)$$

with the solution

$$v'(t) = v'(0)e^{-t} \quad (7.44)$$

The exponential convergence dynamics can be modified for faster convergence by modification of \dot{q} . So, instead of (7.42), \dot{q} can be calculated as follows:

$$\dot{q} = \frac{v'^{\alpha}}{\|\text{grad}_q v'\|^2} \cdot \text{grad}_q v'(q) = f(q, \alpha) \quad (7.45)$$

where α is a positive real constant. It can be seen that if $\alpha \geq 1$, q converges to q^* asymptotically. If $1/2 < \alpha < 1$, q converges to q^* within a finite time.

In practice, either (7.42) and (7.45) when implemented in discrete form needs careful control of the integration steps or update intervals. Let us consider the discrete version of (7.42),

$$q(k+1) = q(k) + \Delta q(k) \quad (7.46)$$

with k representing the k th iteration. Then $\Delta q(k)$ can be calculated from

$$\Delta q(k) = \dot{q}(k) \cdot \Delta t(k)$$

where $t(k)$ represents the k th update interval, and $\dot{q}(k)$ can be determined from (7.42) or (7.45). The stability associated with the updates rules given by (7.46) is dependent upon the proper selection of update intervals $\Delta t(k)$.

Steepest descent method. Minimization of the performance criterion v' can be done not only by using a Lyapunov function method, but also by applying a numerical technique, namely the steepest descent method. The idea of this method lies in the following algorithm.

Start at a point q_0 . As many times as needed, move from point q_i to the point q_{i+1} by minimizing along the line from q_i in the direction of the local downhill $-\text{grad}_q v(q_i)$.

This algorithm seems to be very simple, but it has some disadvantages. Therefore we propose to solve the inverse kinematics problem by applying the Polak–Ribiere conjugate gradient method. For further information we refer the reader to (William, 1990).

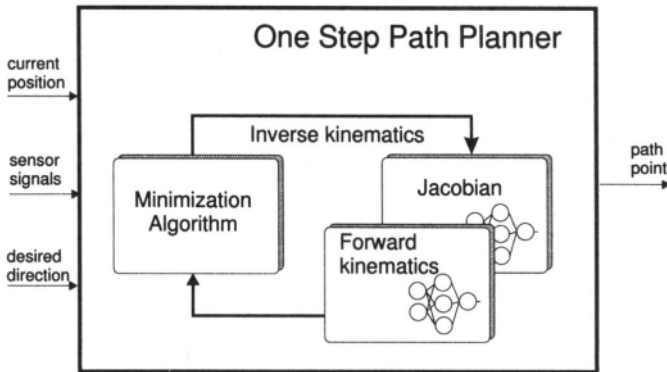


Figure 7.18. Structure of the one-step path planner.

7.4.3.3. One-step path planning based on multisensor data combination. Section 7.4 presented the system structure of the intelligent reactive motion controller. The one-step path planner (OSPP) is the part responsible for finding collision-free positions for the manipulator. The OSPP starts its work when the decision system switches the controller to the on-line mode. Such a situation occurs for the following states of the decision system: *Abandon*, *Detour*, and *Advance*. Based on such inputs as current position, sensor signals, and the desired direction of movement, the OSPP produces consecutive collision-free positions which are input to the one-step trajectory planner.

Structure of the one-step path planner. The structure of the one-step path planner is shown in Figure 7.18. In this structure we can distinguish two parts: the control unit module and the inverse kinematics module.

The *control unit module* is the master of the inverse kinematics module. It provides the initial parameters of the minimization algorithms according to the input signals, it starts the computations in the loop of the inverse kinematics module, and it stops these computations when a solution is found. This module is responsible for communication with the decision system so that when necessary, it runs the inverse kinematics computations (on-line mode of controller) and informs the decision system about the obtained solution. Lack of solutions (in the case of collision, for example) leads to the state *Deadlock* of the decision system.

The *inverse kinematics module* contains parts for forward kinematics and Jacobian computations implemented as neural networks. These parts are incorporated in a closed-loop system to realize the minimization algorithm presented above. The minimization algorithm finds the solution of the inverse kinematics problem including obstacle avoidance. Because the computations performed in the loop are initialized with the desired position of the manipulator end, this position is calculated earlier by the control unit with desired direction of movement of the

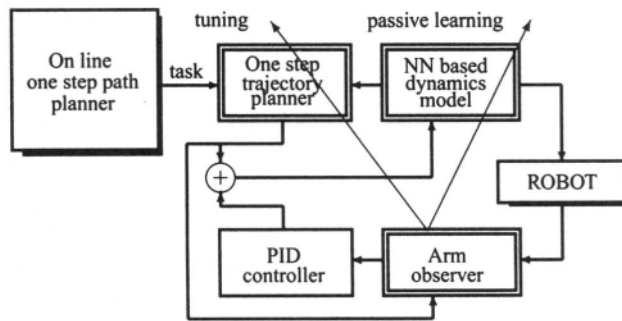


Figure 7.19. Structure of the controller of a robot's arm.

manipulator's end. The other inputs (current position and sensor signals) are taken into account in each step of the algorithm. The sensor signals are obtained after sensor data combination, where the current position input reflects the real manipulator configuration.

Connection with the one-step trajectory planner. The path point that is the output from the OSPP module is nothing but the next manipulator configuration in the joint space that allows the performance of the prescribed task with the avoidance of obstacles. But the path point is not enough to drive the manipulator, since it is a dynamic object. To this aim the time trajectory of the motion has to be known. The task of computing the trajectory of the motion is performed by the one-step trajectory planner.

7.4.4. Neural Network and Fuzzy Logic-Based On-Line Time-Trajectory Planner

The intelligent planning and control system drives a robot arm in the presence of environmental uncertainties. Based on the signals coming from the external sensors as well as its internal signals, the system inputs forces and torques to the robot by which the robot performs a prescribed task.

As we have mentioned, the system consists of two main parts: *the geometric motion planner* based on a multisensor local world-model builder and *the reactive dynamic motion planner and controller*.

The structure of the dynamic motion planning and control system is shown in Figure 7.19. When moving the manipulator from one point to another, the geometric motion planner (see Section 7.1 for complete description) uses the local world model to check whether the point is reachable without collision and without violating the limiting constraints on the joint. When this is possible, the point is passed to the reactive dynamic motion planner and controller.

The reactive dynamic motion planner and controller consists of the following blocks:

- A *neural network-based robot dynamics model*, which allows the computation of the inverse dynamics problem which is a necessary step in robot control (see Section 5.2 and (Jacak et al., 1995g))
- A *one-step trajectory planner*, which produces admissible trajectories based on a sequence of points from the robot's paths and exploits the model of robot dynamics
- A *motion controller*, which is a typical closed-loop computed torque controller equipped with a neural-computed dynamic model tuned during the control process

This section describes the details of the one-step trajectory planner. The first three subsections consider the signal part of this module; then a fuzzy tuning system is applied to obtain values of the one-step trajectory planner's parameters. The section finishes with examples of the implementation of the one-step trajectory planner.

7.4.4.1. One-step trajectory planner. A time trajectory of the motion is needed in order to control a robot arm along a given path. In the presented system this task is performed by the one step trajectory planner. Because movements of the arm are performed in an environment with uncertainties, only local information concerning the path is available, which makes planning of the whole time trajectory impossible.

Thus the part of the arm's trajectory leading to the next point on the arm's path is planned in every iteration of the planner. It is assumed that the arm's motion is to be performed with piecewise constant acceleration in the internal space. Since it is necessary to know the robot's maximal velocities and accelerations for trajectory planning and the only known constraints are put on the robot's input torques and forces, a model of the dynamics must be used to calculate the required limits. Here the neural network dynamic model with zero payload is used [see Section 5.2 or (Jacak et al., 1995g)].

Now the task is to find functions parametrized by time that describe changes of the joints' accelerations, velocities, and positions such that they start from the manipulator's current state and go to the desired position without violating the constraints on the accelerations and velocities.

In every iteration of the one-step trajectory planner the following data are used as the planner input:

- *maximal input torques and forces to the robot*
- *desired end-effector velocity*

- *current manipulator position in Cartesian coordinates*
- *current manipulator state in joint space*
- *desired position in Cartesian as well as in joint coordinates*

Using these data, the trajectory planner performs the following calculations in every iteration:

- Phase 1. Based on the robot's maximal input torques and force values and the current and desired manipulator states in joint space, it uses the neural network dynamic model with zero payload to calculate the limits on joint velocities and acceleration.
- Phase 2. Based on the desired end-effector velocity, the current and desired manipulator positions in Cartesian coordinates, and the limits on joint velocities and accelerations, it computes the time period in which the current movement must be performed.
- Phase 3. Using the current and desired manipulator states in joint space, the time in which the current movement must be performed and the limits on joint velocities and accelerations, it computes the current trajectory segment.

Each of these steps will be described below.

Computation of limits on the joints. To calculate the maximal velocities and maximal and minimal accelerations which cannot be violated along the path segment considered in the present iteration, we assume that the movement is to be performed along a straight line in joint space. Since the present task is only to estimate the limiting values, and since for typical situations consecutive points on the joint path lie close to one another, the limits obtained with this assumption are very similar to the real ones. This assumption allows us to write

$$\mathbf{q}_i(s(t)) = \mathbf{q}_{i0} + (\mathbf{q}_{id} - \mathbf{q}_{i0})s(t), \quad i = 1, \dots, n \quad (7.47)$$

where \mathbf{q}_{i0} is the current manipulator configuration in joint space (at t_0), \mathbf{q}_{id} is the desired manipulator configuration in joint space (at t_d), and $s(t)$ is an unknown, strictly monotonic function of time t such that

$$(\forall t)((s(t) \in [0, 1] \wedge s(t_0) = 0) \rightarrow (s(t_d) = 1))$$

With (7.47) the joint velocities and accelerations along the path are given as

$$\begin{aligned} \dot{\mathbf{q}}_i(s) &= (\mathbf{q}_{id} - \mathbf{q}_{i0})\dot{s} \\ \ddot{\mathbf{q}}_i(s) &= (\mathbf{q}_{id} - \mathbf{q}_{i0})\ddot{s}, \quad i = 1, \dots, n. \end{aligned} \quad (7.48)$$

Substituting (7.47) and (7.48) into the robot dynamic model given as

$$M(q)\ddot{q} + \dot{q}^T N(q)\dot{q} + G(q) = F \quad (7.49)$$

where M is the robot's inertia matrix, N is the robot's array of centrifugal and Coriolis terms, G is the robot's vector of gravity terms, and F is the vector of the robot's input torques and forces, we obtain

$$\tilde{M}(s)\ddot{s} + \tilde{N}(s)\dot{s}^2 + \tilde{G}(s) = F \quad (7.50)$$

By choosing $s = s(t_0)$ and setting the components of F to their limiting values, we can easily compute the limiting values of \ddot{s} and \dot{s} . This, together with the dependences (7.48), leads to the limiting values of joint velocities and accelerations in the neighborhood of q_0 and q_d . As mentioned before, a neural network dynamic model with zero payload is used as the model of the robot dynamics.

Computation of duration of movement. Having computed the limits on joint velocities and accelerations, we are in a position to find the time period within which the manipulator will be able to perform the desired movement. First we compute the lower bound of the movement realization time $t_{d \min}$ with the use of the desired end-effector velocity v_d as

$$t_{d \min} = \frac{\|x_0 - x_d\|}{v_d} \quad (7.51)$$

where x_0 and x_d are the initial and desired Cartesian positions, respectively, and $\|\cdot\|$ denotes the Euclidean norm.

Then for each joint we check if it is possible within time $t_{d \min}$ to realize the movement from q_{i0} to q_{id} without violating the velocity and acceleration limits.

- If yes, then the time $t_{d \min}$ is the time we are looking for.
- If not, then for the joint for which the condition was not satisfied we compute a new value of the time t_d such that the move is now realizable and check if it is valid for the rest of the joints. We repeat the procedure until it is possible to perform the movement of each joint within the calculated time.

Trajectory segment planning. Because for every manipulator joint the motion planning is performed in the same way, from now on we will continue the description for one joint only.

From the previous steps of the calculation we obtained the final time t_d within which the movement must be realized, and joint velocity and acceleration limits in the form

$$\begin{aligned} -\dot{q}_{\min} &\leq \dot{q} \leq \dot{q}_{\max} \\ \ddot{q}_{\min} &\leq \ddot{q} \leq \ddot{q}_{\max} \end{aligned} \quad (7.52)$$

With the assumption that we move with piecewise constant acceleration in the internal space and the conclusion that each move can be realized with no more than two switchovers, the expression for joint acceleration is

$$\ddot{q}(t) = \ddot{q}_0 + 1(t - t_0)(\ddot{q}_1 - \ddot{q}_0) + 1(t - t_1)(\ddot{q}_2 - \ddot{q}_1) \quad (7.53)$$

where $\ddot{q}_0, \ddot{q}_1, \ddot{q}_2$ is the sequence of joint accelerations, and t_1 is the time of the switch from \ddot{q}_1 to \ddot{q}_2 , $t_0 \leq t_1 \leq t_d$.

For (7.53), the velocity can be computed as

$$\begin{aligned} \dot{q}(t) &= \dot{q}_0 + \int_{t_0}^t \ddot{q}(\tau) d\tau \\ &= \dot{q}_0 + (t - t_0)1(t - t_0)\ddot{q}_1 + (t - t_1)1(t - t_1)(\ddot{q}_2 - \ddot{q}_1), \end{aligned} \quad (7.54)$$

where \dot{q}_0 is the initial joint velocity.

Continuing with the integration we obtain the expression for the path increment

$$\begin{aligned} \Delta q(t) &= \int_{t_0}^t \dot{q}(\tau) d\tau \\ &= \dot{q}_0(t - t_0) + \frac{1}{2}(t - t_0)^2 1(t - t_0)\ddot{q}_1 + \frac{1}{2}(t - t_1)^2 1(t - t_1)(\ddot{q}_2 - \ddot{q}_1). \end{aligned} \quad (7.55)$$

Since there are typically infinitely many possible trajectories satisfying the admissibility conditions, an optimization criterion is needed to select only one trajectory. Two criteria are proposed:

- the minimization of jerks
- the minimization of overshoots

THE MINIMIZATION OF JERKS: To describe the first criterion, we have to find an expressions for the jerks. After computing the time derivative of acceleration (7.53) we obtain

$$\ddot{\ddot{q}}(t) = \frac{\partial \ddot{q}}{\partial t} = \delta(t - t_0)(\ddot{q}_1 - \ddot{q}_0) + \delta(t - t_1)(\ddot{q}_2 - \ddot{q}_1) \quad (7.56)$$

The absolute value of $\ddot{\ddot{q}}(t)$ is simply expressed by

$$|\ddot{\ddot{q}}(t)| = \delta(t - t_0)|\ddot{q}_1 - \ddot{q}_0| + \delta(t - t_1)|\ddot{q}_2 - \ddot{q}_1| \quad (7.57)$$

From (7.57) it follows immediately that the jerk minimization criterion can be written as

$$Q_J = \int_{t_0}^{t_d} |\ddot{\ddot{q}}(t)| dt = |\ddot{q}_1 - \ddot{q}_0| + |\ddot{q}_2 - \ddot{q}_1| \quad (7.58)$$

THE MINIMIZATION OF OVERSHOOTS: To state the overshoot minimization criterion we need to introduce a formula describing a linear path, since an overshoot is defined as the difference between the real and linear paths in joint space. The linear path can be expressed as

$$\tilde{q}(t) = q_0 + \frac{q_d - q_0}{t_d - t_0}(t - t_0) \quad (7.59)$$

Now we can formulate the overshoots minimization criterion in two different ways:

$$Q_o = \|q(t) - \tilde{q}(t)\| = \begin{cases} \max_{t_0 \leq t \leq t_d} |q(t) - \tilde{q}(t)| \\ \int_{t_0}^{t_d} (q(t) - \tilde{q}(t))^2 dt. \end{cases} \quad (7.60)$$

Without going into details, we can state that both criteria have the form

$$Q = \Phi(\ddot{q}_1, \ddot{q}_2, t_1)$$

Now we formulate the optimization problem as follows:

Find the accelerations \ddot{q}_1, \ddot{q}_2 and the switch moment t_1 such that

$$\begin{aligned} Q_j &\rightarrow \min & Q_o &\rightarrow \min \\ -\dot{q}_{\max} &\leq \dot{q}(t_1) \leq \dot{q}_{\max} \\ -\dot{q}_{\max} &\leq \dot{q}(t_d) \leq \dot{q}_{\max} \\ \Delta q_d - \epsilon &\leq \Delta q(t_d) \leq \Delta q_d + \epsilon \end{aligned} \quad (7.61)$$

with

$$\begin{aligned} \dot{q}(t_1) &= \dot{q}_0 + (t_1 - t_0)\ddot{q}_1 \\ \dot{q}(t_d) &= \dot{q}_0 + (t_d - t_0)\ddot{q}_1 + (t_d - t_1)(\ddot{q}_2 - \ddot{q}_1) \\ \Delta q(t_d) &= \dot{q}_0(t_d - t_0) + \frac{1}{2}(t_d - t_0)^2\ddot{q}_1 + \frac{1}{2}(t_d - t_1)^2(\ddot{q}_2 - \ddot{q}_1), \end{aligned}$$

The joint interval $\Delta q_d = q_d - q_0$ denotes the desired path increment (obtained from the one-step path planner) and ϵ is the maximal allowed difference from q_d .

One can see that this problem is a typical polyoptimization problem. It can be solved by transforming it into a monooptimization problem in the following way. First we normalize both criteria according to the formula

$$\tilde{Q} = \frac{Q - Q_{\min}}{Q_{\max} - Q_{\min}} \quad (7.62)$$

Next we combine the criteria to get a new criterion in the form

$$Q' = \lambda \tilde{Q}_j + (1 - \lambda) \tilde{Q}_o \quad (7.63)$$

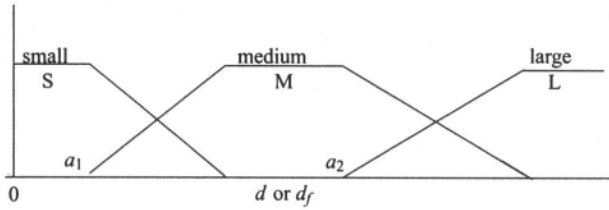


Figure 7.20. The primary fuzzy sets.

Because this method takes some parameters (i.e., λ and ϵ) as inputs that have to be set beforehand according to some criterion, we use a fuzzy logic-based self-tuning method to establish their values. The principles of this method will be sketched in the next subsection.

7.4.4.2. Fuzzy tuning. We apply a fuzzy rules-based decision system (see (McDermott, 1982; Efstathiou, 1987; Zadeh, 1973) for more details) to calculate the values of the one-step trajectory planner parameters. As input to the system we use two signals coming from the on-line path planner, namely the current minimal distance d between the robot body and obstacles in the scene, and the distance d_f to the last point on the path.

Fuzzyfication. These input values do not form a fuzzy set, but are crisp values. Therefore they first have to be fuzzyfied. In general, linguistic variables will be associated with a term set, with each term in the term set defined on the same universe of discourse. A fuzzyfication or fuzzy partition determines the number of primary fuzzy sets.

The possible primary fuzzy sets to describe these distances are, for example, *small* (S), *medium* (M), *large* (L). For each term S, M, and L a trapezoid-shaped membership function is used to cover the whole domain (universe) (see Figure 7.20). In principle, any shape is possible, for example, a bell-shaped function, but the simple form has many advantages: it is easy to represent and it lowers the computational complexity of the system. Because the universe is nonnormalized, the term could be asymmetrical and unevenly distributed in the domain. The union of the primary fuzzy set satisfies the epsilon-completeness property. The membership function of a primary fuzzy set depends on a vector of parameters, which determine its shape.

The same method can be used to define the terms and the primary fuzzy sets for the outputs of the fuzzy tuning system.

The same linguistic values can be assigned to both variables λ and ϵ : *small* (S), *medium* (M), and *large* (L).

Now we define a decision system as shown in Figure 7.21.

The rule system. In general, a fuzzy decision rule is a fuzzy relation which is expressed as a fuzzy implication. The choice of fuzzy implication reflects not

only the intuitive criteria for implication, but also the effect of the connective *also*. In general, fuzzy implication functions can be classified into three main categories (Zadeh, 1973; Baldwin and Guild, 1980):

- the fuzzy conjunction
- the fuzzy disjunction
- the fuzzy implication

Just like in classical fuzzy control, we use only *if-then* rules. The rule base consists of all *if-then* rules. An *if-then* rule is represented by a fuzzy relation (Baldwin and Guild, 1980). The rule base can be derived from expert knowledge, or can be extended by learning (Wang and Mendel, 1992). It is composed of a set of fuzzy rules with multiple distance variables (statistics inputs) and a single decision variable (output distances), represented as

if (x is A and y is B), then z is C.

For the sake of simplicity, we use a simple rule base in this case. We consider fuzzy decision rules with two inputs (d, d_f) and two separate outputs (λ, ϵ). A list of fuzzy rules applied in the tuning system is given below.

For the output λ :

- R_1: if d is S and d_f is S, then λ is S
- R_2: if d is S and d_f is M, then λ is S
- R_3: if d is S and d_f is L, then λ is S
- R_4: if d is M and d_f is S, then λ is S
- R_5: if d is M and d_f is M, then λ is M
- R_6: if d is M and d_f is L, then λ is M
- R_7: if d is S and d_f is S, then λ is M
- R_8: if d is S and d_f is M, then λ is L
- R_9: if d is S and d_f is L, then λ is L

For the output ϵ :

- R_10: if d is S and d_f is S, then ϵ is S
- R_11: if d is S and d_f is M, then ϵ is S
- R_12: if d is S and d_f is L, then ϵ is M
- R_13: if d is M and d_f is S, then ϵ is S
- R_14: if d is M and d_f is M, then ϵ is S
- R_15: if d is M and d_f is L, then ϵ is M

R_16: if d is L and d_f is S, then ϵ is S

R_17: if d is L and d_f is M, then ϵ is M

R_18: if d is L and d_f is L, then ϵ is L

The premises are compared with the input values of the fuzzy tuning system so that it can be decided which rules can be used and in which way, and which rules cannot be used. Thus we can determine which rules can be fired, together with the strength of each firing operation. This strength depends on how much the input values and the premises of the rule correspond to each other.

The decision-making system. The rules can be fired according to the fuzzyfication interface, each with a particular strength. This strength determines the amount of influence the conclusion of a particular rule has on the general conclusion of the system. The inference mechanisms employed in the fuzzy tuning system are similar to those used in a typical expert system. In the decision system, we reduce the inference mechanisms to only one-level, forward data-driven inference. In other words, we do not employ chaining inference mechanisms.

In the system, the firing strength s_i of each rule R_i is expressed by the intersection operator. This strength determines the amount of influence the conclusion of a particular rule has on the general conclusion of the system. Usually this is done in the following way. Say the strength is a , then the output fuzzy set λ is “cut off” at a or all the membership degrees are multiplied by a . Hence, mathematically, the output fuzzy set k^* is determined by

$$\mu_{k^*} = \min(a, \mu_k) \quad \text{or} \quad \mu_{k^*} = a \cdot \mu_k$$

or by a more complex operation. The result of this is a set of “cut off” fuzzy sets. They are combined using the union function and handed to the defuzzification interface.

Defuzzification. The fuzzy set that comes in the form of output has a very complicated form because it is a combination of several “cut-off” fuzzy sets as described above. The goal of the defuzzification interface is to find one single crisp value that summarizes this output fuzzy set. There are several mathematical operation that perform this, each with its own advantages and disadvantages. The most frequently used operation is the center-of-gravity method; another is the middle-of-maxima method. This method simply takes the first value that has the highest membership degree. The complete one-step trajectory planning system is shown in Figure 7.21.

Example 7.4.1. The performance of the one-step trajectory planner has been tested for a three-DOF Puma-type robot manipulator (see Figure 5.20). Two types of simulation for the one-step trajectory planner were performed, a simulation applying either the jerk minimization or the overshoot minimization criterion separately. To make this example more illustrative, it is assumed that in each step, \ddot{q}_2 [see (7.53) in Section 7.4.4.1] is equal to zero. The manipulator had to follow

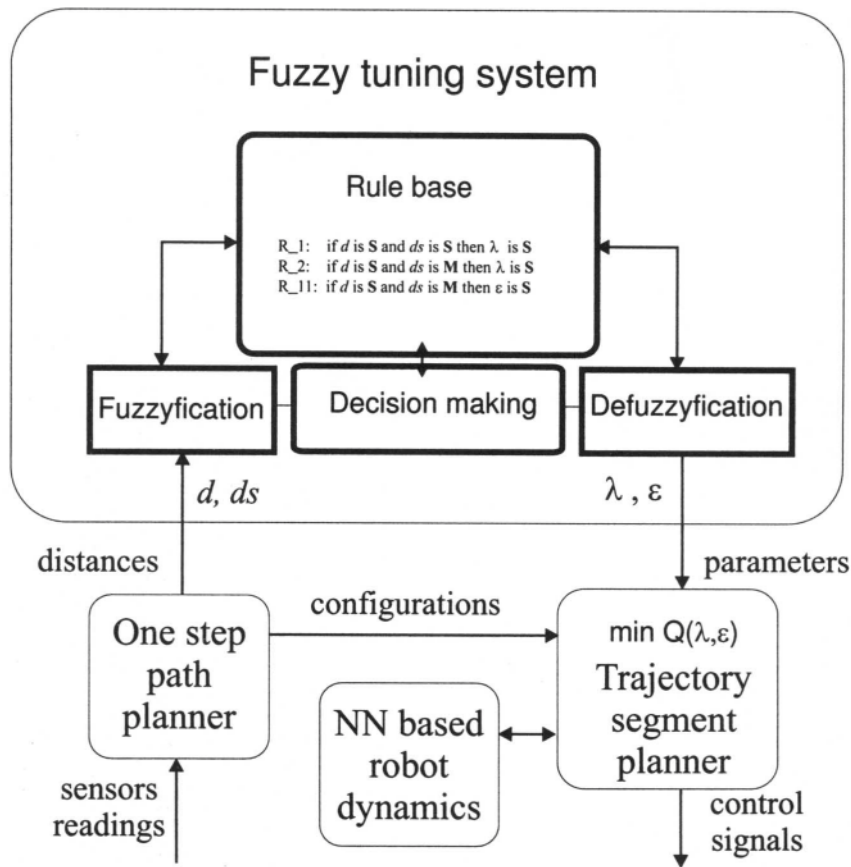


Figure 7.21. Structure of a fuzzy tuning system.

Table 7.1. The Coordinates of the Points for the Example Path

Point number	Cartesian position			Joint position		
	x	y	z	q_1	q_2	q_3
1	0.3128	-0.1963	0.5487	0.00	0.00	0.00
2	0.3201	-0.1859	0.5462	0.05	0.05	0.00
3	0.3274	-0.1755	0.5436	0.10	0.10	0.00
4	0.3348	-0.1647	0.5410	0.15	0.15	0.05
5	0.3423	-0.1537	0.5384	0.20	0.20	0.10
6	0.3499	-0.1423	0.5357	0.25	0.25	0.15
7	0.3541	-0.1329	0.5352	0.28	0.26	0.20
8	0.3453	-0.1468	0.5384	0.22	0.20	0.25
9	0.3381	-0.1580	0.5410	0.17	0.15	0.25
10	0.3309	-0.1689	0.5436	0.12	0.10	0.20
11	0.3237	-0.1795	0.5462	0.07	0.05	0.15

a straight-line segment back and forth in the work space. For greater clarity in the figures a small offset is put between the path points when tracing them back and forth. Table 7.1 presents the Cartesian coordinates of the path points to be followed and the corresponding points from joint space.

The first experiment illustrates the performance of the one-step trajectory planner while applying the jerk minimization criterion. Figure 7.22 shows the Cartesian position of the manipulator's end effector and the position and acceleration of the manipulator joint for the case of jerk minimization and precise path tracing, i.e., when $\lambda = 1$ and $\epsilon = 0$. As could be expected, for some types of paths considerably large overshoots occurred with use of this criterion, however, the jerks corresponding to the planned trajectory are small.

Now we show the behavior of the planner when the overshoot minimization criterion is used. In Figure 7.23 the Cartesian position and the joint position and acceleration are shown with the overshoot minimization criterion applied in the precise path tracing mode ($\lambda = 0$ and $\epsilon = 0$). With this criterion the planner planned trajectories as close to linear in joint space as possible; however, the controls needed to follow them take maximal values in short time intervals. That causes very large values of input jerks. This is extremely strenuous to the robot body.

We also considered the application to trajectory planning of the same overshoot minimization criterion but now allowing the joint to not pass exactly through all points on the path. The results for this case (i.e., $\lambda = 0$ and $\epsilon = 0.005$) are shown in Figure 7.24. In this case the number of acceleration switches is reduced drastically. A comparison of Figure 7.23 with Figure 7.24 shows how large an improvement can be achieved.

Remarks. The simulations performed on the one-step trajectory planner lead to the following conclusions:

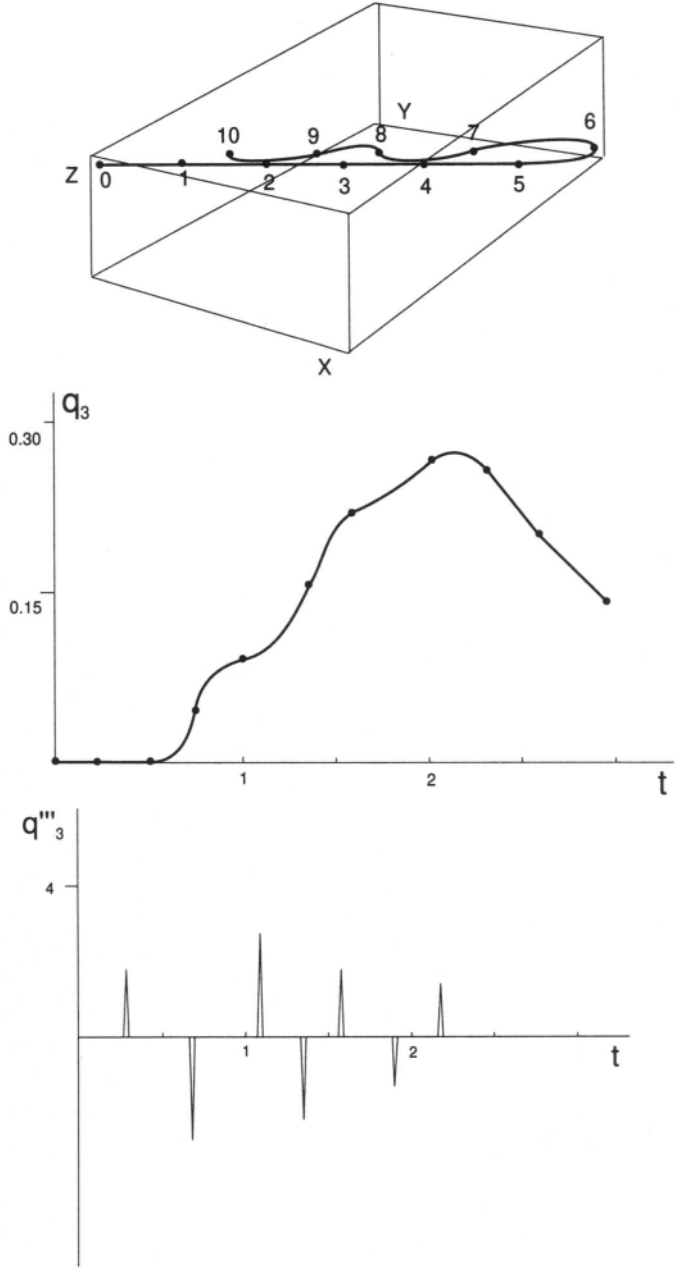


Figure 7.22. Cartesian position and joint position for jerks with $\lambda = 1$ and $\epsilon = 0$.

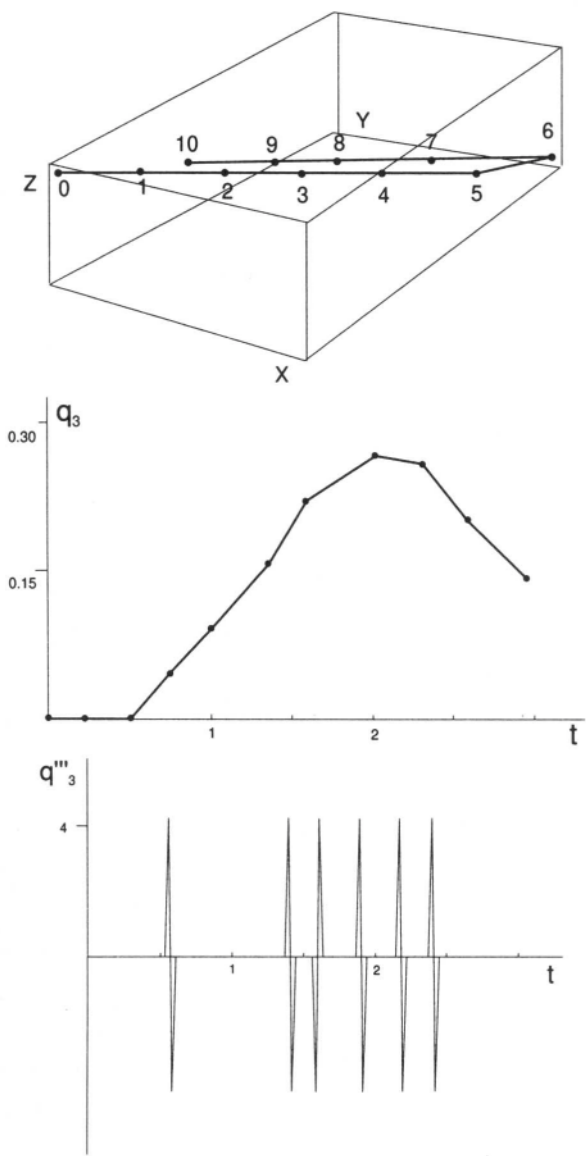


Figure 7.23. Cartesian position and joint position for jerks with $\lambda = 0$ and $\epsilon = 0$.

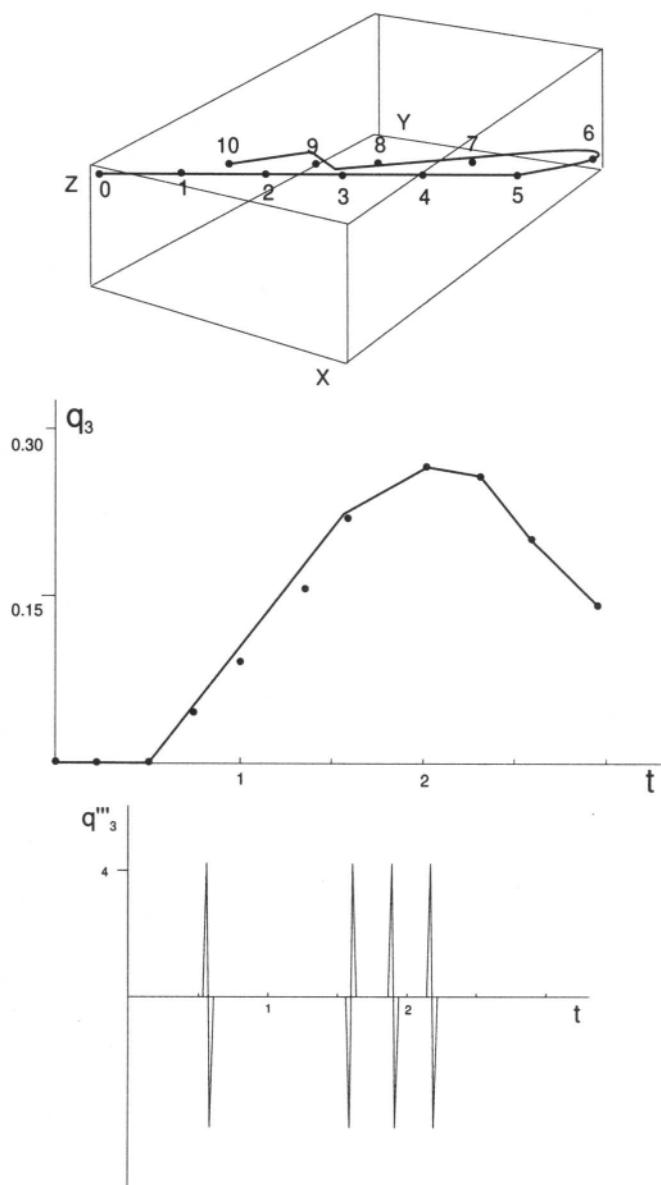


Figure 7.24. Cartesian position and joint position for jerks with $\lambda = 0$ and $\epsilon = 0.005$.

- The presented planning method allows us to plan robot trajectory segments realizing a desired path in real time
- The use of the pure jerk minimization criterion can lead to large overshoots between path points
- The disadvantage of applying the pure overshoot minimization criterion lies in large values of jerks occurring during the path tracing
- Allowing the joint to not pass exactly through all points on a path usually reduces the number of acceleration switches
- Combining jerk minimization with overshoot minimization yields trajectories with features between the two cases mentioned above. Appropriate choice of parameters (using fuzzy logic) yields trajectories with the prescribed features

7.4.5. Neural Computation-Based Reactive Executor of an Agent's Action in the Presence of Uncertainties

Since a robot dynamic is usually highly nonlinear and coupled, sophisticated methods have to be applied to control it. In the present application the trajectory preplanned by the one-step trajectory planner is executed by a typical computed torque controller (Spong and Vidyasagar, 1989) with PD loop. The main idea of such a controller is to apply a dynamic linearization module to obtain a dynamic is relatively simple to control.

7.4.5.1. Motion controller with passive learned neural dynamics. The inverse dynamics calculation unit applies a neural network-based model of the robot dynamics. The model is created for the nominal dynamics and then calibration is performed to improve the performance of the controller. The calibration is carried out in two stages: an *active mode* of learning and a *passive mode* of learning. The main difference between the two modes lies in the ability (active mode) or inability (passive mode) to perform active experiments on the robot, i.e., the robot can be controlled by allowable controls.

First, the active mode of neural net learning is used (for more details see Section 5.2). This step of the calibration yields the real dynamics of a robot with zero payload. Next, while performing a real task in a robotic workcell, the passive mode of learning is applied to modify the net coefficients. This has to be done because of the possibility of changes in the manipulator's work conditions (i.e., different payloads).

Such an approach allows us to train the neural network model to fit the actual robot dynamics during the control process.

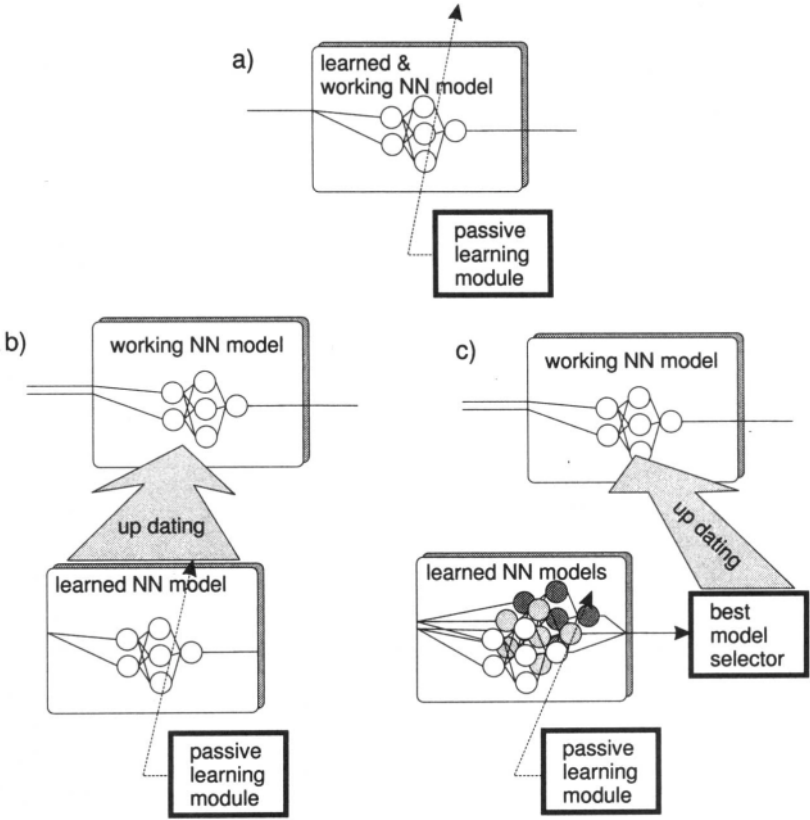


Figure 7.25. Architectures of learning strategies for neural net (NN) modeling of robot dynamics.

Generally, many architectures of neural network training can be used in the robot controller. The simplest one is when the net used for robot control is trained while performing a task (see Figure 7.25a). Another strategy is to train a copy of the working net and set up the working net when the process of training is completed. This situation is illustrated in Figure 7.25b).

Finally, a more advanced strategy is to apply a multinet updating mechanism. The main idea of this strategy consists in preparing a collection of several neural nets pretrained (using, for example, the active mode of learning) for the parameters of different gripped objects and using one of them in the controller. The selected net (for example, that which gives as output value the one closest to the actual robot input) is applied in one of the previous modes. Figure 7.25c presents this situation.

We use two approaches to identify the dynamic coefficients. Both take advan-

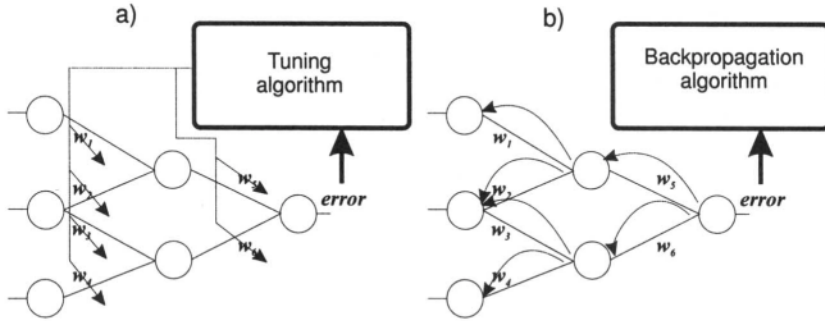


Figure 7.26. The neural net learning schemes with the use of (a) tuning and (b) backpropagation algorithms.

tage of the neural representation (Figure 7.26). The main difference between the two methods lies in the parallel (for tuning) and the sequential (for backpropagation) manner of the changes in net coefficients. For details of the algorithms and the methods used to test performance in the static mode see Section 5.2.

The computed torque (CT) controller for the robot dynamics is described by

$$M(q)\ddot{q} + \dot{q}^T N(q)\dot{q} + R\dot{q} + G(q) = F \quad (7.64)$$

with M , N , and G defined as in Equation (7.49) and R denoting the robot's viscous friction matrix; the controller is given by

$$\begin{cases} F = M(q)w + \dot{q}^T N(q)\dot{q} + R\dot{q} + G(q) \\ w = \ddot{q}_d - K_d(\dot{q} - \dot{q}_d) - K_p(q - q_d) \end{cases} \quad (7.65)$$

where q_d , \dot{q}_d , \ddot{q}_d are the desired position, velocity, and acceleration, respectively, and K_p , K_d are the controller gains, which are symmetric, positive-definite matrices (usually diagonal to get a set of simple, easy-to-check stability conditions).

The detailed structure of the controller is depicted in Figure 7.27.

Plugging Equations (7.65) into Equation (7.64) and denoting errors as

$$e = q - q_d, \quad \dot{e} = \dot{q} - \dot{q}_d, \quad \ddot{e} = \ddot{q} - \ddot{q}_d$$

we obtain the following error equation:

$$\ddot{e} + K_d \dot{e} + K_p e = 0 \quad (7.66)$$

It is an easy to see that the matrices K_p and K_d being symmetric and positive definite is a necessary and sufficient condition for asymptotic stability of Equation (7.66) and thus for the stability of the control system given by Equations (7.65).

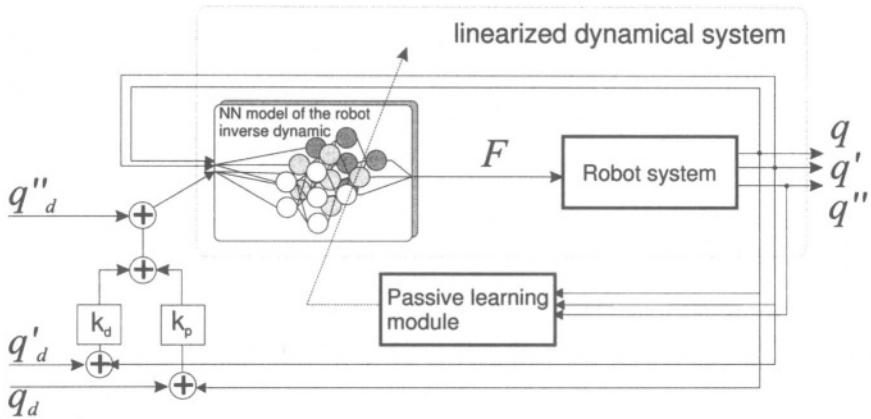


Figure 7.27. Computed torque controller with a neural model of the inverse dynamics.

7.4.5.2. The performance analysis. The performance of the computed torque controller with a neural model of the robot dynamics will be tested for the same Puma-type robot manipulator as the one-step trajectory planner (see Figure 5.20).

Example 7.4.2. The trajectory preplanned by the one-step trajectory planner (see Example 7.4.1) will be executed by the control system. The main aims of the performed simulation are to evaluate methods for neural network training presented in Section 5.2 in dynamic circumstances, and to observe phenomena which do not arise in static circumstances.

We present first the behavior of a typical computed torque controller in which no adaptation processes are applied, and then a set of various simulations with the CT controller based on the neural model of inverse dynamics. The aim of the first simulation is to create a basis for comparing the typical computed torque controller (i.e., in which a nominal dynamics model without any calibration and adaptation is applied) with the controller proposed above. Figure 7.28 shows how the computed torque controller with nominal dynamics works in driving a real manipulator without any payload, slightly different from nominal one. It is a straightforward observation that such a controller is unable to control the robot properly. Additional steps need to be taken to improve controller performance.

Now we show the behavior of the same controller as in the previous simulation, when the real dynamics differs from the nominal dynamics (calculated for zero payload), but with the controller using the neural-computed and calibrated model of the robot dynamics. We assume that the robot carries an object at time 0.7 sec. Figure 7.29 shows the performance of the controller when a payload is gripped. Such a system works well when there are no disturbances in the robot dynamics, but it cannot deal well with them. One can easily observe that there is a large difference between the executed and desired trajectories.

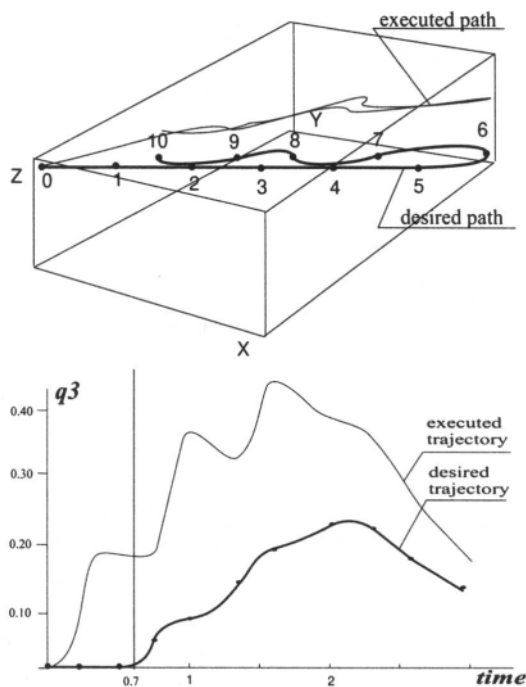


Figure 7.28. Cartesian position and joint control obtained with a typical CT controller with standard calculated dynamics without any calibration.

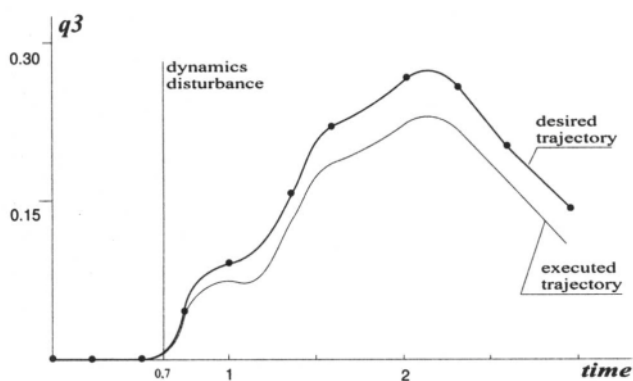


Figure 7.29. Joint control obtained with the CT controller with a calibrated neural model of the robot dynamics and with a real dynamic disturbance (without passive learning).

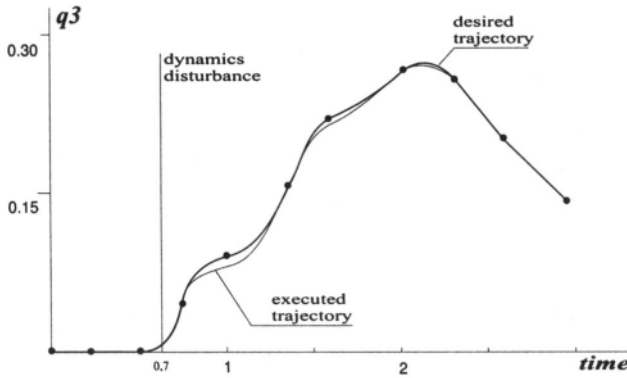


Figure 7.30. Joint control obtained with the CT controller with a neural model of the robot dynamics and with a real dynamic disturbance while applying passive learning.

To avoid tracing errors as in the previous simulation, the passive learning of the neural network can be applied. As before, the robot grips a payload at time 0.7. The performance of the system for disturbed robot dynamics is shown in Figure 7.30. One can see a large improvement in the system properties. There is a small error in the trajectory tracing just after the dynamic disturbance occurs, but thanks to the fast adaptation of the dynamics model to the new conditions, it goes to zero rapidly.

In addition to the above investigations, we show the results of a simulation in which we forced network learning to start with a delay. The other simulation conditions are the same as in the previous simulations. The results of this run are illustrated in Figure 7.31. This simulation shows how important it is to train the

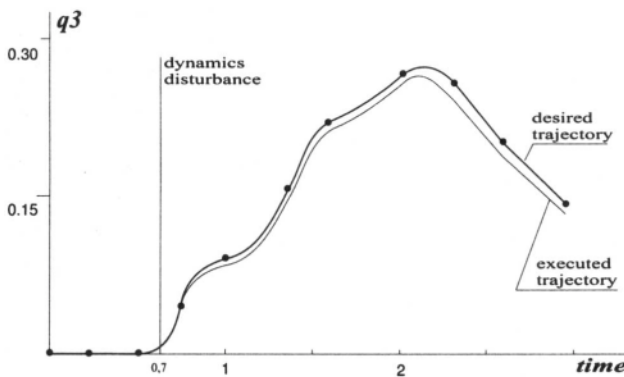


Figure 7.31. Joint control obtained with the CT controller with a neural model of the robot dynamics and with a real dynamic disturbance while applying delayed passive learning.

network to new conditions in a very short time after disturbances occur. It is worth noting that in spite of the fact that the dynamics was learned properly, an error which appeared just after gripping an object stayed at almost the same level until the end of the trajectory.

Remarks. The following remarks can be made based on the performed simulations:

- The performance of the computed torque controller with a neural model of the robot dynamics is already good enough for robotic arm control.
- The passive learning method based on tuning seems to be better than the passive learning method based on a backpropagation algorithm.
- To perform the learning process for tuning it is enough to have information in the form of a point in the state space, while for the backpropagation algorithm information should be collected over an interval of time.
- The necessity of collecting patterns for the backpropagation algorithm causes the period of time to be long between the occurrence of a disturbance and the moment when the neural model fits well to the real is long.
- It is difficult to construct a good strategy of transferring error from the output of the inverse dynamics model to the outputs of neural nets modeling the dynamics coefficients.
- The period of time between the occurrence of a disturbance and the moment when the neural model fits well to the real dynamics should be as short as possible. Otherwise an unavoidable error occurs (see Figure 7.31).
- If it is impossible to adopt the neural network to new conditions quickly, an integrating part needs to be added to the PD loop. This part would work only for a short time after the occurrence of a disturbance to reduce the constant part of the error.
- To speed up the learning, we can use network updating based on either internal controller signals or external signals. The weights of the neural network dynamic models are stored for several different manipulators payloads, and in the case of obtaining an external signal the proper set of weights is updated.

This page intentionally left blank.

CHAPTER 8

The Coordination Level of a Multiagent Robotic System

The activation of each machine is caused by an external event generated by the model of a robot. The events generated by the cell's robots depend on the states of the workstations d_i and the given fundamental plan $\wp \in \text{PROCESSES}$. The coordination level of the intelligent control system of a robotic cell consists of two main components, the acceptor and the cell controller.

8.1. Acceptor: Workcell State Recognizer

We define a system (called an *acceptor*) which observes and recognizes the states of each workstation. The acceptor acts as a filter. Only states waiting for the processing cell are recognized and transferred to the cell controller. Other states are ignored. The acceptor is defined as the following discrete event system (Jacak and Rozenblit, 1992b; Jacak and Rozenblit, in press):

$$A = (X_A, \text{PROCESSES}, Y_A, \lambda^A) \quad (8.1)$$

Here

- $X_A = \{(s, t) | s \in S_{\text{cell}} \wedge t \in \text{Time}\}$ where $S_{\text{cell}} = S(d_1) \times S(d_2) \times \dots \times S(d_D) \times S(m_1) \times \dots \times S(m_M)$ is the general state set of robotic cells and *Time* is the time base.
- PROCESSES is the set of technological processes \wp generated by the route planner.
- Y_A is the output set, i.e., the set of accepted states,
- λ^A is an input-output function.

The role of the acceptor is to select the global events which will lead the robots to service the workstations. Let us define the set of acceptor outputs as the set of subsets of operations and position indexes, i.e.,

$$Y_A \subset 2^{AOP}$$

where the set $AOP = O \times ((D \cup M) \times N)^2$, and N denotes the set of natural numbers. More exactly, if $y \in AOP$, then $y = (a, (w, i), (v, j))$, which means “the operation a waits on the device w in the position i for transport to the device v in the position j .”

The acceptor’s input–output function is given by

$$\lambda^A : X_A \times \text{PROCESSES} \rightarrow Y_A$$

and

$$\lambda^A((s, t), \wp) = \{(a_k, (w, i), (v, j)) \mid \text{Activ}(a_k, s) = \text{TRUE} \wedge a_k \in \wp\}$$

where $s = (s(d_1), s(d_2), \dots, s(d_D), s(m_1), \dots, s(m_M))$.

The predicate $\text{Activ}(a_k, s)$ determines which operation from $\wp \in \text{PROCESSES}$ requires service when the cell’s state is equal to s , namely:

For $a_k \in \wp \in \text{PROCESSES}$ and $s \in S_{\text{cell}}$ the predicate $\text{Activ}(a_k, s) = \text{TRUE}$ iff one of following conditions holds:

- i. $\alpha(a_k) = v$ and for $s(v) = (s_v, b_v)$ the j th coordinate $b_{vj} = (0, 0)$ and $\alpha(a_{k-1}) = w$, and for $s(w) = (s_w, b_w)$ the i th coordinate $b_{wi} = (b_{k-1}, 1)$, and deadlock-avoidance conditions (4.9) for the workstation v are satisfied, i.e., $\text{Avoid_Dead}(v) = \text{TRUE}$.
- ii. $\alpha(a_k) = v$ and for $s(v) = (s_{vd}, b_v)$ the j th coordinate $b_{vj} = (0, 0)$ and $\alpha(a_{k-1}) = \{u, w\}$, and for all coordinates $b_{ui} \neq (a_{k-1}, 1)$ and for $s(w)$ the i th coordinate $s_{wi} = a_{k-1}$, and deadlock-avoidance conditions (4.9) for the workstation v are satisfied, i.e., $\text{Avoid_Dead}(v) = \text{TRUE}$.
- iii. $\alpha(a_k) = \{w, v\}$ and for $s(w) = (s_w, b_w)$ the i th coordinate $b_{wi} = (a_k, 1)$ and $\alpha(a_{k+1}) = u$, and for all coordinates $b_{uj} \neq (0, 0)$, and for $s(v)$ the j th coordinate $s_{vj} = 0$

$\text{Activ}(a_k, s) = \text{FALSE}$ for all other cases.

The output of the acceptor can be an empty set or it contain indexes of only those operations which can be executed without deadlock.

Not all operations can be performed simultaneously. Therefore, to select the operations which can be executed concurrently, we introduce a discrete event system called the *cell coordinator and controller*.

This system can be constructed based on different principles, depending on the intelligence of the robotic workcell components. The most popular industrial coordination system concentrates the decision-making power at a higher level, i.e., the coordination level. The role of the components of the cell is reduced to the nonautonomous execution of the coordinator's decisions. Such a system is called a *centralized robotic system coordinator*.

In modern industrial practice the decision-making power tends to be distributed among all workcell components. The components have become more autonomous and can adapt their behavior to changes in the surrounding environment. Such a workcell coordination system is called a *distributed robotic system coordinator*. This system needs only partial knowledge about the workcell behavior.

In the extreme case the system has very poor knowledge about the surrounding environment (for example, a system which acts in the real world). In this case the system component (i.e., robotic agent) should be extremely autonomous and should be able to self-replan and to execute its actions in a reactive way.

8.2. Centralized Robotic System Coordinator

The centralized robotic system coordinator needs full knowledge about the workcell and the possible changes of its states.

The cell controller is a discrete event system defined as follows (Jacak and Rozenblit, 1994; Jacak and Rozenblit, 1993):

$$\text{Cell_Contr} = (S(c), X(c), \delta_{\text{ext}}^c, \delta_{\text{int}}^c, \{Z_r\}) \quad (8.2)$$

where

- $S(c) = Y_A \times R_f$ is the state set where Y_A is the acceptor output and $R_f \subset 2^R$ is the set of unemployed robots, i.e., $s(c) = (\lambda^A((s, t), \varphi), r_f)$.
- $X(c) = \{\text{done}(r) | r \in R\}$ is the external event set.
- δ_{int}^c is a set of parametrized internal state-transition functions:

$$\begin{aligned} \delta_{\text{int}}^c &= \{\delta_{\text{int}}^u | u \in U\} \\ \delta_{\text{int}}^u &: S(c) \rightarrow S(c) \end{aligned}$$

where U is the strategy set (see Chapter 9).

- δ_{ext}^c is an external state-transition function:

$$\delta_{\text{ext}}^c : S(c) \times X(c) \rightarrow S(c)$$

- $\{Z_r\}$ is the set of functions which generate the external events for the robots

$$\{Z_r\} = \{Z_r : S(c) \rightarrow X(r) | r \in R\}$$

8.2.1. Coordinator of Nonautonomous Actions of the Robotic Agent

The principal task in the synthesis of the robotic cell coordinator is the establishment of the internal state transition function δ_{int} . This function is mathematical model of control law for coordination activity.

The internal state transition function chooses from the set of waiting operations (given by the cell acceptor) only that subset of operations which are simultaneously realizable without collisions among robots and have the greatest priority with respect to the strategy given by the organization level.

The transition function δ_{int} acts on the acceptor outputs in the following steps: Let $y = \lambda^A((s, t), \varphi) \in Y_A$ be the operation set waiting for servicing which is generated in the acceptor output. A state of the cell controller is given by $s(c) = (y, r_f)$, where r_f is the set of unemployed robotic agents.

8.2.1.1. Internal selection rule: Elimination by collision prevention. In the first phase the nonrealizable operations from the operation set given by the acceptor output should be eliminated, i.e., from the set of operations we should choose a subset for which the robot actions do not result in a collision.

$$y_{\text{exe}} = \{(a_k, (w, i), (v, j)) \in y \mid (\exists r \in r_f) (\{w, v\} \subset \text{Group}(r)) \wedge (\forall r' \in \tilde{r}_f) (\neg r \kappa r')\}\}$$

Here $\tilde{r}_f = R - r_f$ and the relation κ denotes the collision relation (see next paragraph). From the set of operations y one chooses a subset for which the robot actions do not result in a collision between currently active robots.

Collision relation. In general, a collision relation between two robots r and r' can be defined as

$$r \kappa r' \Leftrightarrow (\exists t) (Vol_{q(r)}(t) \cap Vol_{q(r')}(t) \neq \emptyset) \quad (8.3)$$

where $Vol_{q(r)} : \text{Time} \rightarrow 2^{E_0}$ denotes the time function describing the robot's movement in the Cartesian base frame E_0 when it realizes the trajectory $q(r)$; and $Vol_{q(r)}(t)$ is the volume occupied by robot r in configuration $q(t)$ when the time trajectory $q_{\text{trac}}^r \in TR(r)$ [Equation (5.104)] is realized.

Hence, $r \kappa r'$ means that the simultaneous actions of robots r, r' lead to a collision.

The above definition is too complex for testing in real time, and for this reason we usually use the *gross approximation of the robot track* [Equation (5.60)] to establish the collision-free condition between two moving robots. Recall that for a given track q_{track}^r we can establish the subset of raster elements (voxels) which are visited by the robot manipulator during the motion $Tor^r(q_{\text{track}})$ (Section 5.1).

This makes it possible to create a new relation defining the collision-free condition between two robots r, r' . Let robots r and r' realize the trajectories q_{track}^r and $q_{\text{track}}^{r'}$, respectively.

Definition 8.2.1 (Strong collision-free condition). The robots $r, r' (r \neq r')$ are strongly collision-free along their trajectories if

$$r \text{ Col_Free } r' \Leftrightarrow \text{Tor}'(q_{\text{track}}) \cap \text{Tor}'(q_{\text{track}}^{r'}) = \emptyset$$

where $\text{Tor}(q)$ is defined by Equation (5.60).

Such a strong collision-free condition is only a sufficient condition for general collision-free relation (Definition 8.3).

Fact 8.2.1. *It is clear that*

$$r \text{ Col_Free } r' \Rightarrow \neg r \kappa r'$$

and

$$\text{Srv_Sp}(r) \cap \text{Srv_Sp}(r') = \emptyset \Rightarrow r \text{ Col_Free } r'$$

where $\text{Srv_Sp}(r) \subset E_o$ denotes the service space of robot r .

□

□

When $\neg r \text{ Col_Free } r'$, it does not mean that $(\exists t)(\text{Vol}_{q(r)}(t) \cap \text{Vol}_{q'(r')}(t) \neq \emptyset)$, because the common voxels can be visited by the robots r and r' at different moments of time.

An example of a raster representation of robot trajectories is shown in Figure 8.1. The black voxels represent the collision subspace of the service space of both robots r and r' . We assume that the job of robot r' has a greater priority than the job of robot r . In this case, robot r cannot be activated until robot r' has finished its motion. Such a strategy of collision prevention leads to additional waiting time by robot r .

To avoid this, we apply priority-based *Start–Stop* synchronization of the robot motions. The robot with greater priority r' realizes its trajectory without testing for possible collisions. The robot with lower priority r moves only to the boundary of the collision subspace and waits there until this subspace becomes free. Then it continues with the rest of the trajectory. In this case the current robot positions need to be tested. An example of priority-based start–stop synchronization is presented in Figure 8.2. Another, more complex method for testing for collision-freeness can be found in (Latombe, 1991).

The strong collision-free relation can be used to perform the first step of calculating the transition function δ_{int} .

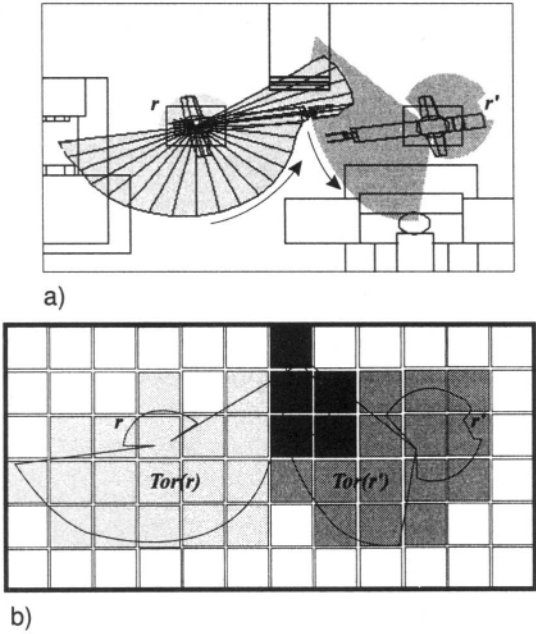


Figure 8.1. Raster model of collision subspace.

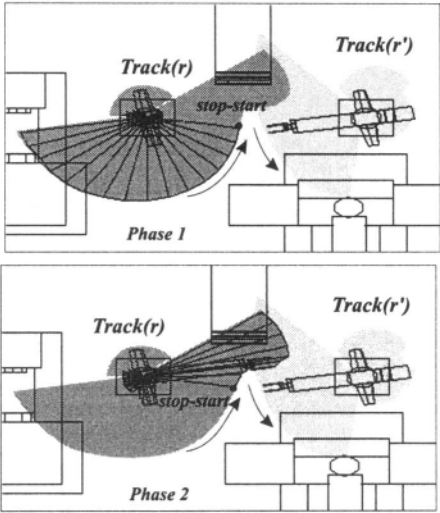


Figure 8.2. Priority-based start-stop synchronization.

8.2.1.2. *External selection rule: Priority.* For the set \mathcal{Y}_{exe} , only operations which have the highest priority should be chosen. Each function δ_{int} is parametrized by the external parameter u which is loaded into the cell controller from a higher level of control, namely from the workcell organizer (see Chapter 9). The parameter $u \in \mathcal{U}$ is the operation's choice function, and represent the priority strategy of workcell CARC [Equation (9.6)]. The strategy u defines the fuzzy set over the set of technological operations

$$u : \mathcal{O} \rightarrow \{(\sigma, \mu_u(\sigma)) | \sigma \in \mathcal{O}\}$$

The choice function u defines the membership function for the operation set which is created by the fuzzy organizer of CARC. The selection function u defines the type of priority rule under which operations are chosen to be processed from the set \mathcal{Y}_{exe} .

To perform the operation selection we find the one-to-one partial function σ_u which assigns the operation to unemployed robots. More exactly we construct the partial function

$$\sigma_u : \mathcal{Y}_{\text{exe}} \rightarrow \mathcal{R}_f \quad \text{and} \quad [(\sigma_u \downarrow) \subset \mathcal{Y}_{\text{exe}}] \quad (8.4)$$

with the following constraints:

- i. There must exist a robot r which services the devices realizing the chosen operation

$$((a_k, (w, i), (v, j)), r) \in \sigma_u \Rightarrow \{w, v\} \subset \text{Group}(r)$$

- ii. σ_u is a one-to-one function, i.e., one robot can service only one operation

$$\sigma_u \circ \sigma_u^{-1} = \text{id}$$

- iii. A maximum number of operations should be executed simultaneously

$$\overline{\overline{\sigma_u}} \rightarrow \max$$

- iv. For the selected subset of operations the robot actions do not result in a collision, i.e.,

$$(\forall r, r' \in [\sigma_u \uparrow])(\neg r \kappa r')$$

The collision-freeness can be tested by using the strong collision-freeness relation Col_Free .

- v. The selected operations should have the highest priority, i.e.,

$$\sum_{x \in [\sigma_u \downarrow]} \mu_u(x) \rightarrow \max$$

This rule represents an external selection strategy which applies to the set of all operations waiting to be executed. The priority strategy u is loaded into the cell controller from the organization level.

The symbols $[\sigma_u \downarrow]$ and $[\sigma_u \uparrow]$ denote the domain and codomain of the function σ_u , respectively.

Based on such a function σ_u we define the state transition function δ_{int}^u as follows:

$$\delta_{int}^u(y, r_f) = ([\sigma_u \downarrow], \quad r_f - [\sigma_u \uparrow]) \quad (8.5)$$

Fact 8.2.2. *It is easy to observe that if*

$$\delta_{int}^u(y, r_f) = (y', r_f')$$

then

$$y' \subset y \quad \text{and} \quad r_f' \subset r_f$$

□

□

In the case of nonautonomous robotic agents, the coordination level should be able not only to select the most important operation, but also to prevent collision among active robots. For this reason the knowledge base contains off-line preplanned phase trajectories of the motions for each agent.

The centralized structure of the coordinator of nonautonomous agents is shown in Figure 8.3.

8.2.2. Interaction between Robotic Agents — Coordinator

The interaction between cell controller and robots (nonautonomous agents) is modeled by an external state transition function of DEVS and a set of functions $\{Z_r\}$. The external transition function δ_{ext} is defined below.

Let $s(c) = (y, r_f) \in S(c)$; then

$$\delta_{ext}((s(c), t), \text{done}(r)) = (y, r_f \cup \{r\}) \quad \text{for} \quad \text{done}(r) \in X(c)$$

The last component of *Cell-Contr* is the set of functions which generate external events for robots,

$$\{Z_r\} = \{Z_r : S(c) \rightarrow X(r) | r \in R\}$$

The function Z_r is defined as follows:

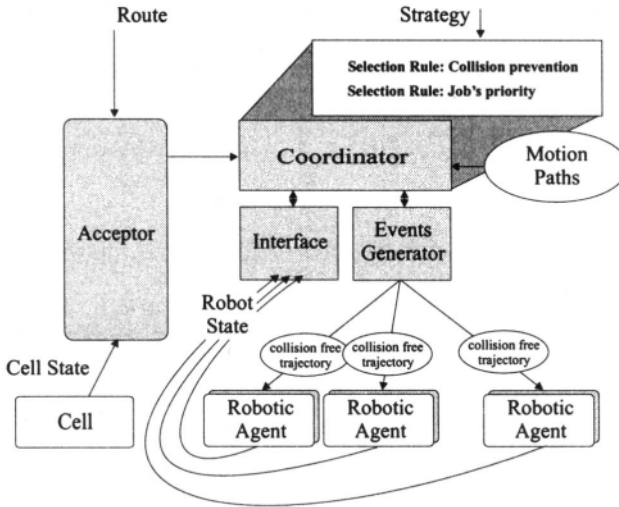


Figure 8.3. Centralized coordinator of robotic agent actions.

$$\text{For } s(c) = ([\sigma_u \downarrow], r_f - [\sigma_u \uparrow])$$

$$Z_r(s(c)) = e_r(a_k(w, i), (v, j))$$

where $r = \sigma_u(a_k(w, i), (v, j))$.

This completes the cell's discrete event model specification. The general structure of the cell levels is shown in Figure 8.4.

8.3. Distributed Robotic System Coordinator

The coordination system presented in this section allows distribution of the computational effort usually carried out by the coordinator of a robotic multiagent system to the agents themselves. By using intelligent robotic agents, the intelligence can therefore be shifted down from the coordinator level to the individual manipulators. A further characteristic of such autonomous manipulators is their ability to directly *react to the environment* without having to contact a supervising system (see Section 7.4).

8.3.1. Distributed Control of Autonomous Agents

The cell controller, as in the previous case, is discrete event system:

$$\text{Cell_Contr} = (S(c), X(c), \delta_{\text{ext}}^c, \delta_{\text{int}}^c, \{Z_r\}) \quad (8.6)$$

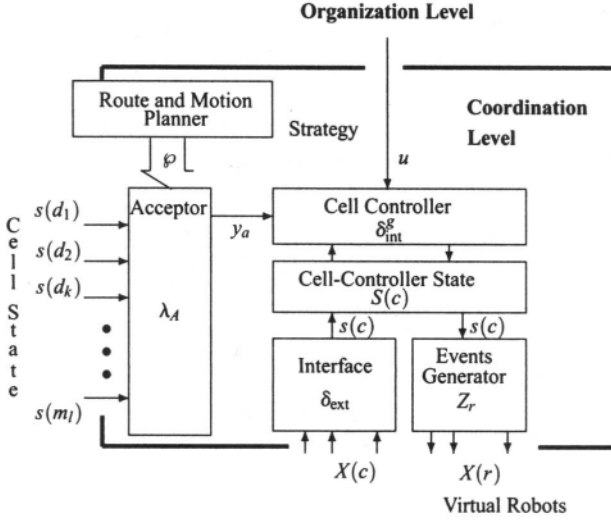


Figure 8.4. DEVS model of CARC coordination level.

The internal state transition function δ_{int}^c establishes the coordination law of the autonomous agents. We assume that each robotic agent is equipped in an *intelligent reactive control system*. Recall that such a robot is able to plan a trajectory of motion in real time, and this trajectory is adapted according to the current sensor readings (see Section 7.4).

Therefore, the internal state transition function δ_{int}^c of a distributed robotic system coordinator chooses from the set of operations (given by the cell acceptor) only that subset of operations which have the greatest priority with respect to the strategy given by the organization level. Then the calculation of the state transition function is based only on the modified external selection rule.

Let $y = \lambda^A((s, t), \phi) \in Y_A$ be the set of operations waiting for servicing, which is generated in the acceptor output, and let the state of the cell controller be $s(c) = (y, r_f)$.

8.3.1.1. External selection rule: Priority. For the set y , only those operations should be chosen which have the highest priority. Each function δ_{int}^c is parametrized by the external parameter u , which is loaded into cell controller from a higher level of control, namely from the workcell organizer. The selection function u defines the type of priority rule under which the operations are chosen to be processed from the set y .

To perform the operation selection we find the one-to-one partial function σ_u which assigns the operation to the unemployed robots. More exactly we construct

the partial function

$$\sigma_u \subset \mathcal{Y} \times \mathcal{R}_f$$

with the following four constraints:

- i. There must exist a robot which services the devices realizing the chosen operation

$$((a_k, (w, i), (v, j)), r) \in \sigma_u \Rightarrow \{w, v\} \subset \text{Group}(r)$$

- ii. σ_u is one-to-one function, i.e., one robot can service only one operation

$$\sigma_u \circ \sigma_u^{-1} = \iota$$

- iii. A maximum number of operations should be executed simultaneously

$$\overline{\overline{\sigma_u}} \rightarrow \max$$

- iv. The selected operations should have the highest priority, i.e.,

$$\sum_{x \in \{\sigma_u\}} \mu_u(x) \rightarrow \max$$

The priority measure μ is loaded into the cell controller from the organization level.

Based on such a function σ_u we define the state transition function δ_{int}^u as in the previous case [Equation (8.5)]. The distributed structure of the coordinator of the autonomous agents is shown in Figure 8.5. In contrast to centralized coordination, the autonomous agent obtains from the coordinator only information about the pick and place positions for the transfer operation and the agent itself plans the collision free dynamic trajectory which realizes the transfer operation.

8.4. Lifelong-Learning-Based Coordinator of Real-World Robotic Systems

The development of intelligent knowledge-based autonomous agents that learn by themselves to perform complex real-world tasks is an open challenge for system and control theory, robotics, and artificial intelligence.

In this section we present the concept of an *autonomous robotic agent* that is capable of showing *machine learning-based* and *reactive* behavior. Experience has shown that especially in real world acting domains neither purely reactive

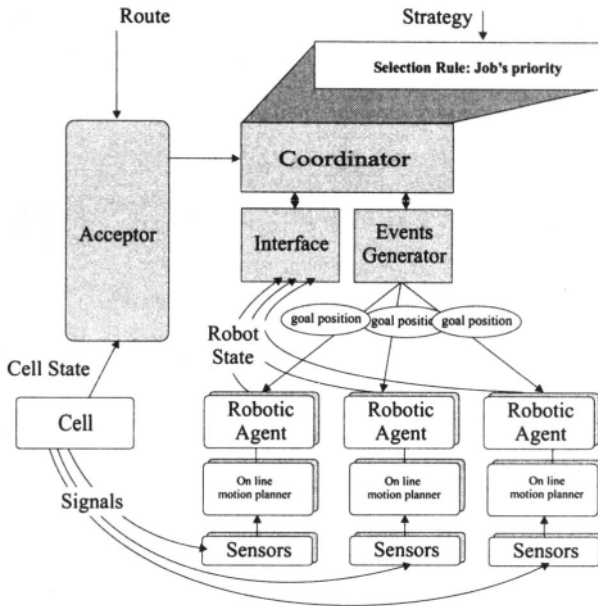


Figure 8.5. Distributed coordinator of the actions of autonomous robotic agents.

nor purely machine learning-based approaches suffice to meet the requirements imposed by the environment.

In multiagent robotic systems, one is primarily interested in the behavior and interactions of a group of agents based on the models of the agents themselves. The idea of the reactive model is to specify an agent by its behavior, i.e., the way the agent reacts to certain environmental stimuli. With every perceptual input one associates a certain action, an effector output, which is expressed in the form of rules or procedures that calculate the reaction of the agent. Reactive systems have no internal history or long-term plans, but calculate or choose their next action solely upon the current perceptual situation (Jacak and Buchberger, 1996b).

On the other hand, machine learning-based models are motivated by the representation of the system's knowledge. The adoption of symbolic AI techniques has led to the introduction of beliefs and intentions into the reasoning processes of the system. Intentions enable a system to reason about its state and the state of the environment. Such cognitive models permit the use of more powerful and more general methods than reactive models, though they have inadequacies for real-time applications.

Designing an autonomous system requires prior knowledge about the system itself, its environment, and the task it is to perform. Some knowledge is usually easy to obtain, but other knowledge might not be accessible or very hard to obtain.

Usually an agent has only partial information about the state of the as world obtained by its perception system. Machine learning aims to overcome limitations such as *knowledge bottlenecks and engineering and tractability bottlenecks* (Thrun, 1994) by enabling an agent to collect its knowledge on the fly, through real-world experimentation. More complex tasks require more training data to achieve a task. The collection of training data is an expensive undertaking due to the slowness of the system's hardware. Thus, the time required for real-world experimentation has frequently been found to be a limiting factor in rigorous machine learning techniques. The task of learning from real-world experiments can be simplified by considering a system that encounters collections of control learning problems over its entire lifetime.

8.4.1. *Structure of Lifelong-Learning-Based Coordinator and Controller*

The section gives the concept of an intelligent controller of robotic agents that uses both machine learning and reactive behavior. Machine learning is used to collect information about the environment and to plan robot actions based on this information. Processing, storing and using information obtained during several task executions is called lifelong learning. Reactive behavior is needed to execute actions in a dynamically changing environment. The structure of such robotic agent (presented in Figure 8.6) consists of two independent subsystems: the action planning system (coordination level), and the action execution system (control level).

The coordination of the activities of these subsystems is realized by exchange of messages via a communication channel.

In classical approaches (Geneserth and Nilsson, 1987) we assume that the full model of the agent and its influence on the surrounding world is known. More exactly, the set of world states S_{World} is given and the model of the agent's activities is specified by two functions: the action function do and the observation function see :

$$\text{do} : S_{\text{World}} \times A \rightarrow S_{\text{World}} \quad (8.7)$$

$$\text{see} : S_{\text{World}} \rightarrow O \quad (8.8)$$

where A is the action (control) set and O is the observation set obtained from perception subsystem of the agent.

There are various methods for specifying the model of an agent, based on the use of graph searching and resolution procedures (Geneserth and Nilsson, 1987), for finding the sequence of actions

$$\text{fundamental plan} = a^* = (a_1, a_2, \dots, a_n)$$

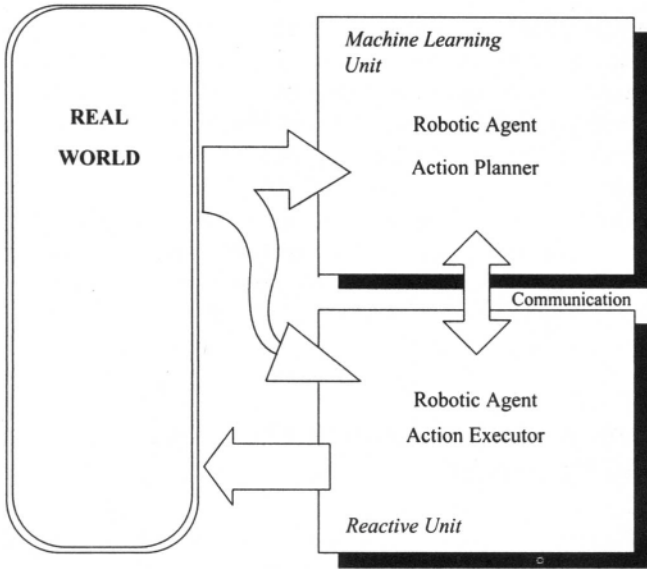


Figure 8.6. Components of a intelligent robotic agent.

called *the fundamental plan* (see Chapter 4), by which the agent goes from the current state to the given goal state. One such approach, based on a finite state machine formalism, is presented in (Sierocki, 1996).

In contrast to the above approaches, we assume that an agent has no prior information about the state of its environment, so it cannot establish the world state observation function see: $S_{\text{World}} \rightarrow O$.

Since the observation space of the agent's perception subsystem O consists of the set of all possible vectors which are collections of appropriate sensor signals, this space will be mapped to (interpreted as) a "conceptual" state space S of the real world. To this aim, one has to construct the generalization function

$$\text{gen} : O \rightarrow S \quad (8.9)$$

which maps any observation vector $o \in O$ to a conceptual world state class $s \in S$ in such a way that the closer points are in O , the closer their corresponding classes will be in S , with neighboring points in O being mapped to the same conceptual world state class. This property of the mapping is called *local generalization* (Han and Zhang, 1994).

Having found a generalized state in the agent's conceptual space, one has to find an agent's action sequence leading it toward its goal. This can be done during the action planning stage, when the next action a_{i+1} can be determined from the

previous action a_i and the conceptual state s_{i+1} with the function

$$\mathbf{plan} : S \times A \rightarrow A \quad (8.10)$$

as the action that maximizes the value function $\mathcal{Q}(s, a)$ (Thrun, 1994):

$$a_{i+1} = \mathbf{plan}(s_{i+1}, a_i) = \mathbf{argmax}_{a' \in A} \mathcal{Q}(\mathbf{do}(s_i, a_i), a')$$

where:

- The value function $\mathcal{Q}(s, a)$ is a function that returns scalar utility values. These values have to be learned over the whole lifetime of the agent while it is acting in the same environment. This learning can be accomplished with the Q-learning method (Thrun, 1994), a popular method for learning to select actions from delayed or sparse reward (Section 8.4.4).
- The function \mathbf{do} models the effect of an agent's behavior on the conceptual state space as

$$\mathbf{do} : S \times A \rightarrow S \quad (8.11)$$

One can see that the machine-learning-based behavior of an autonomous agent can be completely characterized by the three functions \mathbf{gen} , \mathbf{do} , and \mathcal{Q} . During the action planning stage, these three functions interact in the following way:

1. Start by obtaining a conceptual state s_i from \mathbf{gen} as

$$s_i = \mathbf{gen}(o_i)$$

where o_i is the current observation vector.

2. Obtain the next action a_i from \mathcal{Q} as

$$a_i = \mathbf{argmax}_{a' \in A} \mathcal{Q}(\mathbf{do}(s_{i-1}, a_{i-1}), a')$$

3. Obtain the next conceptual state s_{i+1} as

$$s_{i+1} = \mathbf{do}(s_i, a_i)$$

4. If the goal is not reached, let $i \leftarrow i + 1$ and go to step 2.

The advantage of this approach to action planning is that the sequences of actions and conceptual states can be derived without moving the agent. The agent controller consists of five main parts:

- The *conceptual world state generalization* module, which generalizes the agent's sensor system observations

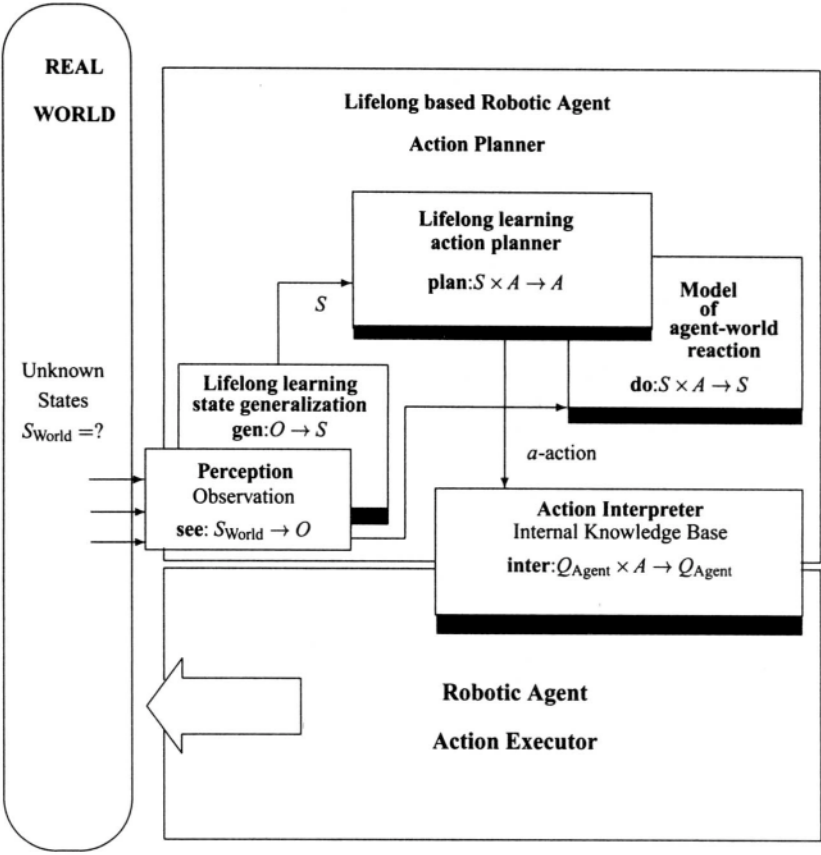


Figure 8.7. Structure of an autonomous agent based on the lifelong-learning approach.

- The *action planner*, which finds sequences of agent actions realizing desired goals
- the *modeler of the agent behavior on the environment*, which models the interaction between the agent and the conceptual world
- The *action interpreter*, which transforms elementary agent actions to goal states in its joint space
- The *action executor*, which calculates controls that drive the agent toward its goal state

The structure of such an intelligent agent is shown in Figure 8.7. The components of the autonomous agent that implement the reactive behavior (the action

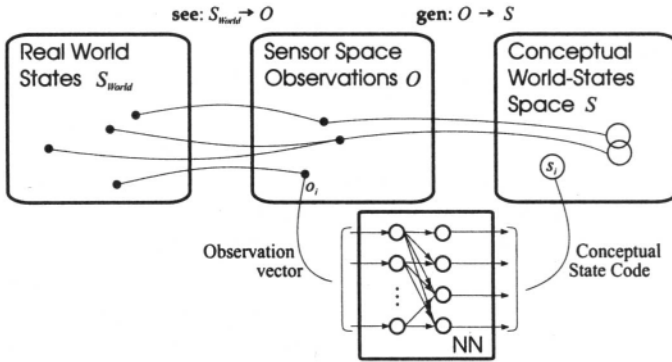


Figure 8.8. Generalization process.

interpreter **inter** and the action execution controller **contr**; see Figure 8.7) are described in the next section.

8.4.2. Conceptual State Space of the Real World

One of the main components of the machine learning layer in an autonomous robotic agent is the module for building a function that gives conceptual states from sensor data. Recall that in our structure of the autonomous agent based on the lifelong learning approach (see Figure 8.7) we have only incomplete knowledge about the environment and are thus unable to establish the world state observation function **see**: $S_{\text{World}} \rightarrow O$. The only information available to us comes from the m -dimensional vectors of the perception subsystem that consists of the external sensor signals.

The goal is thus to construct a world state generalization function **gen**: $O \rightarrow S$ that forms conceptual states as categories of observation vectors. This leads to a clustering of the observation vectors, with vectors in the same cluster being mapped to the same conceptual state by the function **gen**. Unknown real-world states are perceived by the sensor system as observation vectors o that are then processed and clustered by the **gen** function, leading to the conceptual states. The observation process is shown in Figure 8.8. The function that produces conceptual world states from observation vectors can be implemented as a neural network, so that new conceptual states can be incorporated into the implementation of **gen** by an unsupervised learning process.

8.4.2.1. Neural implementation of generalization algorithm. The problem of obtaining conceptual world states from sensor input in such a way as to preserve proximity information among the sensor vectors can be solved by a neural network that forms clusters in its input space, and produces a “good” representative of this

cluster as output. The network is trained in an unsupervised manner, so no other input but the sensor vectors is necessary.

The network operates in a way that is a combination of the Kohonen clustering algorithm (Kohonen, 1988) and the class creation and pruning methods incorporated in the fuzzyART and fuzzyARTMAP algorithms (Carpenter et al., 1991; Carpenter and Tan, 1993). The topology of the network consists of an input layer and an output layer, with full connection between these two layers. The input layer has dimensionality m , and the output layer grows and shrinks as new category neurons (each representing a conceptual state) are added and deleted.

For the training process (described below), the robotic agent is first operated in an *active mode*, i.e., sensor readings are taken with the robot in positions that can be chosen by the user to get desirable sensor readings. At each position, sonar readings are taken while the position of the manipulator is perturbed slightly; this is done to obtain a set of similar sensor vectors. This process is repeated for several positions of the manipulator. The sensor readings obtained in this active mode serve as the initial training data for the generalization network.

The *passive mode* of learning takes place when the robot performs a task in its environment. Again, sensor readings are taken and the network is trained, but in contrast to the active mode, it is not possible for the user to specify the position of the robot.

Just as in the Kohonen algorithm, the clustering algorithm presented here is based on measuring the similarity of an input vector with the category information that is stored for each category as the weights of the corresponding output neurons. The similarity measure between sensor vector o and weight w used in the algorithm is the Euclidean distance, i.e.,

$$d(o, w) = \sqrt{\sum_{i=1}^m (o_i^2 - w_i^2)} \quad (8.12)$$

When an input vector is presented to the network, all the output neurons calculate their distance (i.e., the distance of their weight vector to the input) in parallel. The neuron with the smallest distance wins.

At this point, we use an idea from the fuzzyART algorithm and check whether the winning neuron is close enough to be able to represent the input vector, or whether the input vector is so dissimilar from the winning neuron's weights (and thus from all the other categories as well) that it has to be placed in a new category. For this, we define a *similarity radius* r_s as the maximum distance that an observation vector can be from the winning neuron's weights to still be considered close enough to fall into that category.

8.4.2.2. Category neuron training. If the observation vector o is within one similarity radius of the winning neuron's weight \tilde{w} , i.e.,

$$d(o, \tilde{w}) \leq r_s \quad (8.13)$$

then the weight $\tilde{\mathbf{w}}$ is used to reflect the new entry in this category by moving in the direction of o :

$$\tilde{\mathbf{w}}^{\text{new}} = \tilde{\mathbf{w}}^{\text{old}} + \beta(o - \tilde{\mathbf{w}}^{\text{old}}) \quad (8.14)$$

where β is a scalar factor in $[0,1]$ that determines how far $\tilde{\mathbf{w}}^{\text{old}}$ is changed in the direction of o .

8.4.2.3. Category neuron creation. If the sensor vector is not within the similarity radius of the winning neuron, it is not similar enough to any of the current categories to be included in one of them, and a new category has to be created. In this algorithm, this corresponds to the creation of a new output neuron that is fully connected to the input layer, and whose weight vector is equal to the sensor vector. The network starts with one output neuron that is created upon presentation of the first input pattern. Output neurons are created when an observation vector is found not to be within one similarity radius of any of the output neurons. It is clear that the smaller the similarity radius, the more output neurons will be created, because the criterion for similarity is the stronger, the smaller is this radius.

8.4.2.4. Category neuron pruning. To prevent a proliferation of output neurons, we include a pruning step that cuts output neurons. The pruning can be done according to two different criteria:

- A neuron is cut when it has not been the winning neuron for a given period of time.
- A neuron is cut when it encodes too small a fraction of sensor vectors compared with the output neuron that encodes the most observation vectors.

The second method produces neurons with a longer lifetime, i.e., they are not pruned and recreated as often as in the first method. More precisely, we can cut a neuron when it represents a category with less than 10% of the number of observation vectors in the category that has the highest number of vectors.

The output associated with the clustering algorithm is *not* the output of the network, as these are only the similarity numbers for the various categories. Instead, the output of the algorithm is the weight vector of the winning output neuron. In other words, the algorithm output is the representative of the category that is most similar to the input presented to the network. If the observation vector is within the similarity radius of one of the output neurons, the output is the weight vector of this neuron; otherwise, it is the observation vector itself (which is also incorporated in the network as a new category neuron).

The network presented above produces conceptual states from observation vectors. These conceptual states are then used as inputs for the **do** function that models the effects of the agent's behavior on the conceptual state space. The network that implements this function is presented in the next section.

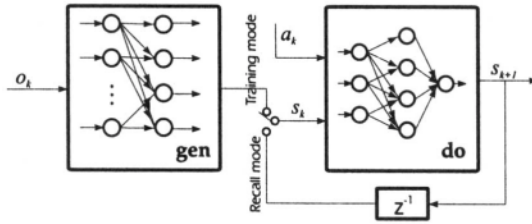


Figure 8.9. Structure of the network modeling the function **do**.

8.4.3. Learning of Agent Actions for Prediction and Coordination

The function **do** can be seen as the state transition function of a dynamical system with states in S and inputs in A . As such, it can be implemented by a multilayer feedforward network that learns and refines the mapping as the robot operates. The neural network runs in two different modes:

- The *training mode*, when input patterns (s_i, a_i) and desired outputs s_{i+1} are presented to the network, and the network is trained on this static mapping.
- The *recall mode*, when the conceptual state information is not obtained from the **gen** function, but from memory cells that store the previous state output of the **do** network and feed it back as input with the next action.

The structure of such a network, operating on state information from the **gen** function in the training mode and on state information from its own delayed output in the recall mode is shown in Figure 8.9. The multilayer feedforward network that implements the static function **do** can be trained with the backpropagation algorithm. The training process is slightly different from the standard case, as the desired network output s_i^d is a conceptual state that represents many observation vectors, and thus all actual network outputs s_i^a belonging to this same category should be treated as correct outputs. This is accomplished by presenting s_i^a as an input to the **gen** network (this is possible because of the same dimensionality of O and S). If $\text{gen}(s_i^a) = s_i^d$, i.e., if the actual and the desired outputs of the **do** network are in the same conceptual state, then no training is performed. Otherwise, the network is trained with the error being the difference between s_i^a and s_i^d .

8.4.3.1. Extensions of the learning algorithm. The above algorithm can be extended in a variety of ways to increase the accuracy of the modeling. Some possible extensions are:

- a. Changing the number of hidden neurons in the action effect modeling network.

- b. Eliminating old patterns when their inclusion in the training set makes it inconsistent.
- c. Extending the number of previous actions and conceptual states used as inputs for the action effect modeling network.

a. The current implementation uses a fixed number of hidden neurons in the **do** network. Extensions are made to increase the number of hidden neurons when an increase in the number of category neurons in the **gen** network (and thus the possible conceptual state inputs to **do**) decreases the performance of the **do** network. A possible approach is to use a second network with more hidden neurons and train it in parallel with the regular network, and when performance decreases in the regular network, use the second network to replace the regular network.

b. Another limitation of sensor-based systems is that the range of the sensors limits the perception range of the agent to a small area. With these limitations, it is possible that the training set for the **do** network contains different desired conceptual state outputs for the same pair of state and action. This makes training the network impossible, as **do** would then be a relation and not a function. One straightforward solution would be to eliminate the older pattern, as it reflects a situation that is not as relevant as the current situation.

c. Since eliminating patterns, as proposed in (b), is a very crude approach and much information is lost, we can use a different, more elaborate method. Instead of using only the current conceptual state s_i and the current action a_i to predict the next conceptual state s_{i+1} , a “history” of states and actions before s_i and a_i could be taken into account as well. The number of previous states and actions that should be taken into consideration cannot be determined beforehand and depend on the complexity of the robot environment. A possible solution is to use an overdetermined system (some large number of previous states and actions), and use a genetic algorithm to find those past states and actions that can contribute to determining the next state, while eliminating those that are not relevant. Such a genetic algorithm was developed in a different context (Jacak and Dreiseitl, 1996; Jacak and Dreiseitl, 1995).

Example 84.1. A two-dimensional 4-DOF manipulator was placed next to an obstacle in a simple scene. Sensors were mounted on both sides in the middle of each link with sensor readings scaled from 0 (contact) to 1 (time out at infinity). Since there are no objects on the left of the manipulator, we disregarded the sensors on that side since they always showed a distance of infinity, reducing the observation space dimensionality to 4.

The training of the two networks for the **gen** and **do** function was performed in two steps:

- The active mode of learning for the generalization network

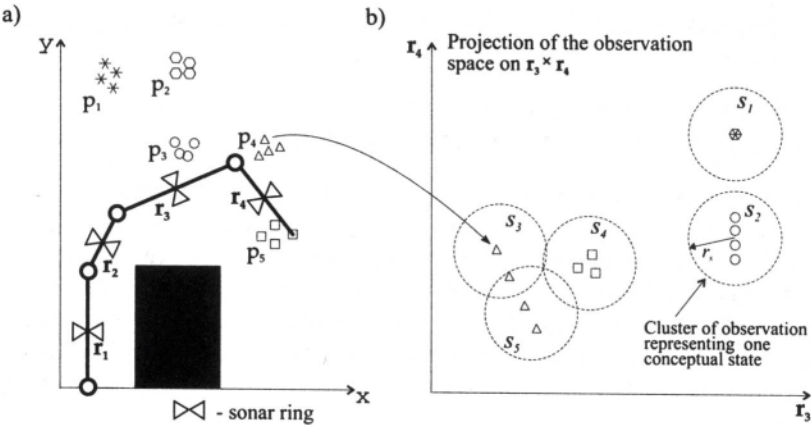


Figure 8.10. (a) Sensor readings taken during the active mode of learning. (b) Conceptual states of sensor readings on the last two links.

- A simulated normal operation of the robot when both the generalization and the state transition network were trained

During the active learning of the generalization network, we placed the manipulator into five positions p_1, \dots, p_5 that were far apart in the manipulator's configuration space. Each of these positions was changed slightly four times and sensor readings were taken (see Figure 8.10a). Using this training data of 20 observation vectors for the generalization network resulted in five conceptual state categories (see Figure 8.10b). However, these five categories were the result of one category being formed for two positions that were identical in sensor space (both

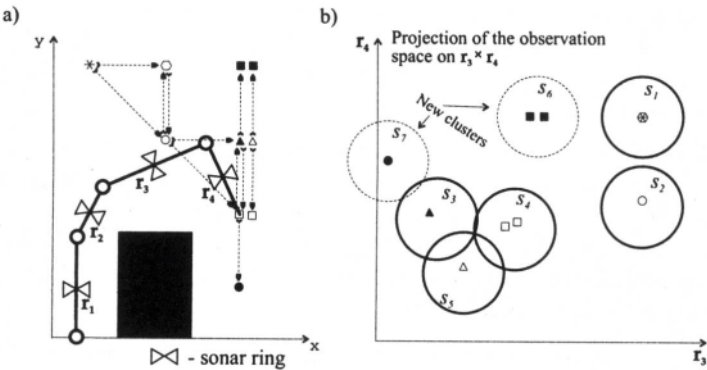


Figure 8.11. (a) Sensor readings taken during normal operation of the robot (passive learning). (b) Conceptual states of sensor readings on the last two links.

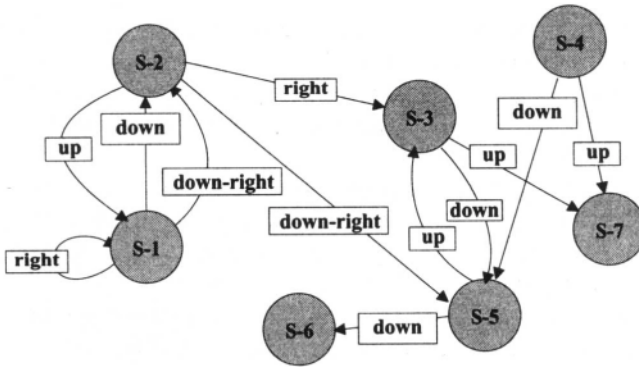


Figure 8.12. State transition graph of the function **do** obtained during simulation.

now represented by conceptual state s_1), and the sensor reading of one position being split into two distinct conceptual states (s_3 and s_5) because the small changes caused one sensor not to see the obstacle.

During the simulated normal operation of the robot, the actions (i.e., movements of the effector end) up, down, left, and right and appropriate combinations thereof were allowed. Actions were performed with the manipulator being in one of the five original positions used for training the generalization network (see Figure 8.11a). The states reached during operation of the robot are shown in Figure 8.12. During the work of the state transition network, two more conceptual states (s_6 and s_7 , see Figure 8.11b) were created by the generalization network and incorporated into the training of the state transition network. Both networks were able to learn the training patterns in fewer than 1000 presentations of the patterns.

8.4.4. Q-Learning-Based Intelligent Robot Action Planner

The inadequacy of classical planning algorithms for real-time applications involving lack of full knowledge about the surrounding world has motivated the use of learning approaches in order for rapidly finding timely behavior of the agent. The task of learning from scratch can be simplified by considering agents that memorize whole collections of actions and their effects over their entire lifetime. In such a lifelong learning scenario (Thrun, 1994), learning tasks are related in that they all play out in the same surrounding world, and they involve the same agent hardware.

To synthesize the action planner we use Q-learning method (Thrun, 1994). Q-Learning is a popular method for learning to select actions from delayed and sparse rewards. The goal of Q-learning is to learn the strategy for generating whole action sequences which maximize an externally given *rewardfunction*. The

reward may be delayed and/or sparse, i.e., reward is only received upon reaching the goal of the task or upon total failure.

Let \mathcal{O} be the set of all possible system percepts and $\mathbf{gen}: \mathcal{O} \rightarrow \mathcal{S}$ be the generalization function mapping the current observation o into the conceptual state of the surrounding world.

We now use the restrictive Markov assumption, i.e., we assume that at any discrete point of time, the agent can observe the complete conceptual state of the world. (See (Thrun, 1994; Thrun and Mitchell, 1993) for approaches to reinforcement learning in partially observable worlds.)

This assumption is motivated by the fact that an agent without external knowledge about the surrounding world is not enable to decide if an observation contains the complete state of the world. Roughly speaking, we assume that each observation represents the full knowledge about the world achieved by the agent.

Additionally, let \mathcal{A} be the action set of the agent. Based on the observation o and the adequate state $\mathbf{gen}(o)$, the agent picks an action $a \in \mathcal{A}$. As a result, the world state changes. The trainer also receives a scalar *reward value*, denoted by $r(s, a)$, which measures the action's performance. Such a reward can be exclusively received upon reaching a designated goal or upon total failure, respectively.

The Q-learning method finds an action strategy

$$\pi: \mathcal{S} \rightarrow \mathcal{A} \quad (8.15)$$

mapping from conceptual world states \mathcal{S} to actions \mathcal{A} which, when applied to action selection, maximizes the so-called *cumulative discounted future reward*,

$$R = \sum_k \gamma^k r(s_k, a_k) \quad (8.16)$$

where $\gamma \leq 1$ is the discount factor.

To find rapidly the best action in the current state s the key of Q-learning is to learn a *value function* for picking the actions. A value function

$$\mathbf{Q}: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R} \quad (8.17)$$

maps conceptual states perceived by the agent $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$ to scalar utility values.

In the ideal case $\mathbf{Q}(s, a)$ is, after learning, the maximum cumulative reward one can expected upon executing action a in state s . The function \mathbf{Q} schedules actions according to their reward. The larger the expect cumulative reward for applying action a in the current state s , the larger is its value $\mathbf{Q}(s, a)$.

After learning, the value function \mathbf{Q} generates optimal actions by picking the action which maximizes \mathbf{Q} for the current state s , i.e.,

$$\pi(s) = \operatorname{argmax}_{a' \in \mathcal{A}} \mathbf{Q}(s, a') \quad (8.18)$$

The values of $Q(s, a)$ have to be learned over the whole lifetime of the agent acting in the same surrounding world. The function Q is realized as a complex neural network which consists of networks approximating the functions

$$Q_a(s) = Q(s, a) \quad \text{for} \quad a \in A \quad (8.19)$$

Initially, all values $Q(s, a)$ are set to zero. During learning, values are incrementally updated, using the following standard recursive procedure.

Suppose the agent just executed a whole action sequence which, starting at some initial state s_0 , led to a final state s_F with reward $r(s_F, a_F)$. For all steps i within this episode, $Q(s_i, a_i)$ is updated through a mixture of the values of subsequent state action pairs up to the final state. This standard procedure has the following form (Thrun and Mitchell, 1993):

$$Q^{new}(s_i, a_i) = \begin{cases} +N & \text{if } a_i \text{ final action, agent reached goal} \\ -N & \text{if } a_i \text{ final action, agent failed} \\ \gamma[(1 - \lambda) \cdot \max_a Q(s_{i+1}, a) + \lambda \cdot Q^{new}(s_{i+1}, a_{i+1})] & \end{cases} \quad (8.20)$$

Such Q-learning learns individual strategies independently, ignoring opportunity for the transfer of knowledge across different tasks (Thrun and Mitchell, 1993). In order to transfer knowledge the planner needs to learn the predictive action model **plan**: $S \times A \rightarrow A$. The action model describes indirectly the effect of the agent actions. Such a function is *a priori* unknown and has to be learned based on the observed state transition over the whole lifetime of the agent. Once trained, the neural network model can be used to analyze observed episodes by additional explanation. The explanation makes it possible derive the slopes of the valuefunction Q . The slopes are used for generalizing the episodes. The slopes generalize training instances in sensor space and conceptual word state space, since they indicate how small changes will affect the target value function Q . They are extracted from the general action model, which is acquired and used over the entire lifetime of the agent. Hence, in the lifelong learning context such a neural network transfers knowledge.

Example 8.42. To illustrate the Q-learning-based action planning we present some experiment results. The experiment was performed with the Scara Adept One robot equipped with eight sonar sensors. The location of the sensors and the surrounding environment are presented in Figure 8.13.

During the active mode of learning the generalization system creates eight conceptual states. The sensor readings and the aggregation of these readings into conceptual states is shown in Figure 8.13. The partial function **do** (partial knowledge) is stored as the state transition table (Figure 8.14). Based on this knowledge the Q-method generates the following action plan for transition from initial state s_1 to final state s_8 :

Actions = (right, back, right, right, front, right, back)

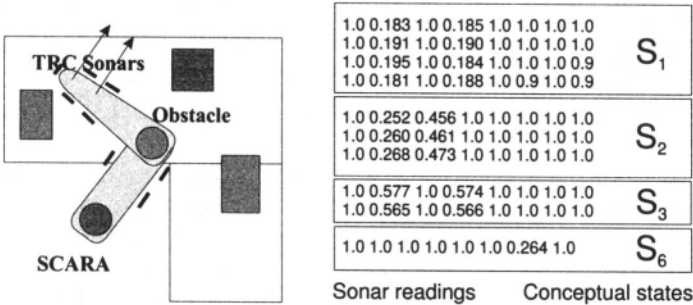


Figure 8.13. Adept One and its conceptual states.

with state trace

$$\text{Trace} = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$$

where “right, back, front, left” are the elementary actions of the robot.

During the execution of this plan there is an error between predicted states s_6 and s_7 and the observed states. This dynamic change in the environment requires the creation of new conceptual states and retraining of the state-transition neural network. The generalization function generates two new states and the Q-learning based function **plan** modifies the plan as follows:

$$\text{Trace} = (s_1, s_2, s_3, s_4, s_5, s_9, s_{10}, s_8)$$

The error occurring during tracing is presented in Figure 8.15.

8.4.5. The Agent's Action Interpreter

While the robot is tracking a path in an unknown environment, some extraneous object may enter the work space. These objects in the real world are detected with the help of sensors mounted on the links of the manipulator. As the perception system of such a robotic agent we install ultrasonic sensors and a sensitive skin

do	s1	s2	s3	s4	s5	s6	s7	s8
right	s2	-	s4	s5	-	s7	-	-
left	-	s1	-	s3	s4	-	s6	-
back	-	s3	-	-	-	s5	s8	-
front	-	-	s2	-	s6	-	-	s7

Figure 8.14. State transition table.

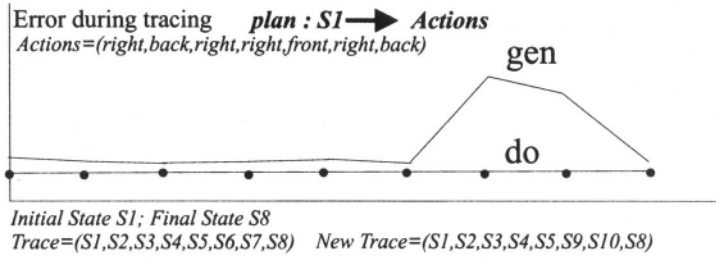


Figure 8.15. Tracing error.

on each manipulator link and then use a neural network to estimate the proximity of objects to the link in question. The vector of sensor signal values together with the distance to the goal position establishes the points in the sensor space, which are used to construct the conceptual state space of the real world.

Let δ be the finite length of the displacement vector of the effector end in Cartesian space. Based on δ we introduce the following set of standard displacement vectors of the effector end:

$$\begin{aligned} \delta p &= \{(+\delta x, 0, 0), (-\delta x, 0, 0), (0, +\delta y, 0), (0, -\delta y, 0), (0, 0, +\delta z), (0, 0, -\delta z)\} \\ &= \{\text{Forward, Backward, Left, Right, Up, Down}\} \end{aligned} \quad (8.21)$$

Additionally, we decide on the form of the geometric shape of the robot's manipulator, which is especially important if the number of degrees of freedom is large. To describe the topological form of the manipulator shape, especially for the last arm links, we use the following set of terms:

$$\text{Wrist Shape} = \{\text{straighten out, arch up, arch down}\} \quad (8.22)$$

Such a set determines the shape of the wrist in the case of a planar manipulator. For a nonplanar wrist the Wrist Shape set has to be extended by additional terms, namely "arch left" and "arch right." Based on these two sets we introduce the elementary action set A :

$$A = \delta p \times \text{Wrist Shape} \quad (8.23)$$

where the pair $(\delta, \text{shape}) \in A$ denotes the displacement of the effector-end action with simultaneous changes of the wrist shape. For the above action set is necessary to find the model of robot kinematics which makes it possible to calculate the resulting configuration. Let Q_{Agent} denote the joint space of the robot. Hence the kinematics model can be expressed as a discrete dynamical system in the form

$$\text{inter} : Q_{\text{Agent}} \times A \rightarrow Q_{\text{Agent}} \quad (8.24)$$

$$q(k+1) = \text{inter}(q(k)a(k)) \quad (8.25)$$

where $q(k)$ is the intern manipulator state at the time k , $a(k)$ is the applied action, and $q(k+1)$ is the resulting state. The main problem of action interpretation for this model of the kinematics is how to find the next state function **inter**. Recall that the function is to be found by solving the underdetermined kinematics equations of the manipulator. In order to choose one configuration from among many possibilities, we use the modified method proposed in (Jacak, 1989a; Jacak, 1991). Let $t_i(q)$ for $i = 1, \dots, n$ denote the i th component of the direct kinematics, i.e., $t_i(q) = P_i$ describes the position of the i th joint in Cartesian space. Based on the displacement vector δ we can define the goal point of the motion as $P_F = [t_n(q(k)) + \delta]$.

In order to choose one configuration from among many possibilities, we decompose the whole manipulator into two submanipulators. We introduce two parameter calculated on the basis of two sets of joint numbers. Let $U = \{1, \dots, n-2\}$ denote an arbitrary set of initial joints of the submanipulator (i.e., subsequence of the kinematic chain from 1 to $n-2$), and let $G = \{k, \dots, n\}$ stand for the set of terminal joints of the submanipulator. From these sets we choose joints which decompose the manipulator into active and passive parts. The selection strategy for changing the configuration acts as follows:

$$(l, g) = \max_{i \in U} (\min_{j \in G} \{C_{j,\min}^i \leq c_j^i \leq C_{j,\max}^i \text{ and } i < j-1\}) \quad (8.26)$$

where $C_{j,\min}^i$ ($C_{j,\max}^i$) denote the relatives maximum (minimum) reach between joints i and j [i.e., the maximum (minimum) size of the kinematic chain connecting the joints i, j], and $c_j^i = \|t_i(q(k)) - [t_j(q(k)) + \delta]\|$. These two parameters determine the active and passive parts of the manipulator. The new position for each joint in Cartesian space is determined as

$$P_i^{\text{new}} = \begin{cases} t_i(q(k)) + \delta & \text{for } i = g, \dots, n \\ t_i(q(k)) & \text{for } i = 1, \dots, l \\ \text{\textit{i}th Joint Position Procedure} & \text{for } i = l+1, \dots, g-1 \end{cases} \quad (8.27)$$

The Joint Position Procedure determines the new position of the joint based on the transformation of the inverse kinematic problem into an optimization problem (Jacak, 1995; Jacak et al., 1995c; Jacak et al., 1995d). For a planar manipulator, the procedure can be reduced to a simple algebraic algorithm (Jacak, 1989a). The activation of the manipulator part depends on the set G . We introduce two strategies:

Active Terminal Joints — Reach Motion Strategy Let $G = \{n\}$. Then due to the above equation the number of terminal joints k of the configuration change is equal to n and the number of initial joints l of the configuration change is determined as

$$l = \max\{i \leq n-2 \mid C_{n,\min}^i \leq c_j^i \leq C_{n,\max}^i\} \quad (8.28)$$

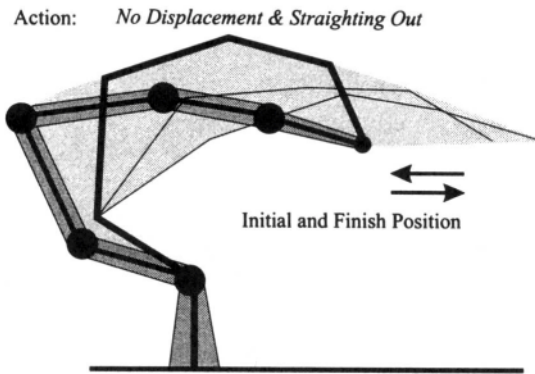


Figure 8.16. Action interpretation.

Such a choice of l ensures that to achieve a desirable shift of the end effector, a minimum number of links will be set in motion, counting from the end of the manipulator. The configuration of the submanipulator from joint 1 to l remains unchanged.

Active Initial Joints — Gross Motion Strategy Let $G = \{3, \dots, n\}$. Thus, the required shift of the effector end is realized by moving a minimum number of joints counting from the base of the manipulator. The submanipulator defined by numbers g and n preserves its geometric shape but the positions of its joints are shifted by a vector δ . A sequence of these configurations corresponds to the gross motion aimed at placing the effector end in the neighborhood of the terminal point.

Observe that for joints l and g the new configuration has already preserved the geometric form of submanipulators from 1 to l and from g to n , i.e., the distances and angles in those segments of the kinematic chain are kept fixed.

By appropriate choice of the set G we activate only one segment of the kinematic chain while keeping the second unchanged in shape. By composing successive configurations one can produce a definite geometric shape of the kinematic chain according to the second element of the action pair (δ, shape) . For example, for the action $a = (\delta, \text{straighten out})$ the new configuration $q^{\text{new}} = \text{inter}(q, a) \in Q_{\text{Agent}}$ can be produced from an actual configuration by a sequence of transition of the following types:

- reach motion strategy for additional displacement $(\delta x, 0, 0)$
- gross motion strategy for negative displacement $(-\delta x, 0, 0)$
- gross motion strategy for desirable displacement δ

Similarly we can define the transition function **inter** for the rest of the actions.

As a result of using such an interpreter we obtain the goal configuration of the movement step in order to realize the given action. This configuration is the input for the reactive motion controller (executor). An example of an interpretation of the robot action is presented in Figure 8.16.

While the robotic agent is tracking a path calculated by the action interpreter, some extraneous dynamic object (another robot) may enter the work space. These dynamic obstacles are detected with the help of sensors mounted on the links of the manipulator. To avoid possible collisions we use the intelligent and reactive robotic agent presented in Section 7.4.

CHAPTER 9

The Organization Level of a Robotic System

The organizer accepts and interprets related feedback from lower levels, defines the strategy of task realization and processes large amounts of information with little or no precision. Its functions are reasoning, decision making, learning feedback, and long term memory exchange.

9.1. The Task of the Robotic System Organizer

The organizer determines the best strategy for realizing the current set of technological tasks (*TASKS*) *quasiordered* by the task–priority relation

$$\pi \subset TASKS \times TASKS \quad (9.1)$$

If for two tasks $Z_1, Z_2 \in TASKS$ we have $Z_1 \pi Z_2$, then the task Z_1 has a higher priority than the task Z_2 .

Each task $Z \in TASKS$ is characterized by a set of technological operation O_Z [Equation (4.1)] and a number of parts N_Z to be processed. Moreover, for some tasks the critical finishing time T_Z^{finish} is given.

In a given state of the computer assisted robotic cell there are many parts waiting for processing (recognized on the execution level and the coordination level) which cannot be processed simultaneously. Based on feedback information from the execution and coordination levels, the organizer creates a strategy u which chooses the technological operation to be performed by determining the priority of the operations waiting for execution.

The inputs (feedback information from the execution and coordination levels) of the CARC organizer are represented by a vector of statistics V calculated from the event trace and the current state of the CARC. The inputs to the organization level are defined as follows (Bedworth et al., 1991; Black, 1988; King, 1980; Wang and Li, 1991):

- ω , the work-in-process factor: the number of jobs (parts) being currently processed in the cell
- τ_w , the mean waiting time of a job during its stay in the cell
- τ_p , the mean production time of a job during its stay in the cell
- C , the mean job cost
- UT , the device utilization vector
- W , the productivity: the number of finished jobs per unit of time

The organizer can be chosen among of such the strategies of operation scheduling as (Prasad, 1989; Bedworth et al., 1991; King, 1980; Wang and Li, 1991):

- just-in-time (JIT)
- maximum waiting time (MW): the operation with longest waiting time is given preference
- minimum setup time (MS): the operation requiring the shortest time to change the processing tool is given preference
- push strategy (PUSH)
- first free buffer (BUF)
- minimum transfer cost (MT): the operation with minimal transfer cost is given preference

These strategies are described in the next section. The set of possible strategies is denoted by U . To realize the organization process we develop a general fuzzy organizer of the robotic workcell.

9.2. Fuzzy Reasoning System at the Organization Level

We apply a fuzzy rules-based decision system to create the organization level of a CARC (Lee, 1990; Valavanis and Stellakis, 1991; Zadeh, 1973; Efstathiou, 1987; Sugeno, 1985). To solve the organization problem we introduce three families of fuzzy sets.

9.2.1. Tasks, Scheduling Strategy, and Input Fuzzyfication

9.2.1.1. *Technological task fuzzyfication.* The order of realization and the importance of tasks at a particular stage of their execution can be described by the time-dependent fuzzy set of tasks

$$\mathcal{T} = \{\mu_T(t, Z) / (t, Z) | (t, Z) \in T_D \times TASKS\} \quad (9.2)$$

where T_D denotes the set of time instants (check points) at which all the tasks from the set $TASKS$ must be realized.

The membership function μ_T is called the task priority function at the moment t_i [$\mu_T(t_i, Z) = \mu_T^i(Z)$] because its value at t_i can be thought of as the preference of the task Z at the realization stage i .

At the initial stage 0 when the set of tasks starts to be realized the values of function μ_T are assumed *a priori* to meet the following condition:

$$Z_1 \pi Z_2 \rightarrow \mu_T^0(Z_1) > \mu_T^0(Z_2) \quad (9.3)$$

It is easy to construct an algorithm which can generate for a finite set $TASKS$ and a given ordering relation π the values of μ satisfying Equation (9.1) (Jacak, 1985).

The considered set of tasks is a dynamic set. The dynamics are expressed by the variation of the set of tasks to be realized at each stage. The dynamic priority of the tasks is represented by the values of the membership function $\mu_T^i(Z)$. The modification of this function at the i th stage depends on the degree to which the task was realized at the previous stage.

Let $\rho_Z(i)$ denote the distance from the current realization state of the task Z to its finished state:

$$\rho_Z(i) = \min \left\{ \frac{k_1(N_Z - n_Z(i))}{N_Z}, \frac{k_2(T_Z^{\text{finish}} - t_i)}{T_Z^{\text{finish}} - T_Z^{\text{start}}} \right\}, \quad k_1, k_2 \gg 1 \quad (9.4)$$

where $n_Z(i)$ is the number of realized jobs from task Z in moment t_i . Based on such the definition we can modify a priority function at the stage $i+1$ as follows (Jacak, 1985):

$$(\forall Z \in TASKS) \left(\mu_T^{i+1}(Z) = \mu_T^i(Z) + \frac{1 - \mu_T^i(Z)}{1 + \rho_Z(i)} \right) \quad (9.5)$$

Due to such a modification, preference is given to tasks which are nearly finished. Additionally, the value of the priority function at each stage of task realization can be modified by a human operator. By introducing such variability into the function μ_T the realization of every task becomes a process of adaptation to the stage of decision making and to changing conditions.

9.2.1.2. Fuzzyfication of scheduling strategies. The second family of fuzzy sets is related to the scheduling strategies. Let U be a set of scheduling strategies. Each strategy $u \in U$ determines the priorities for a given set of machining operations. Such preferences can be represented in terms of the values of a membership function of a fuzzy set related to a given strategy. Then each scheduling strategy, such as JIT, MW, MS, PUSH, BUF, MT, FIFO, or LIFO, is transformed into a fuzzy set over the machining operations for a given technological task.

Let O_Z denote the set of operations for the machining task $Z \in TASKS$. For every strategy $u \in U$ and every task Z we introduce the fuzzy set

$$O_Z^u = \{\mu_u(o)/o | o \in O_Z\} = \sum_{o \in O_Z} \mu_u(o)/o \quad (9.6)$$

The membership function $\mu_u(o)$ represents the priority of realization of the o operation with respect to the u strategy.

The JIT and PUSH strategies are independent of the current state of the workcell and depend only on the sequence of operations in the production route \wp .

JIT strategy If the production route $\wp_Z = (o_1, \dots, o_L)$, then the JIT strategy can be represented by a fuzzy set with a linear membership function:

$$\mu_{JIT}(o_i) = \frac{i}{L} \quad (9.7)$$

PUSH strategy The PUSH strategy is transformed into a fuzzy set with membership function as follows:

$$\mu_{PUSH}(o_i) = \frac{L - i + 1}{L} \quad (9.8)$$

The fuzzy representations for the remaining strategies depend on the current state of the workcell. For these strategies we define the membership function of the related fuzzy set, which depends on the current state $s(t)$ of the workcell. Based on the general state of the workcell $s(t)$, we create a set of parts (jobs) for a task Z in the cell and waiting for processing, i.e., the set $Job_Z(t) = \{job_k(o_i) | \text{part } k \text{ wait for operation } l\}$. It is possible that more than one part (job) wait for the same machining operation.

MS strategy The minimum setup time strategy (MS) has the following fuzzy representation:

$$\mu_{MS}(o_i) = \begin{cases} 0 & \text{iff } o_i \notin Job_Z(t) \\ \frac{t_{\text{setup}}^{\max} - t_{\text{setup}}(o_i, s(t))}{t_{\text{setup}}^{\max} - t_{\text{setup}}^{\min}} & \text{iff } o_i \in Job_Z(t) \end{cases} \quad (9.9)$$

where $t_{\text{setup}}^{\max} = \max\{t_{\text{setup}}(o_i, s(t)) | o_i \in Job_Z(t)\}$ and t_{setup} denotes the time needed to change the workstation's tool.

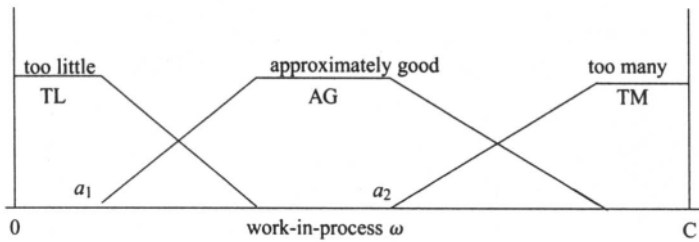


Figure 9.1. The primary fuzzy sets of the work-in-process factor.

The maximum waiting time strategy can be modeled in an analogous way.

The combined scheduling strategies determine the output of the organization level of the CARC.

9.2.1.3. Organizer input fuzzyfication. The third family of fuzzy sets is related to the organizer inputs. The inputs of the CARC organizer represent *the global state of the workcell statistics*. These inputs can be obtained from the event trace analysis or from aggregation of the current state. The input values of the cell organizer do not form a fuzzy set, but are crisp. Therefore they first have to be fuzzyfied. In general, linguistic variables will be associated with a term set each term in the of which is defined on the same universe of discourse. A fuzzyfication or fuzzy partition determines the number of primary fuzzy sets.

If we consider the work-in-process factor ω (the current number of parts being in the cell), then possible primary fuzzy sets to describe the number of jobs being processed are e.g., *too many* (TM), *approximately good* (AG), and *too little* (TL). For each term TM, AG, and TL a certain bell-shaped membership function is used to cover the whole domain (universe). In principle, any shape is possible, but the simple form has many advantages because it is easy to represent and it lowers the computational complexity of the system. To cover the whole domain of the work-in-process factor ω , $[0, C]$, where C is the capacity of whole workcell, we apply a trapezoid-shaped function to define the membership function as shown in Figure 9.1. Because the universe is nonnormalized, the function could be asymmetrical and unevenly distributed in the domain. The membership function of a primary fuzzy set depends on a vector of parameters \vec{a} which determine its shape (see Figure 9.1).

This same method can be used to define the terms and the primary fuzzy sets for the other inputs of the organizer. For the mean waiting time of the job τ_w , we convert input data into three linguistic values, namely *short* (S), *quite good* (G), and *too long* (L), which are viewed as labels of primary fuzzy sets. To express the membership function of the primary fuzzy set, we apply functional definitions (Lee, 1990; Zadeh, 1973) in a trapezoid-shaped form analogously as for the previous input. This can be done for each input of the workcell organizer,

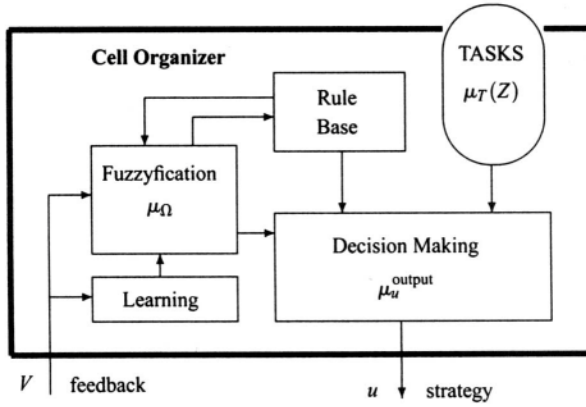


Figure 9.2. Structure of the robotic workcell organizer.

i.e.,

$$(\forall v \in V) (v \mapsto \sum \mu_{\Omega}(v)/v) \quad (9.10)$$

9.3. The Rule Base and Decision Making

To synthesize the workcell organizer we apply the structure of the decision system shown in Figure 9.2.

9.3.1. The Rule Base

In general, a fuzzy decision rule is a fuzzy relation which is expressed as a fuzzy implication. The choice of fuzzy implication reflects not only the intuitive criteria for implication, but also the effect of the connective *also*. In general, fuzzy implication functions can be classified into three main categories: fuzzy conjunction, fuzzy disjunction, and fuzzy implication (Zadeh, 1973; Baldwin and Guild, 1980). Just as in classical fuzzy control, we use only *if-then* rules. The rule base consists of all *if-then* rules. The rule base can be derived from expert knowledge, or can be extended by learning (Wang and Mendel, 1992). It is composed of a set of fuzzy rules with multiple workcell state variables (input statistics) and a single decision variable (output strategy), represented as

$$\text{if } (x \text{ is } A_1 \text{ and } \dots y \text{ is } B_n), \text{ then } z \text{ is } D_k \quad (9.11)$$

In this chapter let us for simplicity take a simple rule base. We consider fuzzy decision rules in the case of two inputs (work-in-process factor ω and mean waiting

time τ_w) and a single output u of the fuzzy organizer. Examples of fuzzy rules using only four scheduling strategies (just-in-time, JIT; maximum waiting time, MW; minimum setup time, MS; and push strategy, PUSH), are as follows:

- R_1: if ω is TL and τ_w is S, then u is PUSH
- R_2: if ω is TL and τ_w is G, then u is PUSH
- R_3: if ω is TL and τ_w is L, then u is MW
- R_4: if ω is AG and τ_w is S, then u is MS
- R_5: if ω is AG and τ_w is G, then u is MS
- R_6: if ω is AG and τ_w is L, then u is MW
- R_7: if ω is TM and τ_w is S, then u is JIT
- R_8: if ω is TM and τ_w is G, then u is JIT
- R_9: if ω is TM and τ_w is L, then u is JIT

The premises are compared with the input values of the cell organizer such that it is decided which rules can be used and which rules cannot be used, and in what way they can be used. As result it is determined which rules can be fired, together with the strength of each firing operation. This strength depends on how much the input value and the premise of the rule correspond to each other.

9.3.2. Decision-Making System

The rules can be fired according to the fuzzyfication interface, each with its particular strength. This strength determines the amount of influence the conclusion of a particular rule has on the general conclusion of the system. The inference mechanisms employed in the organizer's decision-making system are similar to those used in a typical expert system. In our decision system we reduce the inference mechanisms to one-level forward data-driven inference. In other words we do not employ chaining inference mechanisms.

The general consequence u is deduced from the compositional rule of inference by employing the definitions of a fuzzy implication function and the connective *also*. To generate the conclusion of a particular rule (R_{*i*}), we use the Larsen product operator (Baldwin and Guild, 1980) expressed as the composition

$$A \rightarrow B = \sum \mu_A(x) \mu_B(y) / (x, y) \quad (9.12)$$

In an on-line process, the global states of the workcell play an essential role in decision actions. The inputs are crisp, $\omega(t) = \omega^o$ and $\tau_w(t) = \tau_w^o$. In general, a crisp value may be treated as a fuzzy singleton. Then the firing strength ε_i of each rule R_{*i*} may be expressed by the intersection operator as follows:

$$\varepsilon_i = \mu_\Omega(\omega^o) \wedge \mu_\tau(\tau_w^o) = \min\{\mu_\Omega(\omega^o), \mu_\tau(\tau_w^o)\} \quad (9.13)$$

Using the Larsen product operation rule as a fuzzy implication function we determine the i th fuzzy decision (for rule R_i) as:

$$R_i: \quad \mu_{ui}^*(o) = \varepsilon_i \cdot \mu_{ui}(o) \quad (9.14)$$

To create the general conclusion of the organizer's decision system, the definition of the connective "also" in the form of triangular co-norms is used (Tilli, 1992; Lee, 1990). Consequently, the membership function μ_u^* of the inferred consequence is pointwise given by

$$\mu_u^*(o) = (\dots(\mu_{u1}^*(o) \oslash \mu_{u2}^*(o)) \oslash \dots \mu_{u9}^*(o)) \quad (9.15)$$

where the symbol \oslash can denote: \vee , union; $\hat{+}$, algebraic sum; \oplus , bounded sum; \uplus , disjoint sum; or \sqcup , drastic sum (Dubois and Prade, 1985). In the given fuzzy organizer we apply the algebraic sum in the form $x \hat{+} y = x + y - xy$.

The general conclusion of the organizer's decision system is computed for each technological task Z . In other words, the fuzzy reasoning system generates the general conclusion in the form of the fuzzy strategy $\mu_u^{Z*}(o)$ for each $Z \in TASKS$. Additionally, the fuzzy priority of tasks $\mu_T^i(Z)$ is used to compose the output of the decision system. The decision system create a new fuzzy set over the set of operations for each task using the Larsen product operator

$$(\forall Z \in TASKS)(\mu_Z^{\text{output}}(o) = \mu_T^i(Z) \cdot \mu_u^{Z*}(o)) \quad (9.16)$$

The membership function μ_Z^{output} represents the preference of operations with respect to the current feedback information from the workcell. This new fuzzy set is transferred into a new scheduling strategy at the execution and coordination levels,

$$u(v) = \left(\mathcal{O}_Z^u = \sum \mu_Z^{\text{output}}(o) / o | Z \in TASKS \right) \quad (9.17)$$

Recall that the execution and coordination levels use directly the values of membership functions to determine the priority of the waiting operations. Hence the defuzzification process can be omitted.

After the selection of operations to be realized on the execution level and their realization new feedback information is communicated from the coordination level to the organization level. This information is used to modify the parameters of the input's primary fuzzy sets.

Consider, for example, the organization level of a robotic workcell which realize L technological operations. We analyze the fuzzy decision system having the two inputs ω (work-in-process factor) and τ_w (mean waiting time) and a single output u . Figure 9.3 illustrates the fuzzy decision-making system with the above "if-then" rules. The activity of rules Rule_8 and Rule_9 as well as the general conclusion of the organizer are presented.

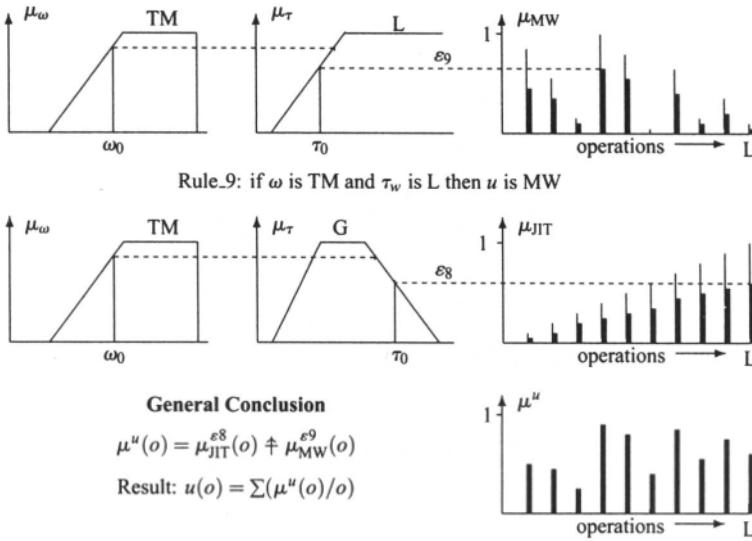


Figure 9.3. Example of fuzzy decision making.

9.3.2.1. *Learning.* After the selection of operations to be realized on the execution level and their realization the new feedback information is communicated from the coordination level to the organization level. This information can be used to modify the input's membership function parameters. A general stochastic approximation learning algorithm can be used, i.e.,

$$\alpha(k+1) = \alpha(k) + b(k) \cdot \xi(\text{grad } V(k))$$

where α is the vector of the input's membership function parameters, b is the updating rate matrix, and $\xi(\text{grad}(V))$ is the stochastic gradient of the statistics V .

Example 9.3.1. We consider now the organization level of a robotic workcell which realizes three technological tasks.

Simulation scenario. The results are based on the simulation of a cell with five robots, ten machines, two input conveyers, and one output conveyor. Its layout is given in Figure 9.4.

We consider three tasks with loops and shared machines for this cell. The logical structure of the routes is given in Figure 9.5. The following simulation runs use a single input parameter WIP (work-in-process factor). We examine and control the number of parts that are within the cell during the run of the simulator. Figure 9.6 shows the number of parts on time using the strategies JIT and PUSH

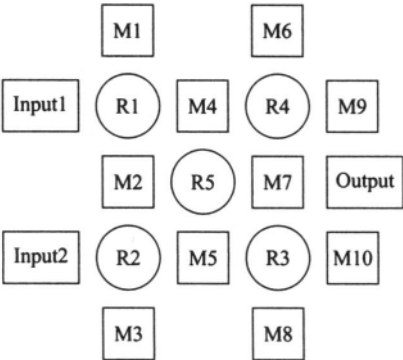


Figure 9.4. Cell layout of the simulation scenario.

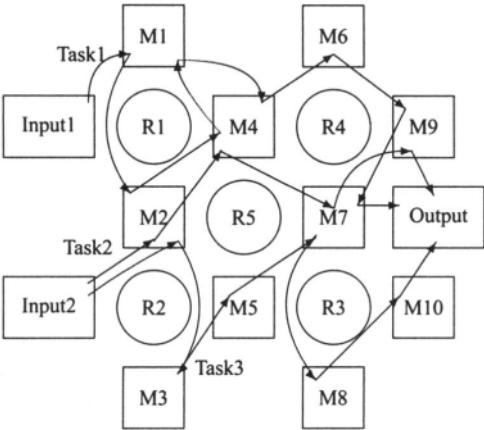


Figure 9.5. Routes of the simulation scenario.

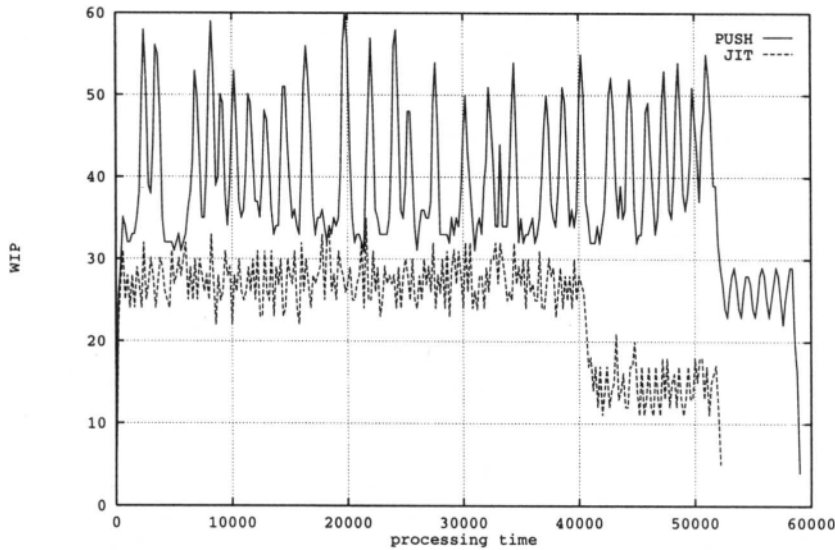


Figure 9.6. Bandwidth of the work-in-process factor WIP.

as proposed in Section 9.2. Note that the PUSH strategy takes longer to run than the JIT strategy. The lower WIP value at the end is caused by the finishing of one or two of the tasks. Here WIP can be regulated between 25 and 40.

Gain. We regulated WIP with several different versions of the linguistic variables within the range 20–40. On every run three linguistic variables (*too low*, *average good*, and *too much*) were defined with the same proportional relationship to one another. Figure 9.7 shows an example, where ω_d denotes the desired WIP factor. If WIP is *too low*, we use the strategy PUSH. This raises the number of parts in the cell. On the other hand, if WIP is *too much* we choose strategy JIT, where parts are pulled out of the cell. For values of WIP that are *average good* we

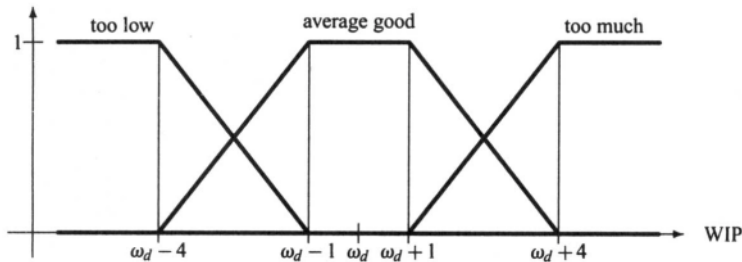


Figure 9.7. Example of linguistic variables for WIP.

Table 9.1. Processing Time for Different Values of the WIP Factors

WIP (parts)	Processing time (time units)	Comparison with ONE (%)
20	51,301	-5.0
22	50,094	-7.2
24	50,077	-7.2
25	51,332	-4.9
26	50,458	-6.5
27	50,181	-7.0
28	49,713	-7.9
29	50,368	-6.7
30	51,235	-5.1
32	52,268	-3.2
34	52,189	-3.3
36	52,645	-2.5
38	54,979	1.8
40	55,690	3.2

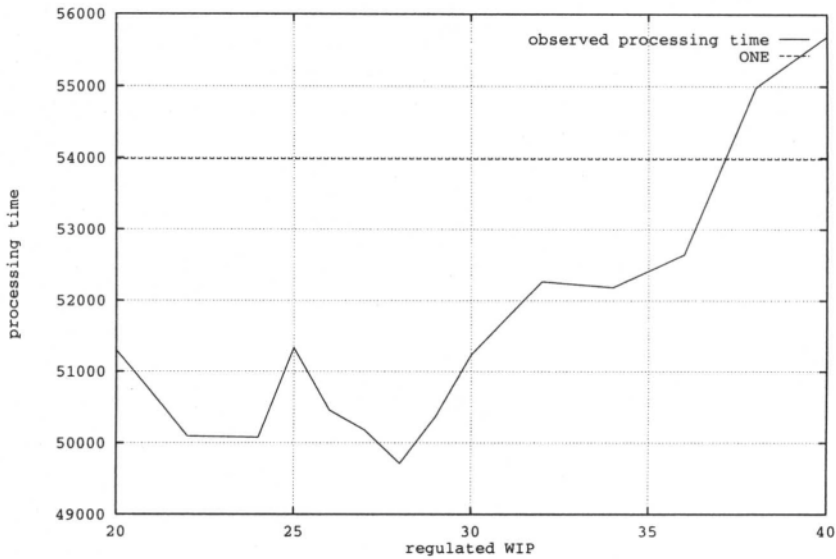


Figure 9.8. Processing time for different values of the WIP factor.

select the neutral strategy ONE (see Chapter 10). Table 9.1 shows the selected ω_d with the total processing time of all tasks and the gain in relation to the strategy ONE. The processing time for ONE was 53,984 time units. It is equal to the behavior without different priorities. Here ω_d goes from 20 to 40 which covers the whole controllable range. The same data are presented in Figure 9.8. As can be seen in this figure, there are several minima in the processing time. The shortest processing time is obtained at a WIP factor of approximately 28 parts. The total gain compared with ONE is 7.9%.

This page intentionally left blank

Real-Time Monitoring

The event-based workcell controller is an integral part of the CARC. The external events generated sequentially by the cell controller and robots activate the workcell's devices and coordinate the transfer actions.

The discrete-event model of the workcell generates a sequence of future events of the virtual cell in a given time window, called a *trace*. A trace of the behavior of a device d is a finite sequence of events (state changes) in which the process has been engaged up to some moment in time.

A trace in a time window is denoted as a sequence of pairs (state, time) (Zeigler, 1984; Jacak and Rozenblit, 1994; Jacak and Rozenblit, 1993), i.e.,

$$\text{tr}_{[t_0, t_0+T]} = \langle (s_1, t_1), (s_2, t_2), (s_3, t_3), \dots, (s_n, t_n) \rangle$$

where $t_1 \geq t_0 \wedge t_n \leq t_0 + T$.

The events from a trace are compared with the current states of the real cell and are used to predict motion commands for the robots and to monitor the process flow. The simulation model is modified at any time the states of the real cell change, and current real states are introduced into the model.

10.1. Tracing the Active State of Robotic Systems

Let $s(t) \in S_V(d)$ be the active state of device d and $t_a(s) = \tau$ and let

$$\bar{x}_n = \langle (e_1, t_1), (e_2, t_2), \dots, (e_n, t_n) \rangle$$

be a sequence of external virtual events in the time interval $[t_0, t_0 + \tau]$, where t_0 represents the moment in time when the device has changed to state s ,

$$t_0 \leq t_1 \leq t_2 \leq t_3 \leq \dots t_n \leq t_0 + \tau$$

Fact 10.1.1. *It is easy to prove that a trace in the interval $[t_0, t_0 + \tau]$ has the following form:*

$$\text{tr}_{[t_0, t_0+\tau]} = \langle v_0, v_1, v_2, v_3, \dots, v_n, v_{n+1} \rangle$$

and

$$\begin{aligned}
 v_0 &= (s, t_0) \\
 v_i &= ((\delta_{ext})^i((s, t_0), e_1, e_2, \dots, e_i), t_i) \quad \text{for } i = 1, \dots, n \\
 v_{n+1} &= (\delta_{int}((\delta_{ext})^n((s, t_0), \bar{x}_n)), t_n + t_a((\delta_{ext})^n((s, t_0), \bar{x}_n))) \\
 &= (\delta_{int}((\delta_{ext})^n((s, t_0), \bar{x}_n)), t_0 + \tau)
 \end{aligned}$$

where

$$(\delta_{ext})^i((s, t_0), e_1, e_2, \dots, e_i) = \delta_{ext}(((\delta_{ext})^{i-1}((s, t_0), e_1, e_2, \dots, e_{i-1}), t_i), e_i)$$

□

□

Moreover, the following facts can be proved:

Fact 10.1.2. *If $\bar{x}_n = \langle \emptyset \rangle$ then*

$$\text{tr}_{[t_0, t_0 + \tau]} = \langle v_0, v_1 \rangle = \langle (s, t_0), (\delta_{int}(s), t_0 + \tau) \rangle$$

and if

$$\bar{x}_n \neq \langle \emptyset \rangle$$

then

$$\text{crd}^1 \delta_{int}(s) = \text{crd}^1 \delta_{int}((\delta_{ext})^n((s, t_0), \bar{x}_n))$$

□

□

10.2. Monitoring and Prediagnosis

Let $\text{tr}_T(d)$ be the trace of virtual events of device d in the time interval $T = [t_o, t_o + T]$, where t_o is the moment of the last updating:

$$\text{tr}_T(d) = \langle (s_o, t_o), (s_1, t_1), \dots, (s_k, t_k) \rangle$$

where $t_k \leq t_o + T$ and s_i is the virtual state of the DEVS model of device d .

The monitoring process shown in Figure 10.1 is performed as follows:

Let $sr(t)$ be the current real state of device d , modeled by a passive Dev_d^p discrete system, and let $t \geq t_o$. Then, the monitoring algorithm has the following form:

If

$$q(t) \neq s_j \quad \text{for } t \in [t_j - \tau_0, t_j + \tau_0]$$

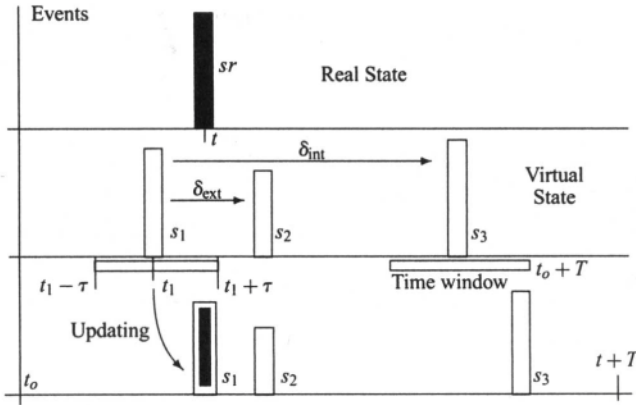


Figure 10.1. Monitoring of event trace.

then

call **diagnosis**

else

call **updating**

10.2.1. Updating

Let $sr_d(t)$ be the current real state of device d at time t , registered by the passive Dev_d^P automaton, and let $s_d(t') \in S_V(d)$ be a virtual state of the simulator of device d at time t' .

If $sr_d(t) = s_d(t')$ and $t' \neq t$, with $t \in [t' - \tau_0, t' + \tau_0]$, where τ_0 is the tolerance time window, then synchronization is performed between the real and the virtual cell.

An easy method for updating is to synchronize every device and robot of the workcell. The device d generates external signals for the updating module of the workcell controller, and the controller performs the so-called *global updating* process, namely

$$(\forall x \in D \cup M \cup R)(up(x) = 1 \Rightarrow \lambda_x(s_x, sr_x, 1) = sr_x \in S_V(x))$$

Such a global updating process need not be necessary for each device, and only parts of some devices may need updating. To specify such a *local updating* process we introduce a causality relation between events.

Let the predicate $\text{Occ}(e)$ denote that the event e has occurred. The causality relation is defined as follows:

$$e \rightsquigarrow e' \Leftrightarrow (\neg \text{Occ}(e) \Rightarrow \neg \text{Occ}(e'))$$

and expresses that event e is one of the causes of event e' .

The causality relation is reflexive, asymmetric and transitive.

Let $\text{Trace}_{[t',t]} = \bigcup \{\text{tr}_{[t',t]}(x) | x \in D \cup M \cup R\}$ be the union of all device and robot traces. Based on the above definition we can construct the set of devices and robots for which the updating of the virtual process is needed.

$$\text{UpDate}(s(t')) = \{x \in D \cup M \cup R | (\exists e \in \text{tr}_{[t',t]}(x))((s(t'), t') \rightsquigarrow e)\}$$

Now we define the *local updating* process as follows:

$$(\forall x \in \text{UpDate}(s(t'))) (up(x) = 1 \Rightarrow \lambda_x(s_x, sr_x, 1) = sr_x \in S_V(x))$$

Fact 10.2.1. *It is easy to prove that the local updating process so defined is equivalent to the global one, i.e., synchronization of devices and robots from the set UpDate is equivalent to the synchronization of all devices and robots of the workcell.*

□

□

Moreover, we have the following result:

Fact 10.2.2. *If $t' > t$, then $\text{UpDate} = \{d\}$ and synchronization is necessary only for the device d .*

□

□

The synchronization process is illustrated in Figure 10.2.

10.2.2. Prediagnosis

Let $sr_d(t)$ be the current real state of device d at time t , as registered by the passive Dev_d^p automaton, and let $s_d(t') \in S_V(d)$ be the virtual state of the simulator of device d at time t' .

Let $sr_d(t) = s_d(t')$ and $t' \neq t$, with $t < t' - \tau_0$ or $t > t' + \tau_0$, where τ_0 is the tolerance time window.

In this case the diagnosis of the real cell is performed. To reduce the complexity of such a process we use the same causality relation \rightsquigarrow in order to eliminate devices which do not need to be diagnosed. By $\text{CON}(s(t'))$ we denote the set of events which are direct causes of event $(s(t'), t')$, i.e.,

$$\text{CON}(s(t')) = \{e \in \text{Trace}_{[t',t]} | e \rightsquigarrow (s(t'), t') \wedge (\neg \exists e')(e \rightsquigarrow e' \wedge e' \rightsquigarrow (s(t'), t'))\}$$

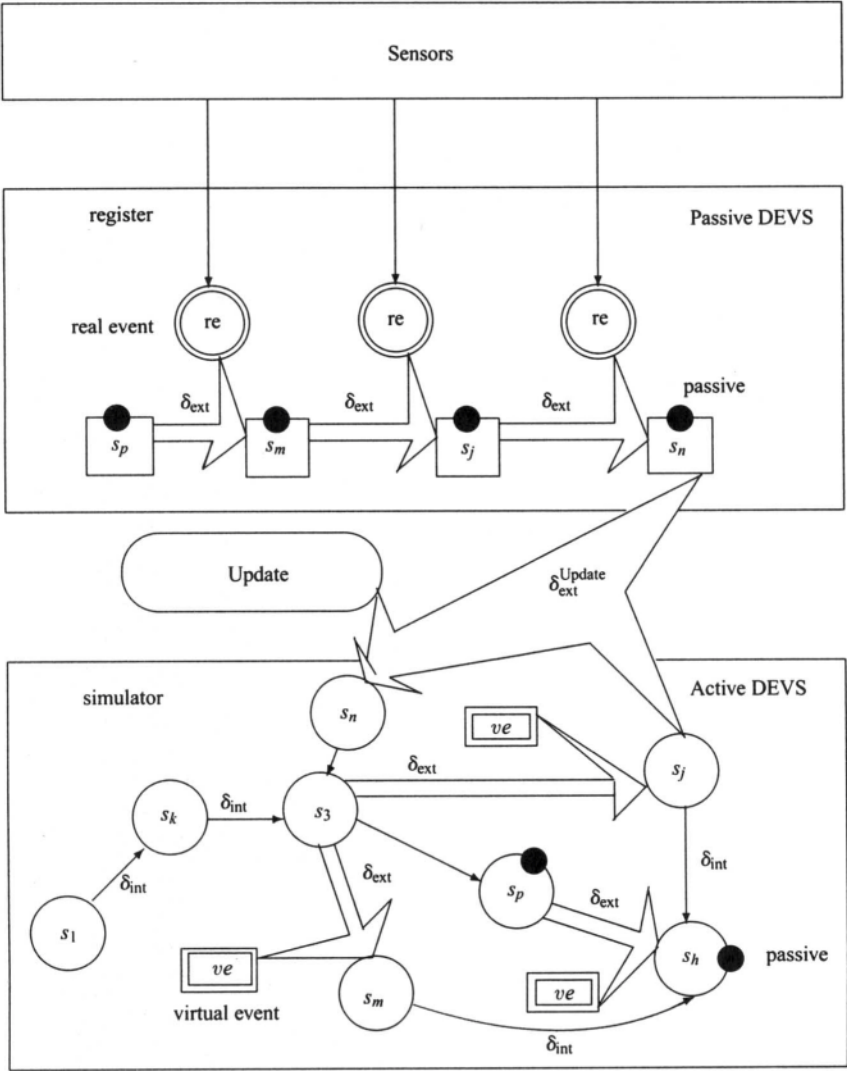


Figure 10.2. Updating process in the DEVS model.

From the set $\mathbf{CON}(s(t'))$ we eliminate those events which were previously monitored, i.e.,

$$\mathbf{Con}(s(t')) = \mathbf{CON}(s(t')) - \mathbf{Monitored_Events}$$

Based on the set $\mathbf{Con}(s(t'))$ we find the devices or robots for which diagnosis is needed as follows:

$$\mathbf{Diag}(s(t')) = \{x \in D \cup M \cup R \mid (\exists e \in \mathbf{tr}_{[t',t]}(x))(e \in \mathbf{Con}(s(t')))\}$$

In addition, a taxonomy of failure types can be performed.

Object-Oriented Discrete-Event Simulator of Intelligent Robotic Cells

Chapters 7–9 present the discrete control of the computer assisted robotic cell (CARC). The on-line control system of the CARC has a hierarchical structure, with three main levels of control: the execution level, the coordination level, and the organization level.

The execution level consists of device controllers. It executes the action programs generated by the coordinator. The coordination level defines routing of the parts from the logical and geometric perspectives and coordinates the activities of workstations and robotic agents. The organizer accepts and interprets related feedback from the executor and the coordinator and defines the strategy of task sequencing to be executed in real time.

Chapter 9 proposed several strategies of task sequencing based on fuzzy rules. To verify and test different task realizations, a model was prepared that simulates the behavior of all the components of the robotic cell. The coordinator level chooses the technological operations to perform based on the priorities of the operations. These depend on the current strategy of operation scheduling and are set up by the organizer.

The presented implementation of the simulator realizes the following objectives:

- It defines various structures of the manufacturing cell and production tasks performed in the cell (these data can be obtained from ICARS system; see Chapter 6).
- It defines the various control strategies produced by the fuzzy rules-based decision-making system.
- It changes the control rule during the simulation.
- It writes various messages which make it possible to observe the behavior of all the devices in the workcell.
- It monitors the WIP (work-in-process) factor and the waiting time during the simulation process and calculates various statistics.

11.1. Object-Oriented Specification of Robotic Cell Simulator

In implementing the robotic cell control simulator we consider a workcell consisting of a set of workstations, stores, robots, and input/output conveyers (see Chapter 2). Such a cell performs the production tasks.

The production task (see Chapter 4) relies on the realization of a pipeline sequence of machining processes on parts. A machining process is an ordered set of operations which have to be performed on a part. The first operation is always performed by an input conveyer which brings a new part. The last operation is always performed by an output conveyer which delivers the finished part. The other operations are machining or storing. The machining operations are performed by workstations and storing operations are performed by stores.

A workstation might need special equipment in order to perform some operations. In this case it has to install this equipment. Installation or uninstallation of equipment is called a *setup*.

All workstations and stores have an input/output buffer. Such buffers store the parts before and after the operations are performed. The parts are transported between devices (workstations, stores, or conveyers) by robots. A robot picks up a part from the buffer of the first device, moves it, and places it onto the buffer of the second device. A robot can move parts only between devices which are in its range. The logical and geometrical model of a workcell (see Chapter 3) can be developed by the ICARS system (see Chapter 6).

11.1.1. Control and Input Files of the Workcell Simulator

On startup the discrete event simulator reads the main input file which contains the description of the logical structure of the robotic workcell and the specification of the production tasks to be performed in this cell.

11.1.1.1. Specification of the workcell resources. The first part of the main input file specifies the logical structure of the workcell (see Chapter 3). It contains the following lists of resources*:

1. List of workstations:

WORKSTATION *name*_{workstation} *number*_{capacity}

where

*Each resource has own logical name, which can be used in the main input file only once.

`nameworkstation` logical name of the workstation
`numbercapacity` capacity of the buffer for parts

2. List of stores:

STORE `namestore` `numbercapacity`

where

`namestore` logical name of the store
`numbercapacity` capacity of the buffer for parts

3. List of input conveyers:

INPUTCONVEYER `nameinput`

where

`nameinput` logical name of the input conveyer

4. List of output conveyers:

OUTPUTCONVEYER `nameoutput`

where

`nameoutput` logical name of the output conveyer

5. List of special equipment*:

EQUIPMENT `nameequipment` `valueinstalltime` `valueuninstalltime`

where

`nameequipment` logical name of the equipment

`valueinstalltime` time needed to install the equipment on a workstation

`valueuninstalltime` time needed to uninstall the equipment from a workstation

*The equipment is installed on a workstation when it is needed. Only one piece of equipment can be installed on a workstation at any given time. If any equipment other than what is currently needed is installed, the previous equipment is uninstalled and the new equipment is installed.

6. List of robots:

ROBOT `namerobot`

where

`namerobot` logical name of the robot

11.1.1.2. Specification of the robot movements. The second part of the main input file specifies all possible robot movements. The current position of the robot is defined by the name of the device where the robot actually is. At the beginning of simulation the robot's position is undefined, and is denoted INIT. There are two lists which describe the robot's motions:

1. List of robot's range:

RANGE `namerobot` `namedevice`

where

`namedevice` is the logical name of the device (workstation, store, input/output conveyer) which is in the range of the robot
`namerobot`

2. List of robot's movements*:

MOVEMENT `namerobot` `namestart` `namefinish` `valueemptytime`
`valueholdingtime`

where

`valueemptytime` is the time of motion from the device[†] `namestart` to the device `namefinish` without any part in the robot's gripper

`valueholdingtime` the time of motion from the device `namestart` to the device `namefinish` with a part in the robot's gripper

*To avoid errors during the simulation all possible movements between devices within the range of the robot should be specified.

[†]The initial robot position is denoted INIT.

11.1.1.3. Specification of the technological tasks. The last part of the main input file contains the specification of the production tasks which should be concurrently performed in the workcell (see Chapter 4). Each task is defined as follows:

```
TASK nametask numberoperations numberparts valuestarttime
      valuefinishtime valuepriority
```

where

name_{task} is the logical name of the task

number_{operations} is the number of operations Which have to be performed on each part

number_{parts} is the number of parts which should be produced

value_{starttime} is the time at which the realization of the task should be started

value_{finishtime} is the time at which the realization of the task should be finished

value_{priority} is the priority of the task

The task declaration is followed by a list of all operations in the form

```
OPERATION nameoperation namedevice nameequipment valueplacetime
      valueoperationtime valuepicktime
```

where

name_{operation} is the logical name of the operation

name_{device} is the logical name of the device which performs the operation

name_{operation}

name_{equipment} is the logical name of the equipment or the keyword

“NONE” if no equipment is needed

value_{placetime} is the time needed for a robot to place a part into the buffer of the device **name_{device}**

value_{operation} is the time needed for the device **name_{device}** to perform the operation

value_{picktime} is the time needed for the robot to pick up a part from the buffer of the device **name_{device}**

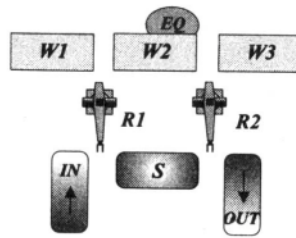


Figure 11.1. Example of a workcell layout.

The first operation has to be performed by the input conveyor and the last one by the output conveyor. The other operations have to be performed by workstations or stores. For all operations performed by workstations without any special equipment and for all operations performed by stores and input/output conveyers the keyword “NONE” should be used instead of `nameequipment`.

Example 11.1.1. As an example of the input file we present here a simple workcell which contains three workstations ($W1$, $W2$, $W3$), a store (S), an input conveyor (IN), an output conveyor (OUT), and two robots ($R1$, $R2$). The workstation $W2$ has special equipment EQ to perform certain operations. To install or uninstall the equipment EQ the workstation $W2$ needs 2.5 units of time. The capacity of the workstations’ buffers is 5, and the capacity of the store’s buffer is 10. The range of the robot $R1$ includes the workstations $W1$ and $W2$, the store S , and the input conveyor IN . The range of the robot $R2$ includes the workstations $W2$ and $W3$, the store S and the output conveyor OUT . The workcell layout is shown in Figure 11.1.

We assume that the time of movement between any two devices is equal to 5 units if there is a part in the robot’s gripper. Otherwise the time of movement between any two devices is equal to 3 units.

The workcell has to perform two production tasks Z_1 and Z_2 . The production process for task Z_1 has four operations: $O1A$, $O1B$, $O1C$, $O1D$. The production process for task Z_2 has five operations: $O2A$, $O2B$, $O2C$, $O2D$, $O2E$. The detailed data needed to specify the production tasks in the main

Table 11.1. Specification Data of the Operations

Operation	task Z_1				task Z_2				
	$O1A$	$O1B$	$O1C$	$O1D$	$O2A$	$O2B$	$O2C$	$O2D$	$O2E$
Device	IN	$W1$	$W2$	OUT	IN	$W2$	S	$W3$	OUT
Equipment	—	—	—	—	—	EQ	—	—	—
Pickup time	1.5	1.0	1.0	0	1.5	1.0	1.0	1.0	0
Operation time	15	35	20	10	15	25	5	40	10
Placement time	0	1.0	1.0	0.5	0	1.0	1.0	1.0	0.5

Table 11.2. Specification Data of the Production Tasks

	Task Z_1	Task Z_2
Number of operations	4	5
Number of parts	1000	600
Start time	0	500
Finish time	1500	2000
Priority	0.7	0.3

input file are given in tables 11.1 and 11.2.

The main input file of the simulator has the following form:

```

WORKSTATION    W1    5
WORKSTATION    W2    5
WORKSTATION    W3    5
STORE          S     10
INPUTCONVEYER  IN
OUTPUTCONVEYER OUT
EQUIPMENT      EQ     2.5    2.5
ROBOT          R1
ROBOT          R2
RANGE          R1    W1, W2, S, IN
RANGE          R2    W2, W2, S, OUT

MOVEMENT      R1    W1    W2    3    5    MOVEMENT      R2    W2    W3    3    5
MOVEMENT      R1    W1    S     3    5    MOVEMENT      R2    W2    S     3    5
MOVEMENT      R1    W1    IN    3    5    MOVEMENT      R2    W2    OUT   3    5
MOVEMENT      R1    W2    W1    3    5    MOVEMENT      R2    W3    W2    3    5
MOVEMENT      R1    W2    S     3    5    MOVEMENT      R2    W3    S     3    5
MOVEMENT      R1    W2    IN    3    5    MOVEMENT      R2    W3    OUT   3    5
MOVEMENT      R1    S     W1    3    5    MOVEMENT      R2    S     W2    3    5
MOVEMENT      R1    S     W2    3    5    MOVEMENT      R2    S     W3    3    5
MOVEMENT      R1    S     IN    3    5    MOVEMENT      R2    S     OUT   3    5
MOVEMENT      R1    IN    W1    3    5    MOVEMENT      R2    OUT   W2    3    5
MOVEMENT      R1    IN    W2    3    5    MOVEMENT      R2    OUT   W3    3    5
MOVEMENT      R1    IN    S     3    5    MOVEMENT      R2    OUT   S     3    5
MOVEMENT      R1    INIT  IN    3    5    MOVEMENT      R2    INIT  W2    3    5
MOVEMENT      R2    INIT  W3    3    5
MOVEMENT      R2    INIT  S     3    5

TASK  Z1    4    1000    0    1500    0.7
OPERATION    O1A    IN    NONE    0     15    1.5
OPERATION    O1B    W1    NONE    1.0   35    1.0
OPERATION    O1C    W2    NONE    1.0   20    1.0
OPERATION    O1D    OUT    NONE    0.5   10    0

TASK  Z2    5    600    500    2000    0.3
OPERATION    O2A    IN    NONE    0     15    1.5
OPERATION    O2B    W2    EQ     1.0   25    1.0
OPERATION    O2C    S     NONE    1.0    5    1.0
OPERATION    O2D    W3    NONE    1.0   40    1.0
OPERATION    O2B    OUT    NONE    0.5   10    0

```

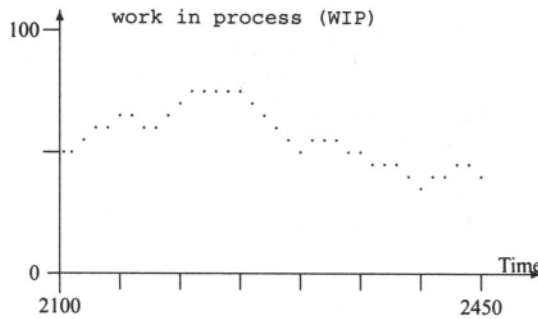


Figure 11.2. An example of a monitor diagram showing the global WIP factor.

11.1.1.4. Monitoring process. The organizer uses the work-in-process (WIP) factor and the mean waiting time as an input variables. In order to observe the current values of these factors the monitor facility is implemented in the simulator. When the simulator works in the monitor mode it draws a diagram showing the current value of the WIP factor. It can draw a global WIP factor or a WIP factor for each task separately. Similarly, it can draw the mean waiting time or the waiting time for each task separately. An example of a monitor diagram is shown in Figure 11.2.

11.1.1.5. Statistics. The statistics calculation module of the simulator prepares the statistics for the organization layer of the CARC. The following options are possible:

robots This option allows the calculation of statistics about the robots. For each robot the simulator calculates the *total times* of:

- waiting
- empty movements
- holding movements
- grasping parts

workstations This option allows the calculation of statistics about the workstations. For each workstation the program writes the *total time* of:

- waiting
- machining
- setting up

In addition, the program writes the mean time for filling of the workstation buffer.

stores This option allows the calculation of statistics about the stores. For each store the program writes the mean time for filling of the store.

tasks This option allows the calculation of statistics about the tasks. For each task the simulator calculates the number of finished parts and the number of parts which are currently in process. If there are some finished parts, the program writes the mean process time, the mean transport time, the mean time of waiting for transport, and the mean time of waiting for machining.

movements This option allows the calculation of statistics about a robot's movements. For each robot the simulator writes the number of empty and holding movements.

The graph module of the simulator makes it possible to draw graphs for, e.g., the WIP factor, the waiting time, the finished parts, and the productivity.

11.2. Object Classes of Robotic Cell Simulator

To implement the simulator the discrete event system specification (DEVS) (see Chapter 7) formalism was employed. In such a formalism one specifies the basic models (atomic DEVS) from which larger ones are to be built and how these models are to be connected together. Each object is modeled by an atomic DEVS; the change of the object's state is called an event. There are active and passive states. If an object's state is active, it changes in a self-acting way into another state after the activation time. A passive state can be changed into another state only under the influence of external events. The application of the DEVS formalism in workcell modeling is described in Chapters 7 and 8.

There are two basic classes of implementation. The first represents atomic DEVS modeling of objects in the workcell and the second represents events. The interdependence between an object's states and events is presented in a graphic way. Figure 11.3 explains the symbols used in the following sections. Subsequent figures show the events and states of atomic DEVS for classes representing various objects.

11.2.1. Classes

The implementation of the discrete event simulator makes extensive use of the class concept of C++. There are classes for all important objects. Figure 11.4 gives a brief overview of the most important classes used in the implementation of the CARC simulator. For each class we give the most important data members and member functions.

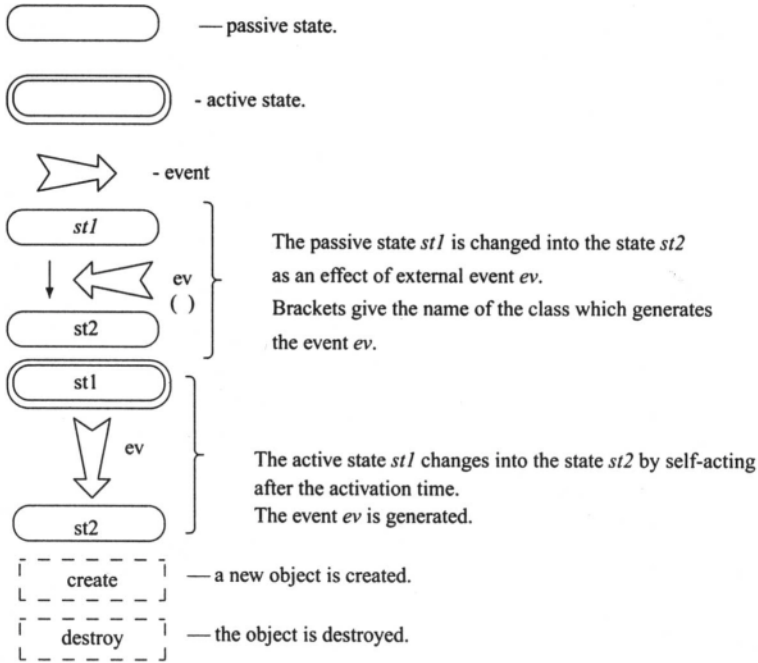


Figure 11.3. Symbols used to explain the interdependence between an object's states and events.

11.2.1.1. Class Buffer. The class **Buffer** is used to represent one place in a buffer. The whole buffer is represented as an array of **Buffer** (see Figure 11.5).
Data members:

Status — State of the place in the buffer. The following values are admissible:
BuffReserved, BuffOccupiedLoading, BuffOccupiedBeforeOper, BuffOccupiedDuringOper, BuffOccupiedAfterOper.

PartPtr — Pointer to the part located in the buffer place.

GrippOcc — Moment when the buffer's place became occupied.

Member function:

Buffer (constructor)—Constructs a new place in the buffer.

11.2.1.2. Class Equipment. The class **Equipment** is used to represent the special equipment needed to perform certain machining operations. All objects belonging to the class **Equipment** are gathered in a list. An object is added to

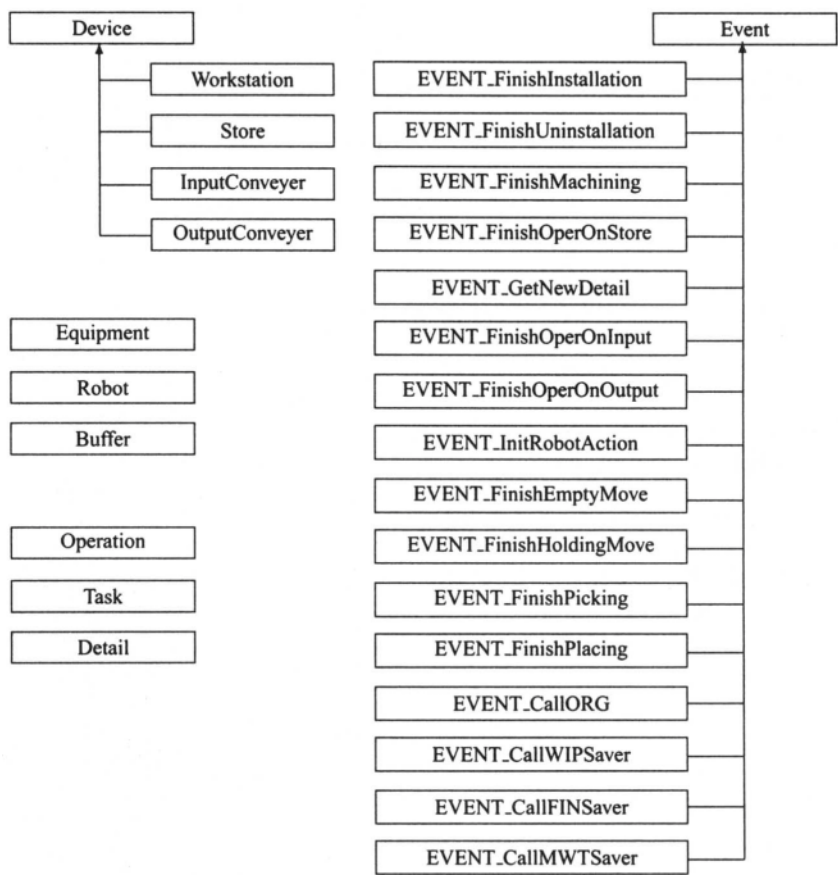


Figure 11.4. Class hierarchy in the discrete event simulator.

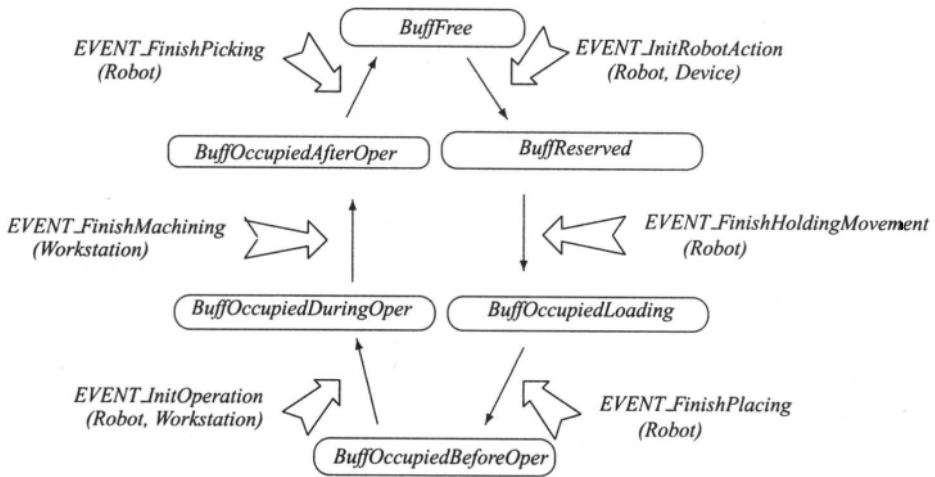


Figure 11.5. States and events for each buffer position in the workstation (class **Buffer**).

the list when the constructor **Equipment** is executed. It is removed from the list when the destructor **~Equipment** is executed.

Data members:

FirstEquipment (static data) — Pointer to the first equipment in the list.

PrevEquipment — Pointer to the previous equipment in the list.

NextEquipment — Pointer to the next equipment in the list.

Name — Name of the equipment.

InstallTime — Time needed by the workstation to install the equipment.

UninstallTime — Time needed by the workstation to uninstall the equipment.

Member functions:

Equipment (constructor) — Constructs new equipment.

~Equipment (destructor) — Destroys equipment.

GetName — Returns the name of the equipment.

GetFirstEquipment (static function) — Returns the pointer to the first equipment in the list.

GetNextEquipment — Returns the pointer to the next equipment in the list.

GetInstallTime — Returns the time needed by the workstation to install the equipment.

GetUninstallTime — Returns the time needed by the workstation to uninstall the equipment.

11.2.1.3. Class Device. This is a base class for all devices in the workcell. All objects belonging to the class **Device** are gathered in a list. An object is added to the list when the constructor **Device** is executed. It is removed from the list when the destructor **~Device** is executed.

Data members:

Name — Name of the device.

FirstDevice (static data) — Pointer to the first device in the list.

PrevDevice — Pointer to the previous device in the list.

NextDevice — Pointer to the next device in the list.

Member functions:

Device (constructor) — Constructs a new object

~Device (virtual destructor) — Destroys an object.

GetName — Returns the name of the object.

DeviceKind (virtual function) — Returns the kind of object; there are the following kinds of objects: IN_CONV, OUT_CONV, WORKST, STORE.

Shared — Returns a nonzero value if the object is shared.

Reserve (virtual function) — Reserves one place in the object's buffer for a part.

OccupyDevice (virtual function) — Occupies a place in the object's buffer.

InitOperation (virtual function) — Initializes an operation on a part.

FreeDevice (virtual function) — Frees one place in the object's buffer.

LW (virtual function) — Returns the number office places in the object's buffer. For the in/output conveyer this function always returns a nonzero value.

11.2.1.4. Class Workstation. This class is used to represent objects performing various machining operations. It is based on the class **Device**. All objects belonging to the class **Workstation** are gathered in a list. An object is added to the list when the constructor **Workstation** is executed. It is removed from the list when the destructor **Workstation** is executed (see Figure 11.6).

Data members:

FirstWorkstation (static data)—Pointer to the first workstation in the list

PrevWorkstation — Pointer to the previous workstation in the list

NextWorkstation — Pointer to the next workstation in the list

Capacity — Capacity of the workstation's buffer

Buf — Pointer to the first place of the buffer The whole buffer is represented as an array which has as many elements as the capacity. Each element of the array is an object belonging to the class **Buffer**.

State —State of the workstation. The following values are admissible: **WAIT**, **UNINSTALL**, **INSTALL**, **WORK**.

AktBuf —Number of active positions of the buffer.

AktEquipment — Pointer to the equipment currently installed on the workstation.

Member functions:

Workstation (constructor) — Constructs a new object.

~Workstation (destructor) — Destroys an object.

GetFirstWorkstation (static function) — Returns the pointer to the first workstation in the list.

GetNextWorkstation — Returns the pointer to the next workstation in the list.

GetCapacity — Returns the capacity of buffer.

StartWork — Starts an operation on the workstation.

FinishInstallation — Finishes equipment installation.

FinishUninstallation —Finishes equipment uninstallation.

FinishMachining — Finishes a machining operation.

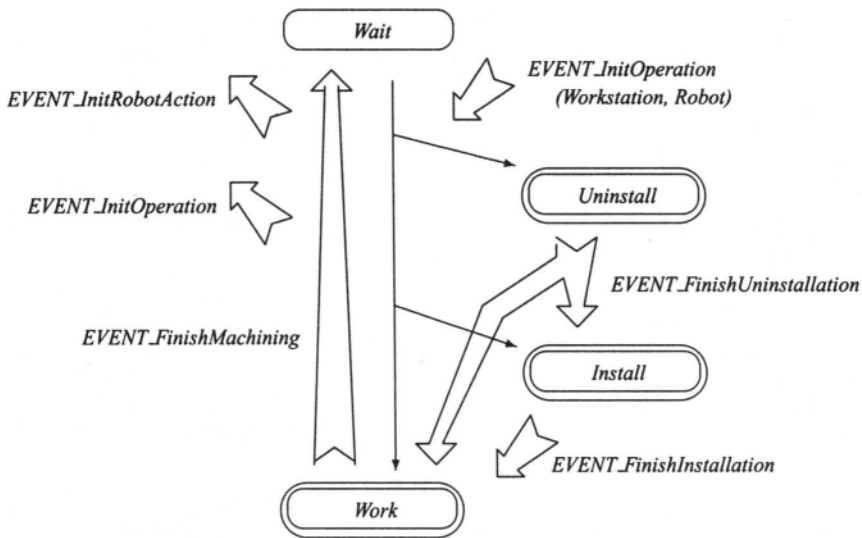


Figure 11.6. States and events for objects belonging to the class **Workstation**.

11.2.1.5. Class Store. This class is used to represent objects that store parts between machining operations. It is based on the class **Device**. All objects belonging to the class **Store** are gathered in a list. An object is added to the list when the constructor **Store** is executed. It is removed from the list when the destructor **~Store** is executed (see Figure 11.7).

Data members:

FirstStore (static data)—Pointer to the first store in the list.

PrevStore — Pointer to the previous store in the list.

NextStore — Pointer to the next store in the list.

Capacity — Capacity of the store's buffer.

Buf — Pointer to the first place of the buffer. The whole buffer is represented as an array which has as many elements as the capacity. Each element of the array is an object belonging to the class **Buffer**.

Member functions:

Store (constructor) — Constructs a new object.

~Store (destructor) — Destroys an object.

GetFirstStore (static function) — Returns the pointer to the first store in the list.

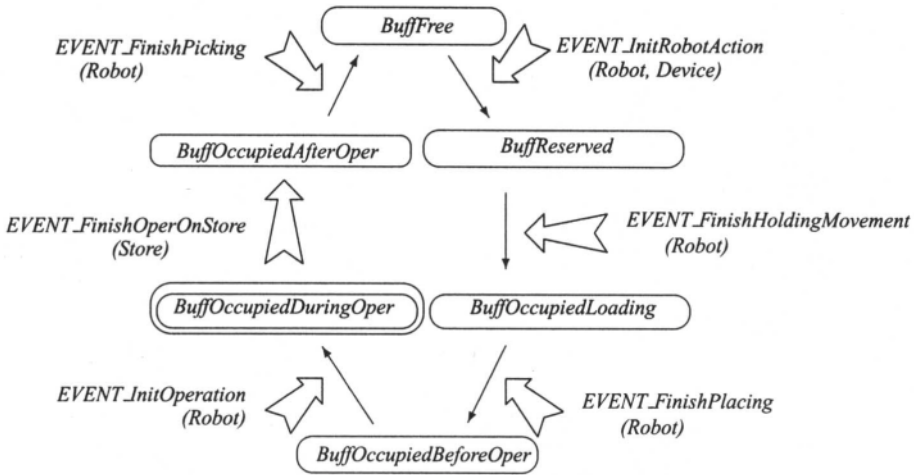


Figure 11.7. States and events for each buffer position in the store (class **Buffer**).

GetNextStore — Returns the pointer to the next store in the list.

GetCapacity — Returns the capacity of the buffer.

StartWork — Starts a storing operation.

FinishStoring — Finishes a storing operation.

11.2.1.6. Class InputConveyer. This class is used to represent objects serving new parts. It is based on the class **Device**. All objects belonging to the class **InputConveyer** are gathered in a list. An object is added to the list when the constructor **InputConveyer** is executed. It is removed from the list when the destructor **~InputConveyer** is executed (see Figure 11.8).

Data members:

FirstInputConveyer (static data)—Pointer to the first input conveyer in the list.

PrevInputConveyer —Pointer to the previous input conveyer in the list.

NextInputConveyer — Pointer to the next input conveyer in the list.

PartPtr — Pointer to the part served by the input conveyer.

ListZad — Pointer to the beginning of the list which contains tasks serviced by the input conveyer.

Member functions:

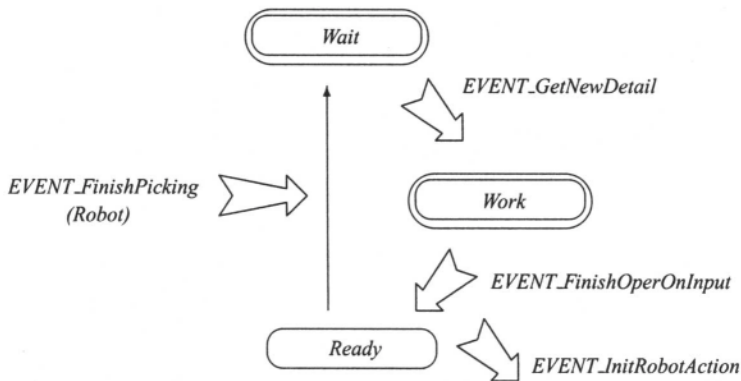


Figure 11.8. States and events for objects belonging to the class **InputConveyer**.

InputConveyer (constructor) — Constructs a new object.

~InputConveyer (destructor) — Destroys an object.

GetFirstInputconv (static function) — Returns the pointer to the first input conveyer in the list.

GetNextInputconv — Returns the pointer to the next input conveyer in the list.

GetNewPart — Starts the first operation on a new part.

11.2.1.7. Class OutputConveyer. This class is used to represent objects sending finished parts. It is based on the class **Device**. All objects belonging to the class **OutputConveyer** are gathered in a list. An object is added to the list when the constructor **OutputConveyer** is executed. It is removed from the list when the destructor **~OutputConveyer** is executed (see Figure 11.9).

Data members:

FirstOutputConveyer (static data) — Pointer to the first output conveyer in the list.

PrevOutputConveyer — Pointer to the previous output conveyer in the list.

NextOutputConveyer — Pointer to the next output conveyer in the list.

PartPtr — Pointer to the part served by the output conveyer.

ListZad — Pointer to the beginning of the list which contains the tasks serviced by the output conveyer.

Member functions:

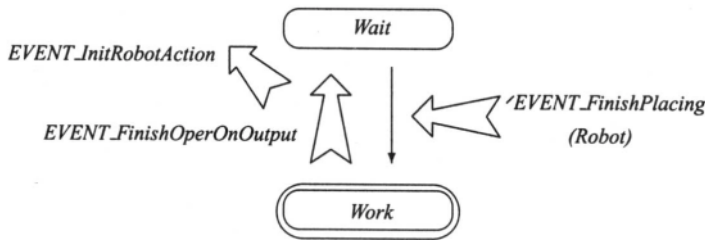


Figure 11.9. States and events for objects belonging to the class **OutputConveyer**.

OutputConveyer (constructor) — Constructs a new object.

~OutputConveyer (destructor)—Destroys an object.

GetFirstOutputconv (static function) — Returns the pointer to the first output conveyer in the list.

GetNextOutputconv — Returns the pointer to the next output conveyer in the list.

11.2.1.8. Class Robot. This class is used to represent the robots in the work-cell. All objects belonging to the class **Robot** are gathered in a list. An object is added to the list when the constructor **Robot** is executed. It is removed from the list when the destructor **~Robot** is executed (see Figure 11.10).

Data members:

FirstRobot (static data) — Pointer to the first robot in the list.

PrevRobot — Pointer to the previous robot in the list.

NextRobot — Pointer to the next robot in the list.

NOFreeRobots (static data)—Number of waiting robots.

Name —Name of the robot.

State — State of the robot. The following values are admissible: **RobWait**, **MoveEmpty**, **Pickup**, **MoveHolding**, **PlaceOn**.

RobPosition — Pointer to the device that the robot is near.

Action — Structure which describes the current robot action. It has the following data members:

PartPtr — Pointer to the part which is moved.

FromDev — Pointer to the device the part is moved from.

ToDev — Pointer to the device the part is moved to.

ToBuf — Position number in the buffer of the goal device.

FromOper — The number of the operation last performed on the moved part.

ToOper — The number of the operation which has to be performed on the part

Range — Pointer to the beginning of the list of the devices which are in range of the robot.

Movements — Pointer to the begin of the list of robot's movements.

Member functions:

Robot (constructor) — Constructs a new robot.

~Robot (destructor) — Destroys a robot.

GetFirstRobot (static function) — Returns the pointer to the first robot in the list.

GetNextRobot — Returns the pointer to the next robot in the list.

GetName — Returns the name of the robot.

InRange — Returns a nonzero value if the specified devices are in the range of the robot.

InitAction — Initializes a new robot action.

InitEmptyMove — Starts an empty movement.

FinishEmptyMove — Finishes an empty movement.

InitPickUp — Starts a "pickup" action.

FinishPickUp — Finishes the "pickup" action.

InitHoldingMove — Starts a holding movement.

FinishHoldingMove — Finishes the holding movement.

InitPlaceOn — Starts a "place on" action.

FinishPlaceOn — Finishes the "place on" action.

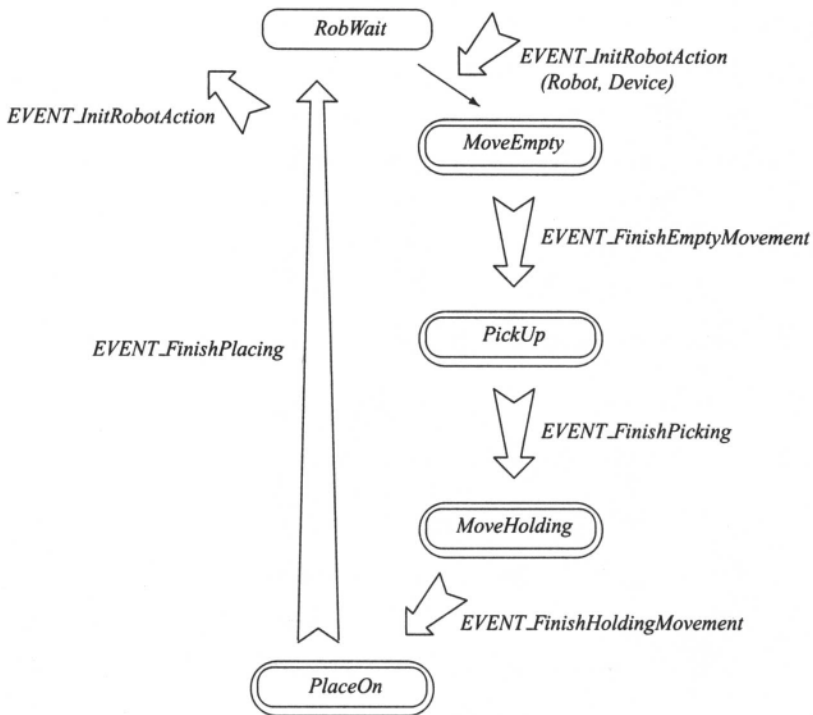


Figure 11.10. States and events for objects belonging to the class **Robot**.

11.2.1.9. Class Operation. This class is used to represent operations which have to be performed on parts. All operations for a task are grouped in an array (see description of the class **Task**).

Data members:

Name —Name of the operation.

Kind —Kind of operation.

Nodefined, GettingPart, SendingPart, StoringPart, MachiningPart.

OperDevice — Pointer to the device which performs the operation.

OperEquipment — Pointer to the equipment which is needed to perform the operation. If no equipment is needed, this pointer takes a value NULL.

SharedDev — Pointer to the first element of the list of shared devices belonging to the same zone as the device carrying out the operation. This information is used by the deadlock-avoidance algorithm.

UnsharedDev — Pointer to the first element of the list of unshared devices belonging to the same zone as the device carrying out the operation. This information is used by the deadlock-avoidance algorithm.

PlacingTime — Time needed for the robot to place a part into the buffer of the device which performs the operation.

MachiningTime — Time needed to perform the operation.

PickingTime — Time needed for the robot to pick up a part from the buffer of the device which performs the operation.

Priority — The priority of the operation. This value is set by the organizer.

Member functions:

Operation (constructor) — Constructs a new object.

~Operation (destructor) — Destroys an object.

GetName — Returns the name of the operation.

GetMachiningTime — Returns the time needed to perform the operation.

GetPickingTime — Returns the time needed for the robot to pick up a part after the operation is finished.

GetPlacingTime — Returns the time needed for the robot to place a part before the operation is started.

GetDevice — Returns the pointer to the device which performs the operation.

GetEquipment — Returns the pointer to an equipment needed to perform the operation.

GetPriority — Returns the current priority of the operation.

SetPriority — Sets a new value of the operation's priority.

11.2.1.10. Class Task. This class is used to represent the production tasks to be realized in the workcell. All objects belonging to the class **Task** are gathered in a list. An object is added to the list when the constructor **Task** is executed. It is removed from the list when the destructor **~Task** is executed.

Data members:

FirstTask (static data) — Pointer to the first task in the list.

NextTask — Pointer to the next task in the list.

PrevTask — Pointer to the previous task in the list.

Name —Name of the task.

NOOper — Number of operations which have to be performed on a part.

OperTabl — Pointer to the first element of the array which contains the operations.

NOPart — Number of parts which have to be produced.

NOStartedPart —Number of parts the production process has been started on.
This value is calculated on-line during the simulation.

Member functions:

Task (constructor) — Constructs a new object.

~Task (destructor) — Destroys an object.

11.2.1.11. Class Part. This class is used to represent the parts which are currently in process. All objects belonging to the class **Part** are gathered in a list. An object is added to the list when the constructor **Part** is executed. It is removed from the list when the destructor **~Part** is executed (see Figure 11.11).

Data members:

FirstPart (static data) — Pointer to the first part in the list.

NextPart — Pointer to the next part in the list.

PrevPart — Pointer to the previous part in the list.

NOActiveParts (static data)—Number of parts with which the operation is realized.

NOWaitingParts (static data)—Number of parts waiting for the next operation.

Name — Name of the part.

TaskPtr — Pointer to the task the part belongs to.

State — **State of the part. The following values are admissible:**
Performed, WaitForMachining, WaitForTransport,
Assigned, Transported.

ChwZm — Time instant the part's state was changed.

WaitingForMachiningTime — Total time the part has been waiting for the machining operation.

MachiningTime — Total time the machining operations have been performed on the part.

WaitingForTransportTime — Total time the part has been waiting for transport.

TransportTime — Total time the part has been transported.

Member functions:

Part (constructor) — Constructs a new part.

~Part (destructor) — Destroys the part.

GetName — Returns the name of the part.

GetFirstPart (static function) — Returns the pointer to the first part in the list.

GetNextPart — Returns the pointer to the next part in the list.

GetActOper — Returns the pointer to the operation which is actually performed on the part.

GetTask — Returns the pointer to the task the part belongs to.

StartMachining — Starts the machining operation on the part.

FinishMachining — Finishes the machining operation on the part.

11.2.1.12. Class Event. This is a base class for all **EVENT_...** classes. All objects belonging to the class **Event** are gathered in an ordered list. An object is added to the list when the constructor **Event** is executed. It is removed from the list when the destructor **~Event** is executed. This class has a virtual function **Handle**. This function is called by the event management system when the event take place. After that the virtual destructor is called and the object is removed from the list. If there is no object in the event list, the simulation is stopped.

Data members:

FirstEvent (static data) — Pointer to the first event in the list.

NextEvent — Pointer to the next event in the list.

PrevEvent — Pointer to the previous event in the list.

ActivationMoment — Time instant the event was generated.

PassMoment — Time instant the event takes place.

Member functions:

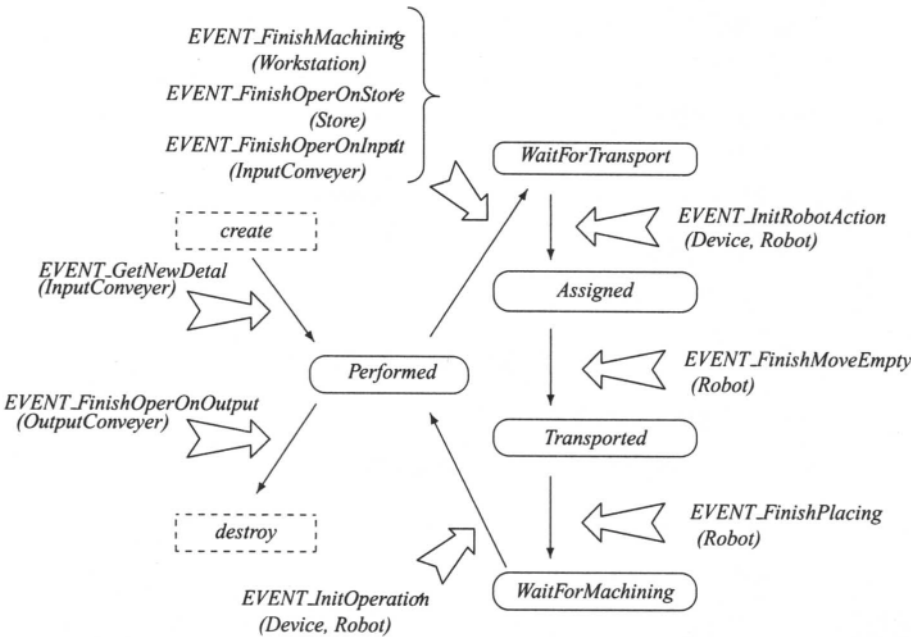


Figure 11.11. States and events for objects belonging to the class **Part**.

Event (constructor) — Constructs a new event.

~Event (virtual destructor) — Destroys the event.

EventKind (virtual function) — Returns the kind of event.

Handle (virtual function)—Executed when the event takes place.

11.2.1.13. Classes EVENT_... Those classes are used to represent all events taking place in the simulated workcell. All objects belonging to these classes are based on the class **Event**. Each class represents a specific kind of event generated by the objects which represent devices, robots, etc.

These classes have a virtual function **Handle**. This function is called by the event management system when the event take place. After that the virtual destructor is called and the object is removed from the list. If there is no object in the event list, the simulation is stopped.

11.3. Object-Oriented Implementation of Fuzzy Organizer

The coordinator of the workcell chooses the technological operations to be performed based on the operations' priorities. The priorities depend on the current strategy of operation scheduling and are set up by the organizer. The organizer is started during the simulation on discrete time instants.

The event management system of the simulator generates the event **EVENT_Call_ORG** periodically. When the event **EVENT_Call_ORG** takes place, the current state of the workcell is measured and the values of the input variables are calculated. Next the main organizer's procedure (*ORG_Do*) is executed. It returns the priorities of the operations for each task.

The *ORG_Do* procedure is called with the following parameters:

- *Time*—the current time instant of simulation
- *ParamPtr*—the pointer to the vector of input variables
- *FinishedPtr*—the pointer to the vector of finished parts
- *WaitingPtr*—the pointer to the array of waiting times

The presented version of the organizer uses two input variables: (1) the work-in-process (WIP) factor which is the number of parts currently in process, and (2) the mean waiting time (MW), which is calculated as follows:

$$MW = \frac{\text{Waiting_Time}}{\text{Process_Time}} \cdot 100\%$$

where *Waiting_Time* is the total waiting time for all parts finished since the last call of the organizer, and *Process_Time* is the total processing time for all parts finished since the last call of the organizer.

The vector of finished parts contains for each task the number of parts which have been finished. The array of waiting times contains for each operation the current total time the parts have been waiting for the operation.

The *ORG_Do* procedure returns the pointer to the array which contains the actual priority for each operation.

11.3.1. Object-Oriented Fuzzy Organizer

This section describes rules for implementing the fuzzy organizer. The theoretical background of the organizer is given in Chapter 9 and (Jahn, 1996).

The organizer is responsible for keeping the load of the cell components at a proper amount. Figure 11.12 shows the embedding of the organizer within the simulator. While the simulator is running, the organizer accepts the state of the

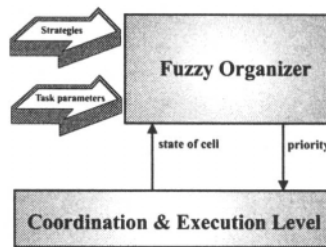


Figure 11.12. Role of the organizer.

cell at discrete time instants. Its task is to define the priorities of the operations for each task. These priorities are used by the execution level to determine which part has to be moved from one machine to another or which operation has to be performed if several parts concurrently require movement or processing.

The process of determining these priorities is influenced by

- the state of the cell — measured by several input variables
- the task specifications — given by the simulator's main input file
- the strategies selected by the rule database — specified by the organizer's control file

The rule database is organized as a fuzzy logic rule database according to Chapter 9. Its structure is defined in the following subsections.

The effect of holding the load of the cell in a specific range is that the flow of parts is regulated, which results in optimized throughput and therefore a shorter completion time for the tasks. Prior to this optimization step the desired values of the input variables need to be determined. This is given in Section 11.3.3.1.

11.3.1.1. Control files of the organizer. On startup the organizer reads the main input file of the simulator and the default control file.* While the simulator is running the user can switch interactively to an alternate control file.

Syntax of the control file. The structure of the control file using EBNF is as follows[†]:

* Both files are specified on the command line when the simulator is started.

[†] Subscripted text is to be interpreted as semantic context; it aids in understanding the meaning of the symbols.

**Table 11.3. List of
Parameters Passed to the
Organizer**

Number	Description
0	WIP (work in process)
1	TW (waiting time)

```

control-file = input sets strategies rules closing.
input-sets = "INPUT_SETS" iset {iset}.
iset = "BASE" namebaiaint "KEY" numberkey ling_block {ling_block}.
ling_block = "LING" nameling ling_val {ling_val}.
ling_val = valuebaseset valueu.
strategies = "STRATEGIES" strat {strat}.
Strat = "JIT" | "POSH" | "MS" | "SOR" | "LOR" | "ONE" | "MW".
rule = namerule ":" "IF" cond {"AND" cond} "THEN" strat.
cond = namebaseset "IS" nameling.
closing = weights distance [weight dprio] report rate.
weights_distance = "WEIGHT PARTS" number "WEIGHT_TIME" number.
weight_dprio = "WEIGHT DPRIOR" number.
report_rate = "REPORT" number.
number = ["+"|"-" ] digit {digit}.
value = number ["." {digit}].
name = alpha {alpha | digit | "-".}.
alpha = "a"|"b"|"c" ... |"z" | "A"|"B"|"C" ... |"Z".
digit = "0"|"1"|"2" ... "9".

```

The control file starts with the definition of input sets in which linguistic variables are defined on the input variables which reflect the state of the cell. **namebaseset** acts as a label to refer to this variable in the rules below. The expression **numberkey** couple **namebaseset** with a parameter passed from the coordination level to the organizer. Table 11.3 shows the parameters available.* To every input variable a set of linguistic variables is assigned using **ling_block**. Such a definition consists of several pairs of numbers which specify the edges of the graph. Linear interpolation is used between the edges. Values outside the defined range are assumed to have the same membership function as the nearest defined edge. Figure 11.13 shows the graphical representation of the definition given in table 11.4. This defines a linguistic variable *good* on a base set WIP using parameter zero.

11.3.1.2. Strategies. The definition of input sets to be used in the rule base is followed by a declaration of the strategies to be used in the rule base. Seven strategies are known by the organizer.

Strategies assign a value to every operation of the production routes. These values are interpreted as the membership function of operations. The abbreviations used to identify these strategies are as follows:

*In addition, the average waiting time for every operation is also passed. This is used by the strategy *Minimum Waiting Time (MW)*.

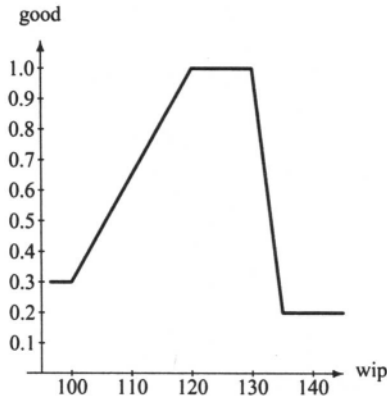


Figure 11.13. Graphical representation of a linguistic variable.

ONE The value 1 is assigned to every operation. In this case every operation has the same priority. This simulates the behavior without control by the organizer. It is useful for testing purposes and in the case where the state of the cell is in a desired state where no intervention is necessary.

JIT Parts that are almost finished by the cell are preferred by this strategy:

$$\mu(o_i) = i/L$$

where o_i denotes operation i of a task, and L denotes the length of the production route under consideration.

PUSH Parts that are at the beginning of their production route are preferred:

$$\mu(o_i) = (L - i + 1)/L$$

MS The operation with the shortest setup time is assigned a membership of 1. Other operations have a lower membership according to their setup time. These membership functions are built on startup and therefore do not regard the current state of the workstations.

Table 11.4. Example of a Linguistic Variable

BASE	wip	KEY 0
	LING	good
		100 0.3
		120 1
		130 1
		135 0.2

MW The waiting time of operations passed by the coordination level is used as a membership function directly. Operations with long waiting times are preferred. See also Section 9.2.

SOR The operation with the shortest processing time is assigned a membership of 1. Other operations get lower membership functions according to their processing time.

LOR The opposite of SOR. Long operations are preferred.

For every strategy declared in the input file the organizer allocates a vector of membership functions for each task. If there are many tasks, then omitting strategies from the **strategies** clause that are not used in the rules base can save a notable amount of main memory.

11.3.1.3. Rule base. The main purpose of the control file is the definition of the rules base. It combines the linguistic variables from the **input_sets** clause and the declared strategies to form a mixed strategy according to the current state of the cell. If the rule base is formed properly, the resulting mixed strategy *pulls* the current state to the desired one.

The rules base consists of a number of rules which select strategies for concrete values for the input variables. For details of evaluating and combining rules see Chapter 9. In a rule not every input variable must occur. Furthermore, the author of the rules base is responsible for ensuring that all cases that may happen during simulation are covered by rules.

11.3.1.4. Closing block. The **closing** block contains constants that define parameters for calculating the realization state of a task. For details see Section 9.2. The influence of the dynamic priority, which itself is based on the parameters discussed above, is regulated by the clause **weight_dprio**. Selecting zero eliminates the dynamic priority from the resulting priorities for the operations. This is useful for tuning the linguistic variables where the realization time for the task is not regarded.

The last clause is named **report_rate**. It determines how often reports are generated from the organizer into the output files. It acts as a divisor to the number of calls to the organizer.

11.3.2. Object Classes for Fuzzy Organizer

The implementation makes extensive use of the class concept of C++. Classes exist for the following objects:

- elements from the main input file
 - equipment

- workstation
- operation
- task
- elements from the control file
 - input set
 - linguistic variable
 - strategy
 - rule

The class **Task** includes vectors of the current membership values for every operation within a task. All of the above objects are gathered in lists.

The lexical analyzer is built with a base class BASELEX which contains methods to read and print lines from an input set. The class SYMLEX is a refinement of BASELEX and introduces methods for lexical analysis such as finding symbols, maintaining a names list, and recognizing and skipping comments.

11.3.3. *Determining the Optimal Values of the Fuzzy Organizer Parameters*

11.3.3.1. Tuning. Finding proper strategies and parameters for the control file is essential for tuning the behavior of the cell. This task is broken down into several subtasks:

1. Determine input parameters.
2. Define linguistic variables on these parameters.
3. Find strategies for combinations of the linguistic variables.
4. Determine the optimal values of the input parameters since they are regulated by the controller.
5. Set the data of the linguistic variables according to the optimal values.

11.3.3.2. How strategies work. Strategies pull the state of the cell (reflected by the input parameters) to proper values. Figure 11.14 shows an example with two input parameters and three linguistic variables defined on every input parameter. The goal is to keep the values of the input parameters in the middle range (**criterion 1 is good, criterion 2 is good**). For this reason we need to place strategies on the other positions that pull the parameters toward the middle position as is done in the upper left corner of Figure 11.14 (*low, high*). Sometimes it is difficult to find a

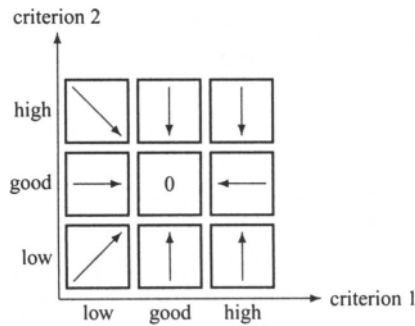


Figure 11.14. How strategies work.

strategy that points exactly to the middle position. In this case another strategy can be selected that goes to other positions, which then point to the middle position as shown in the upper right corner, $(high, high) \rightarrow (high, good) \rightarrow (good, good)$.

The middle position $(good, good)$ reflects the desired state of the cell. So we place a neutral strategy (like ONE) on this position, where the same priority is assigned to all operations.

To find the desired values of the input parameters we first determine the bandwidth of these parameters. As an example, the upper bound of WIP depends on the capacity of the magazines and workstations, and on the deadlocks that can occur when production routes loop in the cell or tasks share the same workstations. The lower bound of WIP is given by the logic of the coordination level: If a part is waiting for transfer or processing and the desired task can be performed and there is no other part waiting for the same task, this part is selected not regarding the current priority of the part.

The lower bound of WIP can be observed by running the simulator with a control file like the following:

```

INPUT_SETS
  BASE wip key 0
  LING any 0 1
STRATEGIES  JIT
  rule_1:    IF wip IS any THEN JIT
WEIGHT_PARTS 1000
WEIGHT_TIME  1000

```

This selects the strategy JIT for all values of WIP and therefore loads as few parts as possible into the cell. The upper bound of WIP can be found by selecting the strategy PUSH for all values of WIP.

The optimal value of WIP which will result in a minimal processing time of all tasks lies somewhere within the resulting bandwidth. It can be found by experiment.

11.3.3.3. Data for linguistic variables. The data that define the linguistic variables set the desired state of the cell. If the rules select strategies that point to the proper direction, we can select any load of the cell within the controllable bandwidth only by changing the data that define the linguistic variables. Again the right shape of these variables can be determined by experiment.

11.3.4. Calculation of Priorities

The priority of an operation is determined by the output of the rule base, the general priority of the task, and the dynamic priority of the task (see Section 9.2). For any operation its priority ρ is built as

$$\rho = \rho_o \rho_s (1 + w_d (\rho_d^i - 1))$$

where

- ρ_o denotes the priority of the operation according to the rule base
- ρ_s is the static priority of the task given in the main input file
- w_d is the weight for the dynamic priority given by the entry `WEIGHT_DPRIO` in the control file; its default is 1
- ρ_d^i denotes the dynamic priority of the task at state i

The priority according to the rule base ρ_o is built on the membership values μ_r of the strategies and the weights w_r for the strategy of rule r ,

$$\rho_o = \frac{\sum (\mu_r w_r)}{\sum w_r}$$

Here a weighted average is used. This guarantees smooth switching from one rule to another when input parameters change with time.

The dynamic priority at state i is calculated as

$$\rho_d^i = \rho_d^{i-1} + \frac{1 - \rho_d^{i-1}}{1 + d_i}$$

where

- ρ_d^{i-1} denotes the dynamic priority at the former state $i - 1$
- d_i is the distance of the task at state i from its conclusion

This distance of a task d_i to its finishing state is calculated as

$$d_i = \begin{cases} 0 & \leftrightarrow t_i < t_s \\ \min \left(w_p \frac{N_f - N_p}{N_f}, w_t \frac{t_f - t_i}{t_f - t_s} \right) & \leftrightarrow t_s \leq t_i \leq t_f \\ 1 & \leftrightarrow t_i > t_f \end{cases}$$

where

- t_i denotes the current time (at state i)
- t_s is the start time of the task given in the main input file
- t_f is the finish time of the task, also from the main input file
- N_f is the whole number of parts to be processed for this task (main input file)
- N_p is the number of parts already finished by the cell at the current time
- w_p denotes the weight of the parts term given in the **WEIGHT_PARTS** entry of the control file
- w_t is the weight of the time term given in the **WEIGHT_TIME** entry of the control file

This page intentionally left blank

References

- Aggarwal, J. K. and Y. F. Wang (1991). Sensor data fusion in robotic systems, *Control and Dynamic Systems*, **39**: 435–461.
- Aggerwal, A., B. Chazelle, L. Guibas, C. O'Dúnlaing, and C. Yap (1988). Parallel Computational Geometry, *Algorithmica*, **3**: 293–327.
- Ahmadabi, M. N. and E. Nakano (1996). A cooperation strategy for a group robots, Proc. of IROS'96, Osaka, pp. 125–132.
- Alander, J. T. (1993). On Robot Navigation Using a Genetic Algorithm, *Artificial Neural Networks and Genetic Algorithms Proc.* (ed.) Albrecht, Reeves, and Steele, Springer Verlag.
- Baldwin, J. and N. Guild (1980). Axiomatic approach to implication for approximate reasoning with fuzzy logic, *Fuzzy Sets System*, **3**.
- Banaszak, Z. and E. Roszkowska (1988). Deadlock Avoidance in Concurrent Processes, *Foundations of Control*, **18**(1/2): 3–17.
- Barraquand, J., B. Langlois, and J. C. Latombe (1992). Numerical potential field techniques for robot path planning, *IEEE Trans. System, Man, Cybernetics*, SMC-22(2), 224–240.
- Bedworth, D. D., M. R. Henderson, and P. Wolfe (1991). *Computer Integrated Design and Manufacturing*. McGraw Hill Int. Edit.
- Black, J. T. (1988). The Design of Manufacturing Cells (Integrated Manufacturing Systems), *Proc. of Manufacturing International Conference*, Vol. III, pp. 143–157.
- Borm, J. H. and C. H. Menq (1991). Determination of Optimal Measurement Configurations for Robot Calibration Based on Observability Measure, *International Journal of Robotics Research*, **10**(1): 51–63.
- Brady, M., ed. (1986). *Robot Motion: Planning and Control*, MIT Press.
- Brooks, R. (1983). Planning Collision-Free Motions for Pick-and-Place Operations, *International Journal of Robotics Research* **2**(4): 19–44.
- Buzacott, J. A. (1985). Modeling manufacturing systems, *Robotics and Computer Integrated Manufacturing*, **2**: 1.
- Carpenter, G. A. and A. H. Tan (1993). Fuzzy ARTMAP, rule extraction and medical databases, *Proc. of the World Congress on Neural Networks*, Portland, Oregon, Vol. 1, pp. 501–506.
- Carpenter, G. A., S. Grossberg, and D. B. Rosen (1991). Fuzzy ART: Fast stable learning and categorization of analog patterns in an adaptive resonance system, *Neural Network*, **4**: 759–71.
- Chang, P. H. (1987). A closed-form solution for inverse kinematics of robot manipulators with redundancy, *IEEE Journal on Robotics and Automation*, RA-3(5), 393–403.
- Chester, D. L. (1991). Why two hidden layers are better than one, IJCNN'90–1, pp. 256–68.
- Chirikjian, G. S. and J. W. Budrick (1990). An obstacle avoidance algorithm for hyper redundant manipulators, *Proc. of IEEE International Conference on Robotics and Automation*, pp. 625–631.
- Chroust, G. (1992). Software-Development Environment, *Informatic Spectrum*, **15**: 165–74.

- Chroust, G. and W. Jacak (1996). Software Process, Work Flow and Work Cell Design Separated by a Common Paradigm? *Lecture Notes in Computer Science*, Springer Verlag, Berlin, Vol. 1030, pp. 403–17.
- Chung, W. J., W. K. Chun, and Y. Youm (1992). Kinematic control of planar redundant manipulators by extended motion distribution scheme, *Robotica*, **10**(3): 255–262.
- Coffman, E. G., M. Elphick, and A. Shoshani (1971). System Deadlock, *Computing Surveys*, **3**(2): 67–78.
- Cohen-Tannoudji, B., C. Diu, and F. Laloe (1977). *Quantum Mechanics*, John Wiley and Sons.
- Craig, J. J. (1981). *Introduction to Robotics*. Cambridge, MIT Press.
- Cutkosky, M. R. (1985). *Robotic Grasping and Fine Manipulation*, Kluwer Academic Publishers, Boston.
- Deitel, H. M. (1983). *An Introduction to Operating Systems*. Wesley, Addison.
- Denavit, J. and R. S. Hartenberg (1955). A kinematic notation for lower-pair mechanisms based on matrices, *ASME Tran. Journal of Applied Mechanics*, **77**: 215–221.
- Dubey, R. and J. Y. S. Luh (1987). Redundant robot control for higher flexibility, *Proc. of IEEE International Conference on Robotics and Automation*, pp. 1066–1072.
- Dubois, D. and H. Prade (1985). The generalized modus ponens under sup-min composition, in *Approximate Reasoning in Expert System*, North Holland.
- Duelen, G. and K. Schröer (1991). Robot Calibration-Method and Results, *Robotics and Computer-Integrated Manufacturing*, **8**: 223–231.
- Edelsbrunner, H. (1987). *Algorithms in Combinatorial Geometry*, Springer Verlag Berlin.
- Efstathiou, J. (1987). Rule based process control using fuzzy logic, in: *Approximate Reasoning in Intelligent Systems. Decision and Control*, L. Zadeh (ed.), Pergamon.
- Elsley, R. K. (1988). A learning architecture for control based on back-propagation neural networks *International Conference on Neural Networks ICNN'88-2*.
- Espiau, B. and R. Boulic (1985). Collision avoidance for redundant robots with proximity sensors, *III Symp. Robotic Research*, pp. 243–251.
- Faverjon, B. (1986). Object Level Programming of Industrial Robots, *IEEE International Conference on Robotics and Automation*, Vol. 2, pp. 1406–1411.
- Feddema, J. T. and Shaheen Ahamed (1985). Determining a Static Robot Grasp for Automated Assembly, *IEEE SMC Conference*.
- Gallant, A. R. and H. White (1988). There exists a neural network that does not make avoidable mistakes, pp. 657–664, *International Conference on Neural Networks ICNN'88-1*.
- Gatrell, L. B. (1989). CAD-Based Grasp Synthesis Utilizing Polygons, Edges Vertexes, pp. 184–89, *IEEE SMC Conference*.
- Geneserth, M. R. and N. J. Nilsson (1987). *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers.
- Gilbert, E. G., D. W. Johnson, and S. Keerth (1988). A fast procedure for computing the distance between complex objects in three-dimensional space, *IEEE Trans. on Robotics and Automation*, RA-4(2), 193–203.
- Goldenberg, A. A., B. Benhabib, and R. G. Fenton (1985). A complete generalized solution to the inverse kinematics of robots, *IEEE Journal on Robotics and Automation*, RA-1 (1), 14–20.
- Guez, A. and Z. Ahmad (1998a). Accelerated convergence in the inverse kinematics via multilayer feedforward networks, pp. 341–344, *International Conference on Neural Networks ICNN'89-2*.
- (1998b). Solution to the inverse kinematics problem in robotics by neural networks, pp. 617–624, *International Conference on Neural Networks ICNN'88-2*.
- Guo, J. H. and V. Cherkassky (1989). A solution to the inverse kinematic problem in robotics using neural network processing, *IEEE Joint International Conference on Neural Network*, San Diego CA, Vol. 2, pp. 299–304.

- Haddadi, A. (1995). *Communication and cooperation in agent system*, Lecture Notes in Computer Science 1056, Springer Verlag, Berlin, New York.
- Han, J. Y. and M. R. Sayeh (1989). Convergence and limit points of neural network and its application to pattern recognition, *IEEE Trans. System, Man, Cybernetics*, SMC-19(5), 1217–1222.
- Han, M. and B. Zhang (1994). Control of robotics manipulators using a CMAC-based reinforcement learning system, *Proc. of IEEE Conference on Intelligent Robots and Systems IROS'94*, Munich, Germany.
- Hecht-Nielsen, R. (1987). Counterpropagation networks, *IEEE First International Conference on Neural Network*, Vol. 2., pp. 19–32.
- Hirsch, M. W. and S. Smale (1974). *Differential Systems and Linear Algebra*, Academic Press, New York.
- Homem De Mello, L. S. and A. C. Sanderson (1990). AND/OR Graph Representation of Assembly Plans, *IEEE Trans. on Robotics and Automation*, 6(2): 188–199.
- Homem De Mello, L. S. and S. Lee (1991). *Computer Aided Mechanical Assembly Planning*, Kluwer Academic Publisher.
- Hornik, K. (1991). Approximation Capabilities of Multilayer Feedforward Networks, *Neural Networks*, 4: 251–257.
- Hornik, K., M. Stinchcombe, and H. White (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks, *Neural Networks*, 3: 551–60.
- Jacak, W. (1985). Time variant goal space: A fuzzy set approach, *Systems Science*, 11(3).
- (1989a). A Discrete Kinematic Model of Robot in the Cartesian Space, *IEEE Trans. on Robotics and Automation*, 5(4): 435–446.
- (1989b). Strategies for Searching Collision-Free Manipulator Motions: Automata Theory Approach, *Robotica*, 7: 129–138.
- (1990). Robot Task and Motion Planning, *AI, Simulation and Planning in High Autonomy Systems*, IEEE Computer Society Press, 168–176.
- (1991). Discrete Kinematic Modelling Techniques in Cartesian Space for Robotic System, in: *Advances in Control and Dynamics Systems*, (ed.) C. T. Leondes, Academic Press.
- (1994a). Fuzzy Rules Based Control of Intelligent Robotic Workcell, in *Cybernetics and Systems'94*, (ed.) R. Trappl, World Scientific, Vienna, Singapore.
- (1994b). Graph Searching and Neural Processing Approach to Robot Motion Planning, *Technical Report of RISC*, Austria.
- (1995). Hybrid Evolutionary approach to control of intelligent arm, *Proc. of II International Symposium on Methods and Models in Automation and Robotics MMAR'95*, August, Szczecin, Poland.
- (in press). ICARS Simulation based CAPP/CAM System for Intelligent Control Design in Manufacturing Automation, in *The International Handbook on Robot Simulation System*, (ed.) D. W. Wloka, John Wiley and Sons, (in preparation).
- Jacak, W. and A. Senanayake (1995). Neural Network Approach for Optimal Grasping, *Proc. of International Conference on Recent Advances in Mechatronics ICRAM'95*, Istanbul, Turkey, August, pp. 917–22.
- Jacak, W. and B. Buchberger (1996a). Intelligent Robotic Arm based on Reactive Control, *Robotics in Alpe-Adria-Danube Region RAAD'96*, Budapest, Hungary, pp 297–303.
- (1996b). Intelligent Systems Combining Reactive and Learning Capabilities, *Cybernetics and Systems '96* R. Trappl (ed.) Vienna, Austria.
- Jacak, W., B. Buchberger, and R. Muszynski (1995a). Symbolic Computation based Synthesis of Neural Network Dynamics and Controller for Robots, *Proc. of IEEE International Conference on Neural Networks ICNN'95*, November IEEE Press, Perth, Australia pp. 2720–2725.

- Jacak, W. and B. Kapsammer (1993). Automatic Programming in CIM based on Intelligent Tools in: *Information Management in Computer Integrated Manufacturing*, (ed.) H. Adelsberger, V. Marik, Lecture Notes in Computer Science, Vol. 973, Springer Verlag, Berlin, New York, pp. 392–418.
- Jacak, W. and G. Jahn (1995). Distributed Intelligent Control of an Autonomous Robotic Cell, in *Intelligent Autonomous Systems IAS'95*, (ed.) R. Dillman, IOS Press, Amsterdam, Oxford, Karlsruhe, pp. 441–49.
- Jacak, W., H. Praehofer, G. Jahn, and G. Haider (1994a). Supervising Manufacturing System Operation by DEVS-based Intelligent Control, *AI, Simulation and Planning in High Autonomy System*, (ed.) Fishwick B., Gainesville, Florida, USA, IEEE Computer Society Press, Los Alamitos, California.
- Jacak, W., I. Duleba, and P. Rogaliński (1992). A Graph-Searching Approach to Trajectory Planning of Robot's Manipulator, *Robotica*.
- Jacak, W., I. Duleba, and R. Muszynski (1995b). Hybrid Computing Approach to Robot Control in the Presence of Uncertainties, *Proc. of IFAC International Conference on Motion Control*, September, Munich, Germany, pp. 664–671.
- Jacak, W. and J. W. Rozenblit (1992a). Automatic Robot Programming, *Lecture Notes in Computer Science* Vol. 585, Springer Verlag, pp. 168–187.
- (1992b). Automatic Simulation of a Robot Program for a Sequential Manufacturing Process, *Robotica*.
- (1993). Virtual Process Design Techniques for Intelligent Manufacturing, *AIS'93 AI, Simulation and Planning in High Autonomy Systems*, IEEE Computer Soc. Press, Tucson.
- (1994). CAST Tools for Intelligent Control in Manufacturing Automation, *Lecture Notes in Computer Science*, Vol. 763, Springer Verlag, pp. 203–220.
- (in press). Model-based Workcell Task Planning and Control, *IEEE Trans. Robotic and Automation* (submitted).
- Jacak, W. and S. Dreiseitl (1995). Genetic Algorithm based Neural Networks for Dynamical Systems Modeling, *Proc. of IEEE International Conference on Evolutionary Computing ICEC'95*, November IEEE Press, Perth, Australia, pp. 602–607.
- (1996). Hybrid Evolutionary Programming — the Tools for CAST, *Lecture Notes in Computer Science*, Springer Verlag, Berlin, No 1030, pp. 289–305.
- (1996). Robotic Agent Control Combining Reactive and Learning Capabilities, *IEEE International Conference on Neural Networks ICNN'96*, June Washington, USA.
- Jacak, W., S. Dreiseitl, T. Kubik, and D. Schlosser (1995c). Distributed Planning and Control of Intelligent Robot's Arm Motion based on Symbolic and Neural Processing, *IEEE International Conference on Systems, Man and Cybernetics SMC'95*, October, Vancouver, Canada, pp. 2898–2903.
- Jacak, W., T. Kubik, and R. Muszynski (1995d). Neural Processing based Robot Kinematics Modelling and Calibration for Pose Control *XII Conference on Systems Science*, Wroclaw, Poland, September pp. 288–295.
- Jacak, W., and et al. (1994b). The Hybrid Evolutionary Method and System for Intelligent Robot Control, Technical Report RWCP, Linz, Tskuba, Japan.
- (1995e). HYROB System: The module for automatic synthesis of the neural representation of a geometric knowledge and fast distance calculation, Technical Report RWCP, Linz, Tskuba, Japan.
- (1995f). HYROB System: The module for the automatic synthesis of neural models for robot direct and inverse kinematics, Technical Report RWCP, Linz, Tskuba, Japan.
- (1995g). Hybrid Programming Approach to Robot Dynamics, Technical Report RWCP, Linz, Tskuba, Japan.
- (1995h). Neural Network based gain scheduling control system for robots in the presence of uncertainties with one-step trajectory planning, Technical Report RWCP, Linz, Tskuba, Japan.

- (1996). HYROB System: Application to the design of intelligent robotic agents acting in a real world environment, Technical Report RWCP, Linz, Tskuba, Japan.
- Jahn, G. (1996). *Event Based Approach to Simulation Based Control of Flexible Manufacturing Systems*, Ph.D. Thesis of Johannes Kepler University Linz.
- Jahn, G. and P. Rogalinski (1993). Simulator of Robotic Workcell: Implementation, Technical Report University Linz, Austria.
- Jones, A. T. and C. R. McLean (1986). A Proposed Hierarchical Control Model for Automated Manufacturing Systems, *Journal of Manufacturing Systems*, **5**(1): 15–25.
- Josin, G., D. Charney, and D. White (1988). Robot control using neural networks, pp. 625–631, *International Conference on Neural Networks ICNN'88-2*.
- Kawato, M., Y. Uno, M. Isobe, and R. Suzuki (1987). A hierarchical model for voluntary movement and its application to robotic, *IEEE First International Conference on Neural Network*, Vol. 4., pp. 573–82.
- King, J. R. (1980). Machine-Component Grouping in Production Flow Analysis: An Approach Using a Rank Order Clustering Algorithm, *International Journal of Production Research*, **20**(2): 213–32.
- Kircanski, M. and O. Timcenko (1992). A geometric approach to manipulator path planning in 3D space in the presence of obstacles, *Robotica*, **10**(4): 321–328.
- Klein, C. A. and C. H. Huang (1983). Review of pseudoinverse control for use with kinematically redundant manipulators, *IEEE Trans. System, Man, Cybernetics*, SMC-13(3), 245–250.
- Kohonen, T. (1988). *Self-Organization and Associative Memory*, Springer Verlag, Berlin.
- Kosuge, K. and T. Oosumi (1996). Decentralized control of multiple robots handling an object, Proc. of IROS'96, Osaka, pp. 318–332.
- Koza, J. R. (1992). *Genetic Programming*, MIT Press.
- Kreinovich, V. Y. (1991). Arbitrary Nonlinearity is Sufficient to Represent All Functions by Neural Networks: A Theorem, *Neural Networks*, **4**: 381–383.
- Krogh, B. H. and Z. Banaszak (1989). Deadlock Avoidance in Pipeline Concurrent Processes, *Proc. of Workshop on Real-Time Programming IFAC/IFIP*.
- Kuc, R. and V. Viard (1991). A physically based navigation strategy for sonar guided vehicles, *International Journal Robotics Research*, **10**: 1–67.
- Kumar, V. R. and K. J. Waldron (1989). Actively Coordinated Vehicle Systems, *ASME Journal of Mech. Trans. and Aut. Design*, **III**(2): June.
- (1991). Force Distribution Algorithms for Multifingered Grippers, *Control and Dynamic Systems*, **39**: Academic Press Inc.
- Kung, S. Y. and J. N. Hwang (1989). Neural network architectures for robotics applications, *IEEE Trans. Robotics Automation*, RA-5(5), 641–657.
- Kusiak, A. (1990). *Intelligent Manufacturing Systems*. Prentice Hall.
- Lambrinos, D. and C. Scheier (1996). Building complete autonomous agent. A case study on categorization, Proc. of IROS'96, Osaka, pp. 170–179.
- Latombe, J. C. (1991). *Robot motion planning*, Kluwer Ac. Pub., Boston, Dordrecht.
- Lee, C. C. (1990). Fuzzy Logic in Control Systems: Fuzzy Logic Controller, *IEEE Trans. SMC*, **20**(2).
- Lee, S. and G. A. Bekey (1991). Applications of neural network to robotics, in *Control and Dynamic System: Advances in Robotic Systems*, (ed.) C. T. Leondes, Academic Press Inc., San Diego, New York, pp. 1–70.
- Lee, S. and J. M. Lee (1990). Multiple task point control of a redundant manipulator, *Proc. of IEEE International Conference on Robotics and Automation*, pp. 988–993.
- Lee, S. and R. M. Kil (1988). Multilayer feedforward potential function network, *IEEE International Conference on Neural Network*, Vol. 1., pp. 641–657.

- (1989). Bidirectional continuous associator based on gaussian potential function network, *IEEE International Joint Conference on Neural Network*, Vol. 1., pp. 45–53.
- (1990). Robot kinematic control based on bidirectional mapping neural network, *IEEE International Joint Conference on Neural Network*, Vol. 3, pp. 327–335.
- Lenz, J. E. (1989). *Flexible Manufacturing*. Marcel Dekker, Inc.
- Liegeois, A. (1977). Automatic supervisory control of the configuration and behavior of multibody mechanisms, *IEEE Trans. System, Man, Cybernetics*, SMC-7(12), 868–871.
- Lifschitz, V. (1987). On the semantics of STRIPS in *Reasoning about Actions and Plans*, M. Georgeff (ed.), San Francisco: Morgan Kaufmann.
- Lin, C. S., P. R. Chang, and J. Y. S. Luh (1983). Formulation and Optimization of Cubic Polynomial Joint Trajectories for Industrial Robots, *IEEE Trans. on Automatic Control*, AC-28(12): 1066–073.
- Lippman, R. P. (1987). An introduction to computing with neural nets, *IEEE ASSP Magazine*, 4: 4–22.
- Lozano-Perez, T. (1981). Automatic Planning of Manipulator Transfer Movements, *IEEE Trans. System, Man, Cybernetics*, SMC-11(10).
- (1986). Motion Planning for Simple Robot Manipulators, *Proc. of the III International Symposium on Robotics Research*, Faugeras O. D. and Giralt G.,.
- (1989). Task-Level Planning of Pick-and-Place Robot Motions, *IEEE Trans. on Computers* 38(3): 21–29.
- Luca, A. De, and G. Oriolo (1991). Issues in acceleration resolution of robot redundancy, *Proc. of Symp. on Robot Control, Syroco'91*, Vol. 2, pp. 665–670.
- Lumelsky, V. J. (1985). On fast computation of distance between line segments, *Inform. Process. Lett.*, 21: 55–61.
- Lumelsky, V. J. and K. Sun (1989). A unified methodology for motion planning with uncertainty for 2D and 3D two link robot arm manipulator *International Journal Robotics Research*, 9: pp 89–104.
- Maciejewski, A. A. and C. A. Klein (1985). Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments, *International Journal Robotics Res.* 4(3): 109–17.
- Maimon, O. (1987). Real-time Operational Control of Flexible Manufacturing System, *Journal of Manufacturing Systems*, 6(2): 125–136.
- Martinez, T. M. and K. J. Schulten (1990). Hierarchical neural net for learning control of a robot's arm and gripper, *IEEE Joint International Conference on Neural Network*, San Diego CA, Vol. 2, pp. 747–52.
- Mayer, W. (1986). Distances between boxes: Applications to collision detection and clipping, *Proc. of IEEE International Conference on Robotics and Automation*, pp. 597–602.
- Mayorga, R. V., A. K. C. Wong, and N. Milano (1992). A Fast Procedure for Manipulator Inverse Kinematics Evaluation and Pseudo-Inverse Robustness, *IEEE Tran. on Systems, Man, and Cybernetics*.
- McDermott, D. (1982). A temporal logic for reasoning about processes and plans, *Cognitive Science*, 6.
- McKerrow, P. J. (1991). *Introduction to Robotics*, Addison-Wesley Publ.
- Meystel, A. (1988). Intelligent control in robotics, *Journal Robotic Systems*, 5: 5.
- Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag.
- Miller, and W. T. et al. (1990). Real-Time Dynamic Control of an Industrial Manipulator Using a Neural-Network-Based Learning controller, *IEEE Trans. on RA*, 6(1): 1–9.
- Miyamoto, H., M. Kawato, and T. Setoyama (1988). Feedback error learning neural networks for trajectory control of a robotic manipulator, *Neural Networks*, 1: 251–265.

- Mooring, B. W., Z. S. Roth, and M. R. Driels (1991). *Fundamentals of Manipulator Calibration*, John Wiley and Sons.
- Nakamura, Y. and H. Hanafusa (1987). Optimal redundancy control of robot manipulators, *International Journal Robotics Res.* **6**(1): 32–42.
- Nilsson, N. J. (1980). *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA.
- Noble, B. (1976). *Methods for Computing the Moore–Penrose Generalized Inverse and Related matters*, (ed.) M. Z. Nashed, Academic Press.
- Park, J. and S. Lee (1990). Neural computation for collision-free path planning, *IEEE International Joint Conference on Neural Network*, Vol. 2, pp. 229–232.
- Parker, J. K. and D. E. Goldberg (1989). Inverse Kinematics of Redundant Robots using Genetic Algorithms, *Proc. of the IEEE International Conference on Robotics and Automation*.
- Pau, L. F. (1988). Sensor data fusion, *Journal Intelligent Robotic Systems*, **1**: 103–16.
- Paul, R. P. (1981). *Robot Manipulators: Mathematics, Programming and Control*. Cambridge, MIT Press.
- Pearl, J. (1984). *Heuristics*, Reading, MA: Addison-Wesley.
- (1988). *Probabilistic Reasoning in Intelligent Systems*, San Francisco, Morgan Kaufmann.
- Pichler, F. (1989). CAST Modelling Approaches in Engineering Design, *Lecture Notes in Computer Science* Vol. 410, pp. 52–68.
- Prasad, B., (ed.) (1989). *CAD/CAM Robotics and Factories of the Future*. Springer Verlag.
- Preparata, F. P. and M. I. Shamos (1985). *Computational Geometry: An Introduction*, Springer Verlag New York.
- Ranky, P. G. and C. Y. Ho (1985). *Robot Modeling. Control and Applications with Software*, Springer Verlag.
- Red, W. E. (1983). Minimum distances for robot task simulation, *Robotica*, **1**: 231–38.
- Rogalinski, P. (1994). An approach to automatic robots programming in the flexible manufacturing cell, *Robotica*, **11**.
- Rozenblit, J. W. and B. P. Zeigler (1988). Design and Modeling Concepts, in: *International Encyclopedia of Robotics, Applications and Automation*, (ed.) Dorf R., John Wiley and Sons, New York, pp. 308–22.
- Rumelhart, D. E., G. E. Hinton, and J. L. McClelland (1986a). A General Framework for Parallel Distributed Processing, *Parallel Distributed Processing*, MIT Press, Cambridge.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986b). Learning Internal Representations by Error Propagation, *Parallel Distributed Processing*, MIT Press, Cambridge.
- Sacerdot, E. D. (1981). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, **15**(2).
- Sanderson, A. C., L. S. Homem De Mello, and H. Zhang (1990). Assembly Sequence Planning, *AI Magazine*, **11**(1): Spring.
- Saridis, G. N. (1983). Intelligent robotic control, *IEEE Trans. Automatic Control* **AC-28**: 5.
- (1989). Analytical formulation of the principle of increasing precision with decreasing intelligence for intelligent machines, *Automatica*.
- Schwartz, J. T. (1981). Finding the minimum distance between two convex polygon, *Inform. Process. Lett.*, **13**: 168–170.
- Sciavicco, L. and B. Siciliano (1988). A solution algorithm to the inverse kinematic problem for redundant manipulators, *IEEE Trans. on Robotics and Automation*, RA-4(4), 403–410.
- Shin, K. and N. McKay (1985). Minimum Time Control of Robotic Manipulator with Geometric Path Constrains, *IEEE Trans. on Automatic Control*, **30**(6): 531–541.
- (1986). A Dynamic Programming Approach to Trajectory Planning of Robotic Manipulators, *IEEE Trans. on Automatic Control*, **31**(6): 491–500.
- Sierocki, I. (1996). An Automata Approach to Design of Feedback Control for an autonomous Agent, *Proc. of Intern. Conference EMCSR'96*, Vienna.

- Speed, R. (1987). Off-line Programming of Industrial Robots, *Proc. of ISIR 87*, 2110–123.
- Spong, M. and M. Vidyasagar (1989). *Robot Dynamics and Control*. Cambridge, MIT Press, Cambridge.
- Sugeno, M. (1985). *Industrial application of fuzzy logic*, North-Holland, Amsterdam.
- Thrun, S. (1994). A lifelong learning perspective for mobile robot control, *Proc. of IEEE Conference on Intelligent Robots and System IROS'94*, Munich, Germany.
- Thrun, S. and T. Mitchell (1993). Integrating inductive neural network learning and explanation-based learning, *Proc. of IJCAI'93*, Chambery, France.
- Tilli, T. (1992). *Fuzzy Logic*, Franzis Press, Munich.
- Valavanis, K. P. and H. M. Stellakis (1991). A general organizer model for robotic assemblies, *IEEE Trans. SMC* **21**(2).
- Vukobratovic, M. and M. Kircanski (1984). A Dynamic Approach to Nominal Trajectory Synthesis for Redundant Manipulator, *IEEE Trans. on System, Man, Cybernetics*, **SMC-14**(4).
- Vukobratovic, M. and N. Kircansky (1985). *Scientific Fundamentals of Robotics, Real-Time Dynamics of Manipulation Robots*. Springer Verlag.
- Wang, H. P. and J. Li (1991). *Computer Aided Process Planning*. Elsevier.
- Wang, L. and J. Mendel (1992). Generating Fuzzy Rules by Learning from Examples, *IEEE Trans. SMC*, **22**(6).
- Wang, Z., E. Nakano, and T. Matsukawa (1996). Realizing cooperative object manipulation using multiple behaviour based robots, *Proc. of IROS'96*, Osaka, pp. 310–317.
- White, H. (1990). Connectionist Nonparametric Regression: Multilayer Feedforward Networks Can Learn Arbitrary Mappings, *Neural Networks*, 535–549.
- Whitney, D. E. (1972). The Mathematics of Coordinated Control of Prosthetic Arms and Manipulators, *ASME International Journal of Dynamic Systems, Measurement, and Control*, **94**(12).
- William, H. (1990). *Numerical recipes in C*, Cambridge, Univ. Press.
- Wloka, D. W., ed. (1991). *Robot Simulation*, Springer Verlag, (in german).
- Wolter, J. D., R. A. Volz, and A. C. Woo (1985). Automatic Generation of Gripping Positions, *IEEE Trans. System, Man Cybernetics*, **SMC-15**(2): Mar.
- Wooldridge, M., J. P. Mueller, and M. Tambe (ed) (1996). *Intelligent Agents*, Lecture Notes in Computer Science 1037, Springer Verlag, Berlin, New York.
- Wu, C. H., J. Ho, and K. Y. Young (1988). Design of Robot Accuracy Compensator After Calibration, *Proc. of the IEEE International Conference on Robotics and Automation, Volume 1* Computer Society Press.
- Wu, C. M., B. C. Jiang, and C. H. Wu (1993). Using neural networks for robot positioning control, *Robotics and Computer Integrated Manufacturing*, **10**(3): 153–168.
- Yeung, D. T. and G. A. Bekey (1989). Using a context-sensitive learning network for robot arm control, *Proc. of IEEE International Conference on Robotics and Automation*, pp. 1441–1447.
- Yoshikawa, H. and T. Holden (ed.) (1990). *Intelligent CAD*. North Holland.
- Yoshikawa, T. (1985). Manipulability of robotics mechanisms, *International Journal Robotics Res.* **4**(2): 3–9.
- Zadeh, L. A. (1973). Outline of a new approach to the analysis complex system and decision processes, *IEEE Trans. SMC*, **3**.
- Zeigler, B. P. (1984). *Multifaceted Modeling and Discrete Event Simulation*, Academic Press.
- Ziegert, J. and P. Datsoris (1988). Basic Considerations for Robot Calibration, *Proc. of the IEEE International Conference on Robotics and Automation*, IEEE, New York.

Index

- Acceleration, trajectory planner, 127
- Acceptor, workcell state recognizer, 211–213
- Action interpreter, 225, 237–240
- Actions, *see also* Execution level
 - learning for prediction and coordination, 230–233
 - planning, 6, 14, 225; *see also* Planning cell actions
- Active learning, 203, 228
- Active mode
 - monitoring, 255–256
 - neural model of robot dynamics, 109–112
- Active model, workstation, 158–162
- Advance, sensor model, 181, 182
- Agent, event-based modeling and control of, 165–169
- Allocation relation, 35, 36
- Assembly tasks, 34, 35–40
- Attractive potential, fine motion planning, 129
- Automatic planning
 - action, 14
 - collision-free path planning, 147
 - task-level, 54
- Autonomous agents, 10, 15–17
 - classification of, 2
 - distributed control, 219–221
- Backpropagation network, 60, 115–117, 205
- Behavior of workcell, 17–19
- Bidirectional search, trajectory planning, 122–123, 127
- BRAEN, 40
- Calibrated kinematic model, 72–74
- Calibrated neural model, computed torque controller, 207
- Calibration
 - kinematics, measurement data interface, 150
 - neural model of robot dynamics, 109–118
 - neural network kinematic model, 67–72
- CAP/CAM systems, 102, 141–151
 - HyRob system structure and design process 148–151
 - intelligent design with ICARS, 143–148
 - structure of ICARS, 141–143
- CARC (computer-assisted robotic cell) control 11–12
- Cartesian space
 - collision-free robot search process, 97
 - discretization of, 56
 - kinematic matrix expressions, 66–67
 - path planning, 82–83
- Category neurons, 229–230
- Cell controller, 4, 12; *see also* Control system
- Cell design: *see* CAP/CAM systems: Workcells
- Cell level of control, 3, 12, 13
- Centralized agent, 15–17
- Centralized robotic system coordinator, 213–219
- Circular-wait-deadlock, 43, 44
- Classification of workcells, 1–2
- Collision-free condition, 215
- Collision-free path planning, 55–99, *see also* Path planner
 - Groplan, 147
 - optimal trajectory of motion, 119–123
- Collision-free path search, 77–78, 91–97
- Collision-free robot configurations, 93–97
- Collision relation, 214–215
- Common-sense rule, 115–116
- Communications networks, 3
- Computational geometry modeling methods, 26–32
- Computed torque controller, 205, 206–209
- Computer-assisted robotic cell (CARC) control task, 11–12
- Conceptual state space, 225, 227–230
- Concurrent process planning, 41
- Conditional path planning, 128–129
- Configuration feasibility testing, 78
- Constant-velocity submodes, 110

- Context-sensitive networks, 60
- Continuous track calculation, 98–99, 100
- Control files, object-oriented implementation of fuzzy organizer, 286–287
- Control system, 3, 4, 10, 12–15, 23; *see also*
 - Execution level; Organization level of system
- ICARS, 141–148, 149, 151
- machining task planning, 40
- multiagent system: *see* Coordination of multiagent system
- robotic workcell components, 9
- unit module, path planner in presence of unknown objects in environment, 188
- Control task, computer assisted robotic cell (CARC), 11–12
- Convergence, path planning, 85–86
- Coordination level of control, CARC, 7, 12, 13
- Coordination level of knowledge, 14
- Coordination of multiagent system, 211–240
 - acceptor, workcell state recognizer, 211–213
 - centralized, 213–219
 - distributed, 219–221
 - lifelong-learning-based, real-world systems, 221–240
 - action interpreter, 237–240
 - conceptual state space, 227–230
 - learning of agent actions for prediction and coordination, 230–233
 - Q-learning-based action planner, 233–237
 - structure, 223–227
- Coordinator, centralized, 15, 213–219
- COPLANNER, 40
- Cost
 - organization inputs, 242
 - robot motion and, 120
 - time-trajectory planning, 122
- Cubic spline interpolation, 127
- Deadlock, 43–46
 - path planner in presence of unknown objects in environment, 188
 - sensor model, 181, 182
- Decision making, organization level, 246–253
- Decision system, 171, 188
 - path planner in presence of unknown objects in environment, 188
 - sensor model, 181–182
- Decomposition approach, assembly-disassembly sequences, 39–40
- Decoupled network topology network, 62
- Deforming condition, grasp, 138
- Denavit-Hartenberg model, 72, 104, 149
- Design systems: *see* CAP/CAM systems
- Detour, sensor model, 181
- DFA, 40
- Diagnostic system, 18
- Direct kinematics
 - sinusoidal neural network, 64–67
 - training module, 150–151
- Disassembly sequences, 39
- Discrete event system (DEVS), 18–19; *see also*
 - Event-based modeling; Object-oriented discrete-event simulator
- control system creation, 155–157
- distributed control, 219–221
- event-based modeling and control, 157–164
- monitoring and updating, 256–259
- Discrete model of robot kinematics, 57–58, 127
- Discretization of space, 56–57
- Distance computing problem, 27–31
- Distributed controls, 16–17
- Distributed coordination of multiagent system, 219–221
- Duration of movement, 175, 192
- Dynamic control, 15, 17
 - machining task planning, 40
 - optimal trajectory of motion, 119, 120, 121
 - time-trajectory planning, 121–122, 126; *see also* Time-trajectory planner
- Dynamic linearization module, neural network-based executor, 203–205
- Encoder resolution, neural network kinematic model calibration, 69
- Equilibrating forces, object-gripper system, 135–136
- Euler-Lagrange equation, 101, 103–107
- Evaluation function, time-trajectory planning, 123
- Event-based modeling
 - agent, 165–169
 - motion planning, 176–182
 - production store, 164–165
 - workstation, 157–164
- Event coordination, 14
- Excluded surfaces, 134
- Execution level, 7
 - agent action in presence of uncertainty, 203–209
- CARC, 12, 13
 - event-based modeling and control of agent, 165–169
 - event-based modeling and control of workstation, 157–164
 - event-based modeling of production store, 164–165

Execution level (*cont.*)

- lifelong learning-based coordinator, 227
 - neural and fuzzy computation-based agents, 169–209
 - intelligent and reactive behavior in presence of uncertainty, 172–176
 - multisensor image processing-based world modeling and decision-making systems, 176–182
 - on-line geometric path planner in presence of unknown object, 182–189
 - on-line time trajectory planner, 189–203
 - reactive executor of agent action in presence of uncertainty, 203–209
 - structure of workcell, 170
- Execution level of knowledge, 14
- Extended model, neural kinematic, 70
- External selection rule
 - coordination of nonautonomous actions, 217–218
 - distributed control, 220–221

Facility level of flexible manufacturing systems, 3

- Feedback network, update rule, 89, 90
- Feedforward network topology, 60
- Final state machine model of robot kinematics, 74–77
- Fine motion planner, 128–131
- Finger sensors, 140
- Finite state machine model of robot kinematics, 74–77
- First free buffer strategy, 242
- Flexible manufacturing systems (FMS), 2–3, 9–12
- Flow time, 41
- Force, grasp, 137–138
- Force factor, gripper, 138–139
- Forward kinematics, 60–63
- Frames Table, 55, 157
- Friction condition, grasp, 137
- Fundamental plan
 - lifelong learning-based coordinator, 224
 - workcell action, 41
- fuzzyART and fuzzyARTMAP algorithm, 228
- Fuzzy computation-based agents, 169–209
 - intelligent and reactive behavior in presence of uncertainty, 172–176
 - multisensor image processing-based world modeling and decision-making systems, 176–182
 - on-line geometric path planner in presence of unknown object, 182–189

Fuzzy computation-based agents (*cont.*)

- on-line time trajectory planner, 189–203
 - reactive executor of agent action in presence of uncertainty, 203–209
- Fuzzy organization level, 7
- Fuzzy organizer, object-oriented simulation, 285–293
- Fuzzy reasoning, organization level, 242–246
- Fuzzy tuner, time–trajectory planner, 195–203
- Gain, fuzzy decision making, 251, 252, 253
- Gear behavior, 69
- Generalization algorithm, neural implementation of, 228–229
- Geometrical model, 24–26, 27, 32
- Geometric control, 15
 - machining task planning, 40
 - time–trajectory planning, 126
- Geometric path planning
 - conditional, 128–129
 - in presence of unknown object, 182–189
- Global methodology of obstacle avoidance, 56
- Global state of workcell statistics, 245
- Global updating, 257–258
- Gradient algorithm, 85–86, 94, 95
- Graphics, GRIM module, 141, 142; *see also* CAP/CAM systems
- Graphics modeling, 26
- GRASP, 40
- Grasping/gripper, 131–140
 - grasp learning, 132–133
 - optimal grasping forces, 133–140
 - selection of mating surfaces, 133–135
 - stability of object in, 135–136
 - symbolic computation-based, 132–133
- Grim module, 141, 142, 143, 144
- GROPLAN, 142, 143, 145–148
- Heuristic function
 - monotone restriction and, 80–82
 - time–trajectory planning, 123–126
- Hierarchical architecture of flexible manufacturing systems, 3, 4
- Hierarchical control, 9, 10, 12–15
- Hopfield-type networks, 60
- HyRob system structure and design process, 148–151
- ICARS
 - HyRob system, 149, 151
 - intelligent design with, 143–148
 - structure of, 141–143
- If–then rules, 195–203, 246

- Inertia submode, 110
- Intelligent controller, 171
- Intelligent control of autonomous robotic system (ICARS): *see* ICARS
- Intelligent and reactive behavior in presence of uncertainty, 172–176
- Intelligent robotic workcell, 2–7, 9–12
 - centralization versus autonomy, 15–17
 - components and definitions, 9–12
 - distributed control of, 16–17
 - hierarchical control, 12–15
 - structure and behavior of, 17–19
 - workcell, 9–12
- Interaction forces, object-gripper system, 135, 136–137
- Internal selection rule, 214–216
- Intersection detection problem, 31–32
- Inverse dynamics, neural network-based executor, 206–209
- Inverse kinematics, 60–63, 82–90; *see also* Neural networks
 - HyRob system, 151
 - with on-line obstacle avoidance, 185–186
 - path planner, 127
 - in Cartesian space, redundant manipulators, 86–92
 - in presence of unknown objects in environment, 188–189
 - sensor data combination algorithm, 178–179
- Jacobian
 - calibrated kinematic model application to, 72–74, 75
 - inverse kinematics builder HyRob system, 151
 - path planning in Cartesian space, 83
 - sigmoid neural network-based training, 63, 64–67
- Jerk minimization, 193, 199, 200, 201, 202, 203
- Job, defined, 42
- Joint motion
 - execution level of system, 175
 - limits, computation of, 191–192
- Joint space discretization, 56
 - collision-free robot search process, 96
 - path planner structure, 80–81
- Joint space position calculation, 174–175
- Joint velocity, 82–83
- Just-in-time strategy, 242, 244
 - fuzzy organizer, 288, 290
 - rule base, 247
- Kinematic models, 57–58
 - neural network, *see also* Neural networks
 - calibration of, 67–72
 - symbolic computation-based neural model of, 67–69, 149
 - successor set generation, 92
 - synthesizer, 150
 - training module, 150–151
- Knowledge base, 10, 14
- Knowledge bottlenecks, 223
- Layout modeling, 26, 32, 144
- Learning, *see also* Neural networks
 - fuzzy decision making, 249
 - neural model of robot dynamics, 109–118
- LEGA, 40
- Lifelong-learning-based systems, real-world, 221–240
 - action interpreter, 237–240
 - conceptual state space, 227–230
 - learning of agent actions for prediction and coordination, 230–233
 - Q-learning-based systems, action planner, 233–237
 - structure, 223–227
- Linear approximation method, sensor model, 178, 179, 180
- Link parameter errors, 69
- Local area networks, 3
- Local methodology of obstacle avoidance, 56
- Local updating, 258
- Logical control, 14, 40
- Logical model, 24, 32
- Lyapunov function, 88, 186–187
- Machine model, final state, 74–77
- Machine selection, 4, 11
- Machining
 - assembly task specification, 37
 - control systems, 4
 - defined, 33
 - flexible manufacturing system architecture, 3
 - operations design, 40
 - planning cell actions, 34–35, 40–43
- Manipulator models
 - position error reduction, 72
 - sensor data combination, 178
- Material handling system, 9
- Material selection, 4
- Materials flow, 5
- Mathematica, 149
- Maximum force condition, grasp, 137–138

- Maximum processing time, 289
- Maximum wait time strategy, 226, 247, 289
- Measurement data interface, robot kinematics trainer, 150
- Minimal energy method, 130–131
- Minimum processing time strategy, 289
- Minimum setup time strategy, 242, 244, 247, 288
- Minimum transfer cost strategy, 242
- Model-based approach, robot group control, 17
- Modeler, lifelong learning-based coordinator, 225
- Modeling of workcell, 4; *see also* Virtual robotic cells
- Monitoring, 4, 7, 255–261
 - object-oriented simulator, 268
 - on-line, 18
 - prediagnosis, 256–261
 - tracing active state of system, 255–256
- Monotone restriction, 80–82
- Motion controller, 4, 192
 - execution level, 172–175
 - neural network dynamics, passively acquired, 203–205
- Motion planning, 5, 6, 7, 19
 - control system problems, 11
 - GROPLAN, 142, 143
 - object-oriented simulator, 264
 - off-line: *see* Robot motion, off-line planning
- Motion track, search technique for, 78–80
- Multiagent system coordination, 16; *see also* Coordination of multiagent system
- Multilayer network topology, 60
- Multisensor systems
 - data combination, path planning, 188–189
 - neural and fuzzy computation-based, 176–182
- Network, 10, 23
- Neural gradient algorithm, 89, 90
- Neural models of robot kinematics, 57–58
- Neural networks
 - execution, 169–209
 - intelligent and reactive behavior in presence of uncertainty, 172–176
 - multisensor image processing-based world modeling and decision-making systems, 176–182
 - on-line geometric path planner in presence of unknown object, 182–189
 - on-line time trajectory planner, 189–203
 - reactive executor of agent action in presence of uncertainty, 203–209
 - HyRob system, 151
 - kinematic model calibration, 67–72
 - Neural networks (*cont.*)
 - kinematics output, 150
 - path planning, conditional, 128–129
 - path planning in Cartesian space, 82–99
 - continuous track calculation, 98–99
 - inverse model of robot kinematics, 82–90
 - search for collision free path, 91–97
 - state transition of system, 91
 - path planning in joint space, 58–82
 - discrete model of robot kinematics, 58–74
 - finite state machine model of robot kinematics, 74–77
 - search strategies for collision-free robot movements, 77–82
 - path planning in presence of unknown objects in environment, 182–189
 - inverse kinematics with on-line obstacle avoidance, 185–186
 - Lyapunov function, method based on, 186–187
 - one-step path planning based on multisensor data combination, 188–189
 - steepest descent method, 187
 - virtual points, 184
 - tuning, backpropagation algorithm, 115–117
- Newton equations, 101
- Nominal kinematics, 69
- Non-autonomous agents, 1, 10
- Null rule, 116–117
- Object modeling, 25
 - Grim module, 144
 - virtual cell modeling, 25
- Object-oriented discrete-event simulator, 261–293
 - fuzzy organizer implementation, 285–293
 - object classes, 289–290
 - optimal values, 290–293
 - organizer, 285–289
 - object classes, 269–285, 289–290
 - buffer, 270, 271
 - device, 273
 - equipment, 270, 271, 272–273
 - event, 283–284
 - InputConveyer, 276–277
 - operation, 280–281
 - OutputConveyer, 277–278
 - part, 282–283
 - Robot, 278–279, 280
 - store, 275–276
 - task, 281–282
 - workstation, 274, 275
 - specification, 262–269

- Objects/obstacles, environmental
 - neural network-based path planner, 182–189
 - sensor model, 181–182
- Obstacle avoidance, 56
- Off-line planning, 171; *see also* Robot motion, off-line planning
- Off-line simulation, 17
- One-step path planning based on multisensor data combination, 188–189
- One-step trajectory planner, 190–195
- On-line obstacle avoidance, 185–186
- On-line planning
 - geometric path planner in presence of unknown object, 182–189
 - time trajectory planner, 189–203
- Operating plan, 4, 5
- Operational control, 14
- Optimal grasping forces, 133–140
- Optimization, time–trajectory planning, 122
- Organization level of knowledge, 14
- Organization level of system, 7, 241–253
 - control, 12, 13
 - fuzzy reasoning, 242–246
 - rule base and decision making, 246–253
 - task of organizer, 241–242
- Organizer input fuzzification, 245
- Output function g modeling, 59–60
- Overshoot minimization, 194, 199, 203

- Parameter tuning, 113–114
- Partial derivative computation, 102
- Partially autonomous agents. 1, 10
- Passive learning
 - computed torque controller, 207, 208, 209
 - generalization algorithm implementation, 228
 - neural network-based executor, 203–205
- Passive mode, neural model of robot dynamics, 109, 112–113
- Passive model, workstation, 163–164
- Passive state register of robot model, 168–169
- Path planner, 127, 147
 - execution level of system, 171
 - motion controller, 172–173
 - segment planning step, 173–175
 - of manipulator, 55–99
 - neural and discrete models of robot kinematics, 57–58
 - neural network-based planning in Cartesian space, 82–99; *see also* Neural networks
 - neural network-based planning in joint space, 58–82; *see also* Neural networks
 - optimal trajectory of motion, 119–123
- Path planner (*cont.*)
 - neural network and fuzzy computation-based in presence of unknown object, 182–189
 - inverse kinematics with on-line obstacle avoidance, 185–186
 - Lyapunov function, method based on, 186–187
 - one-step path planning based on multisensor data combination, 188–189
 - steepest descent method, 187
 - virtual points, 184
- Pause, sensor model. 181, 182
- Penalty function, 80, 93
- Performance function, 94
- Planning, 4, 17, 18; *see also* Execution level; Path planners; Robot motion, off-line planning
- Planning cell actions, 33–54
 - methods for, 38–43
 - assembly task, 38–40
 - machining, 40–43
 - on-line: *see* On-line planning
 - production routes, fundamental plans of action, 43–54
 - process route, algorithm for, 48–50
 - process route interpreter, 50–54
 - route planning, quality criterion, 43–48
 - task specification, 33–38
 - assembly, 35–37
 - machining, 34–35
- Polyoptimization problem, 194
- Potential field method, 129–130
- Precedence relation, 34, 35, 36
- Predagnosis, 256–261
- Priority-based start-stop synchronization, 215, 216
- Process definition, 3
- Processing steps, 4
- Processing time, 41, 289
- Process planner, 19, 141, 142, 143, 144
- Process route, 40
 - algorithm for, 48–50
 - interpreter, 50–54
- Process sequencing, 4
- Product flow simulation, 5
- Production route planning, 43–54
 - machining task, 42
 - process route, algorithm for, 48–50
 - process route interpreter, 50–54
 - quality criterion, 43–48
- Production store, event-based modeling of, 164–165

- Production system, robotic workcell components, 9
- Production time, organization inputs, 242
- Productivity, organization inputs, 242
- Programming, 5, 14
- Projection method, sensor model, 179, 180
- Proportional excitement rule, 116
- Pseudoinertia matrices, 104, 105, 106
- PUMA-like robot
 - one-step trajectory planning, 197
 - robot dynamics modeling, 102–107
- Push strategy, 242, 244
 - object-oriented implementation of fuzzy organizer, 288
 - rule base, 247
- Q-learning-based systems, action planner, 233–237
- Quasistatic submode, 110
- Ranging, sonar, 177–178
- Reactive behavior in presence of uncertainty, 172–176, 203–209
- Real cell, 23
- Real-time monitoring: *see* Monitoring
- Recall mode, 230
- Recurrent loop, 29
- Redundant manipulators, path planning in Cartesian space, 86–92
- Registration, workstation model, 163–164
- Resource allocation policy, route planning, 43–46
- Resource management, 4
- Resources, object-oriented simulator, 262–264
- Reward function, Q-learning, 234
- Robot group cooperation, 16–17
- Robotic workcells: *see* Workcells
- Robot motion, off-line planning, 55–140
 - collision-free path planning of manipulator, 55–99
 - neural and discrete models of robot kinematics, 57–58
 - neural network-based planning in Cartesian space, 82–99; *see also* Neural networks
 - neural network-based planning in joint space, 58–82; *see also* Neural networks
 - fine motion, 128–131
 - grasping, 131–140
 - time trajectory planner, 99–126, 127
 - modeling of robot dynamics, 99, 101–107
 - neural network-computed dynamics for, 121–126, 127
- Robot motion, off-line planning (*cont.*)
 - time trajectory planner (*cont.*)
 - optimal trajectory planning problem, 118–121
 - symbolic and neural network-computed robot dynamics, 107–117
- Route planning
 - production: *see* Production route planning
 - Taskplan, 145
- RPY angles, 73
- RPY coordinates, 66–67, 74
- Rule base, organization level, 246–253
- Scheduling, 4
- Scheduling fuzzification, 244
- Search strategies, 91–97
 - collision-free robot movements, 77–82
 - for collision-free robot movements, 77–78
 - time-trajectory planning, 122–123
- Search technique for motion track, 78–80
- Selection rules, coordination of
 - nonautonomous actions, 214–218
- Sensor data combination, 178–181
- Sensors
 - conceptual states, 232, 233
 - grasp planning, 140
 - multisensor image processing-based world modeling and decision-making systems, neural and fuzzy computation-based, 176–182
 - one-step path planning based on multisensor data combination, 188–189
- Shortest processing time strategy, 289
- Sigmoidal neural network approach, 60–63
- Similarity radius, 228
- Simple model, neural kinematic, 70
- Simulation, *see also* Virtual robotic cells
 - fuzzy decision making, 249, 250
 - GRIM, 141, 142, 143, 144
 - object-oriented: *see* Object-oriented discrete-event simulator
 - off-line, 17
 - on-line, 18
- Sinusoidal function, 108
- Sinusoidal neural network, direct kinematic modeling, 64–67
- Sonar ranging, 177–178
- Stage function, machining process, 42
- Start-stop synchronization, priority-based, 215, 216
- State transition function, 91
- Statistics, object-oriented simulator, 268–269

- Status functions, machining process, 42
- Steepest descent method, 187
- Successor set generation, 92
- Surfaces, unreachable, 134
- Symbolic calculation-based grasp learning, 132–133
- Symbolic calculation-based grasp planning, 140
- Symbolic calculation-based kinematics, 67–69, 149
- Synchronization, workstation model, 164

- Task fuzzification, 243
- Task management, 3, 4
- Taskplan, 144
 - ICARS, 141, 142
 - route planning, 145
- Task planning, 5
- Tasks
 - assembly and machining, 33–38
 - object-oriented simulator, 265–267
 - time-trajectory planning, 126
- Technological operations, machining task specification, 34
- Technological task, 23
- Technological task fuzzification, 243
- Thermal effects, neural network kinematic model calibration, 69
- Time planning
 - Groplan, 147–148
 - organization inputs, 242
- Time-trajectory planner, 99–126, 127
 - Groplan, 147–148
 - modeling of robot dynamics, 99, 101–107
 - neural network and fuzzy logic-based, 189–203
 - fuzzy tuner, 195–203
 - one-step trajectory planner, 190–195
 - neural network-computed dynamics for, 121–126, 127
 - on-line, 189–203
 - optimal trajectory planning problem, 118–121
 - symbolic and neural network-computed robot dynamics, 107–117
- Tooling, machining task planning, 40
- Tool selection, 4
- Torque controller, 205, 206–209
- Tracking, sensor model, 181
- Tractability bottlenecks, 223
- Training mode, 230
- Training module, direct kinematics, 150–151
- Training pattern preparation, grasping forces, 139–140

- Trajectory planning, 127, 192–193; *see also*
 - Time-trajectory planner
 - execution level of system, 171, 175–176
 - one-step, path planner connection, 189
 - optimal motion, 119–123
- Tuning
 - fuzzy organizer optimization, 290
 - neural network-based executor, 205
- Two-dimensional manipulator, 60–63

- Uncertainty
 - reactive behavior in presence of, 172–176
 - reactive executor of agent action in presence of, 203–209
- Unknown objects, neural network-based path planner, 182–189
- Unreachable surfaces, 134
- Update ratio, 30
- Update rule for feedback network, 89, 90
- Updating, monitoring, 257–258

- Value function, Q-learning, 234
- Velocities, trajectory planner, 127
- Virtual points, path planning, 184
- Virtual robotic cells, 4, 23–32
 - computational geometry methods, 26–32
 - defined, 23
 - distance computing problem, 27–31
 - geometrical model, 24–26, 27
 - intersection detection problem, 31–32
 - layout modeling, 26
 - logical model, 24
 - object modeling, 25
- Virtual workcell: *see* CAP/CAM systems

- Wait function, 42, 43–46
- Waiting time, 41, 242
- Workcells, 1–7, 9–12; *see also* Intelligent robotic workcell
 - object modeling, 25
 - virtual: *see* CAP/CAM systems; Virtual robotic cells
- Work-in-process, 268, 285, 290
- Workspace point selection, 174
- Workstation, 3, 4, 23–24
 - event-based modeling and control of, 157–164
 - interaction with agent, modeling, 160–162

- XAP, 40

- Zero-reference position model, 72
- Zones, route planning, 43