Application Oriented Programming and Control of Industrial Robots

Klas Nilsson

Second Edition

LUTFD2-TFRT--3212

Department of Automatic Control, Lund Institute of Technology

Application Oriented Programming and Control of Industrial Robots

Klas Nilsson

To Rosel, Sofie, and Adam

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 LUND
Sweden

© 1992 by Klas Nilsson. All rights reserved
Published 1992
Printed in Sweden

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
LICENTIATE THESIS

Date of issue
July 1992

Document Number
ISRN LUTFD2/TFRT--3212--SE

Author(s)
Klas Nilsson

Supervisor
Lars Nielsen, Karl Johan Åström

Sponsoring organisation
Swedish National Board for Technical Development (NUTEK), contract 88-01777P.

Title and subtitle

Application Oriented Programming and Control of Industrial Robots

Abstract

Efficient use of industrial robots requires a strong interplay between user level commands, the motion control system, and external equipment. It should also be possible for an experienced application engineer to tailor the motion control to a specific application in a convenient way, instead of deficient utilization of the device or tricky user programming which is often the case today. A layered software architecture has been designed based on an application oriented view, considering typical hardware and software constraints. The top layers of the architecture support improved integration of off-line programming with interactive teach-in programming. The proposed solution is based on a transformation of robot programs between an on-line and an off-line representation. A central part of the architecture is an intermediate software layer, allowing the experienced user to introduce application specific motion primitives, on top of the motion control system. Flexibility during system configuration combined with computing efficiency and performance at run-time is of major importance. The solution is based on so called actions, which are methods to be passed between different software layers. Such methods can be specifications of nonlinear control parameters, application specific control strategies, or treatment of external sensor signals. The actions can be implemented efficiently even in the multiprocessor case by using relocatable executable pieces of code generated from a special cross-compilation strategy. The lowest layers, comprising the motion control, have to be efficient and still fit in with the upper layers. In these layers, software solutions include an external sensor interface and a concept of motion pipelining allowing sensor based motions to be partly computed in advance. An experimental platform, built around commercially available robots, has been developed to verify the proposed solutions.

Key words
Industrial robots, Robot applications, Robot control, Robot programming, Open systems, Software architecture, Real-time systems, Motion strategies, Embedded control.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
147

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund

Contents

Acknowledgements

This work was carried out at the Department of Automatic Control in Lund, although most of the problems tackled originate from experience gained in my former work at ABB Robotics. I would like to express my gratitude to my supervisor Lars Nielsen. His understanding of practical and academical relevance for different problems and solutions has been most valuable. Lars also provided guidance and support during the work, and it has been a great pleasure to work with him. I would also like to thank Anders Blomdell and Leif Andersson for valuable discussions about real-time systems, for powerful real-time software components, and for maintaining an excellent host computer environment at the department. Thanks also to Rolf Braun who built the special hardware in the experimental robot control systems. I am grateful to Ola Dahl for the collaboration in robotics research, and to all my colleagues at the department for the nice and friendly atmosphere. The contacts and good relations with people at the Department of Production and Materials Engineering, and at the Department of Industrial Electrical Engineering and Industrial Automation, are also appreciated.

The Swedish National Board for Technical Development is acknowledged for the financial support, and the decision of ABB robotics to put a modern robot in our lab is also appreciated. Professor Karl J. Åström is gratefully acknowledged for creating and maintaining an excellent research environment at the department, and for comments on my work. Thanks also to Björn Wittenmark, Sven-Erik Mattson, Per Persson, Leif Andersson, and Dag Brück for valuable criticism on the manuscript, and to any colleague that has commented on the developed principles. Finally, I thank my wife Rosel, for her understanding, support and patience.

1

Introduction

Industrial robots are important and frequently used components in industrial production. Robots are distinguished from fixed automation mainly on the basis of their programmability and ability to be adaptable to different tasks. This implies that software issues for the control system are central for the applicability of robots. There is also a desire to handle more complex situations since it is likely that future applications will demand even more flexible systems. Apart from flexibility there is also a strong demand for efficiency since performance of the robot system is often related to productivity.

This thesis takes a problem oriented approach to the software issues for robot control. It starts with a discussion of real industrial problems. Solutions to these problems are the major topics, but the problem formulations are in some cases contributions in themselves. There is also emphasis on a software architecture, called the Open Robot Control (ORC) architecture.

The physical characteristics of industrial manipulators, which are rather precise but not perfect, influence many of the design choices If industrial robots were almost perfect, like NC machines, a fixed servo system with a robot independent motion description system or planning system on top of it would suffice. Such a system structure is, however, currently used for industrial robots. This has some drawbacks that will

be discussed in the thesis. The other extreme case is autonomous mobile robots dealing with a very uncertain environment, e.g. in space applications. Such robots must be careful and therefore also slow. They rely heavily on external sensors and maintenance of a world-model data base. The software architectures are designed to support high level (often AI related) software concepts, and with no special coupling to the low level control. Industrial robots, however, typically operate in a well known, but not completely known, environment. External sensors and internal control signals reflect external states that often need to be known at both high and low levels of the control system. A typical situation is when a robot is used for welding or grinding. Software for industrial manipulators must therefore provide a strong interplay between user level commands, sensor signals, and low level control. This interplay is crucial to obtain flexibility and performance, but also to avoid the cost of otherwise necessary external sensors. The software architecture suggested in the thesis pays careful attention to these issues.

An intermediate software layer between the standard motion control system and the user programming environment is suggested [38], instead of trying to obtain a general purpose motion control that can suit all applications. It should then be possible for an experienced application engineer to tailor the motion control to a specific application in a convenient way, instead of deficient utilization of the device or tricky user programming, which is often the case today. The underlying motion control system has also been given a structure to aid in the implementation of state of the art control algorithms for use in an industrial context. These lower layers of the system then have to fit with higher level software for programming and execution of user programs.

The user programming, i.e. the type of robot programming that is possible in current systems, can be carried out either on-line using the physical robot, or off-line without use of the robot. It turns out that these two approaches impose contradictory demands on the system. This is perhaps the reason why existing systems primarily support only one of these approaches. A common framework that merges these approaches is described. The main idea is to find a representation for robot programs that allows transformations between the different approaches. Both the representation and the transformation is treated.

Outline of the thesis

The thesis is organized as follows. Programming of application features for welding or grinding are good examples where the interplay between different software levels is crucial to obtain performance. This is discussed in Chapter 2 which also contains some additional examples. One may note that even a very mature application like spot welding needs special care to obtain the best possible performance in the short moves between spots. The need is even greater in more complicated sensor based applications.

Chapter 3 gives a first coarse partitioning in software levels. They also serve as a coarse design of the ORC architecture. The software in a robot control system should contain an abstraction level with high-level concepts for simple reprogramming, but also a level that allows efficient computation of the low-level control. These two levels are traditional. The interplay between user level programming and low level control, e.g. when solving the application examples in Chapter 2, naturally implies the need for an intermediate software layer for application specific motion control.

Chapter 4 treats the intermediate level and the solutions to the application examples in Chapter 2. It is shown that this level makes it simpler to deal with dynamics, constraints, and external feedback loops. The robot functions are favorably hand crafted considering the constraints of the specific problem. The robot functions are thus based on the skill of the human operator and his knowledge and experience in robot applications. A novel key idea in the proposed solution is to introduce a concept of method passing. This can be implemented in an ordinary compiled language and executed in a multiprocessor control system.

Chapter 5 treats the basic motion control level. Internal layers are proposed and the interface to the application level in Chapter 4 is treated. Special care is given the problem of combining motion commands with an interface for external sensors. A solution to the problem of using the basic control level for off-line programming purposes (see below) is proposed. It is also demonstrated how a control algorithm can be structured and implemented.

Chapter 6 serves as background material, introducing currently used concepts for robot programming. Programming without use of the physical robot, so called off-line programming, is often desirable, and some-

times necessary. Available off-line programming systems simplifies this type of programming. When the robot and its surrounding equipment are used for the programming, so called on-line programming, much of the abstract models are replaced by the physical robot and its environment. A proper programming environment can then make the programming more concrete, and the programming can be done by the production engineer with no or little experience in computer programming. These two concepts, unsufficiently integrated today, are separately described.

Chapter 7 presents a revised view of the robot programming problem. A new way to combine on-line and off-line programming is proposed. The proposed solution differs from today's integration of teach-in procedures [9], which is a way of using teach-in data in off-line programs. Instead, different special purpose programming tools should be used for on-line and off-line programming. This implies that the program needs to be transformed when transferred from the off-line to the on-line programming environment or vice versa. The programming style and the world modeling are primary aspects considered in the proposed solution.

The problems formulated in the thesis, and the solution developed, can not be evaluated by theoretical analysis. An experimental platform built around reconfigured commercially available industrial robots has therefore been developed. Chapter 8 describes how the original robot control systems have been changed to allow full control and programming in an external computer. Chapter 9 presents the complete experimental platform including the computers, their connections, and their use. Debugging of robot systems is a special issue. A robot program may fail due to many reasons, logical errors, bad tuning, broken sensors, unforeseen end-effector forces, etc. Debugging benefits from a special blend of conventional debugging tools for software, and interactive evaluation of signals and their dynamic properties in the system. Software components have been developed which allow connection of host computer software to the robot controller for such analysis.

The architecture, which was first briefly described in Chapter 3 and refined in Chapters 4, 5, and 7, is summarized and discussed in Chapter 10. Conclusions are given in Chapter 11.

2

Application Examples

This chapter gives some examples of applications that are nontrivial to handle with current systems. Such application examples are today, in most cases, handled by modifying the basic motion control of the robot. This can normally only be done by the robot manufacturer, and requires a substantial engineering effort. It seems that the interesting research field of application oriented robot programming and control, as defined in the thesis, has been overlooked within robotics research and development, and the formulations are thus believed to be contributions in themselves.

The approach taken here is that robot control systems should be open for the experienced user on a fairly low level tightly connected with the motion control system. The application problems in this chapter can be better solved in such an open system if it contains an intermediate level for application programming of robot motion control. The first example, which is deburring of castings, will be used extensively in the thesis to explain the design and the implementation of the system. The second example which treats a very mature application, spot-welding, shows that high performance motion can be encapsulated in the intermediate software layer. The third example on arc welding treats software aspects of the path tracking problem. There are then two examples in the section about assembly on how the application layer can improve

the performance of assembly operations without additional sensors or hardware. The first case deals with a single assembly cycle, while the objective in the second case is to improve the performance over longer time periods. A concept of method passing will be used later in the thesis to separate the application specific motion control from the standard motion control. The final application example on materials handling also illustrates how that concept can be used also on the user programming level to add generality of application features.

Additional sensors are sometimes necessary, but they have drawbacks since they cost, fail, complicate the installation, etc. A basic idea in most of the examples is to have a system that makes it possible and convenient to use information already existing in the system, which however is not possible in today's commercially available motion control systems. This can often eliminate the need for additional sensors, i.e. a variable in the software comprises the sensor signal. A more detailed illustration on information already existing in internal signals in the servo system will be given in Section 3.4.

A note on robot programs

The pieces of robot program code appearing in this chapter are supposed to be written by an ordinary robot programmer. The code is then executed in the robot controller, typically by an interpreter. The requirements on the compiled procedure called by the interpreter is the topic in the examples. Computations are programmed in a Pascal-like syntax, but motions are requested with MOVE statements. Rather than having a procedure MOVE with formal parameters, MOVE (and other types of move instructions) is a reserved identifier and parameters are specified with predefined attributes belonging to the MOVE instruction. Example (identifiers written with capitals are reserved names):

```
MOVE grinder TO right_edge
    WITH SPEED=0.15*mps
    WITH FORCE=MyForce1
```

Thus, programs for simple tasks with no or little computing involved are quite readable, also for the user with limited experience from computer programming. The syntax of the language used is of minor importance in the thesis. A syntax similar to the most common robot programming languages [9] is therefore used.

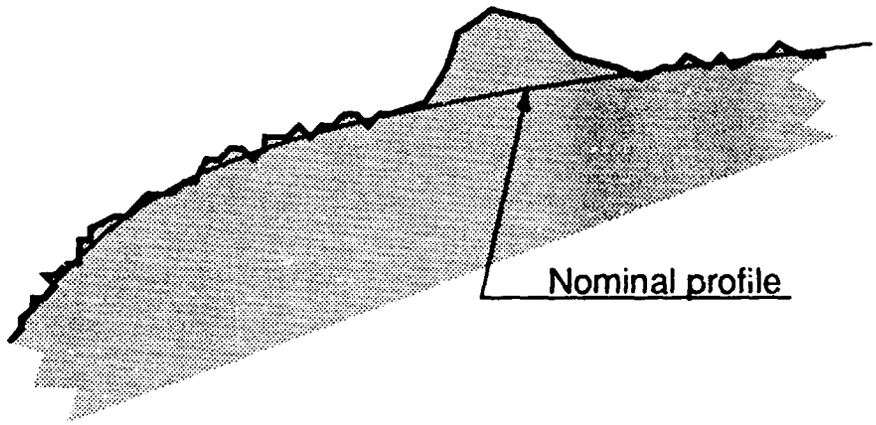

Figure 2.1 The profile of an edge on a casting.

2.1 Deburring of castings

Deburring of castings is a task that is preferably performed by industrial robots. Specific problems are to make the robot recognize where additional grinding is required and if so, to program suitable deburring strategies. This example illustrates the need for convenient possibilities to program control strategies on the intermediate level of the system. Consider an edge of a casting according to Figure 2.1. Deburring will normally be accomplished by moving the grinding tool with position control along a nominal path, and with force control in a direction normal to the path. The tool will make the surface smooth (with proper tuning of speed etc.), but exceptional places with much material may remain. The result after deburring will then be as in Figure 2.2. Computation of additional grinding motions, like the ones a human worker would have

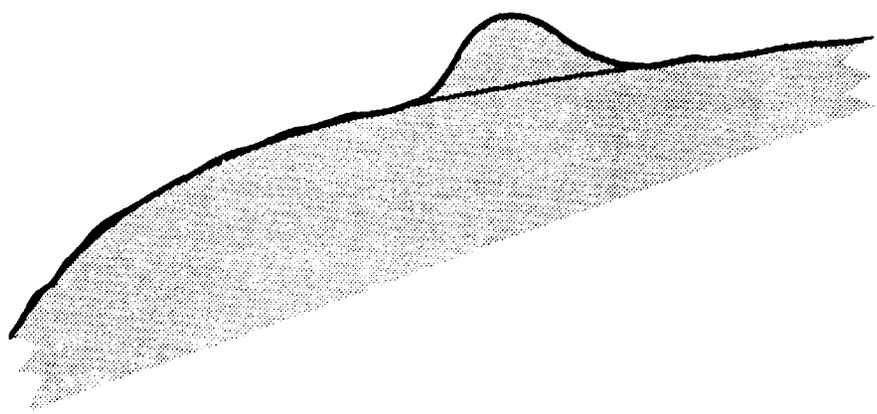

Figure 2.2 The profile of the edge after a single grinding

performed, is possible without additional hardware sensors, since the resulting profile of the edge after deburring is known from the actual joint positions and the kinematics during the motion. One approach is to program some strategy on the user level of the robot controller, i.e. in the robot programming language used. Considering the fact that the detection of the remaining burr and the further grinding of it is quite involved with the motion control, a better approach is to extend the basic MOVE primitive of the system with a special version for deburring. A part of the user level program can then look like:

```
GRINDMOVE grinder ALONG burrpath1
   WITH DEBURRING = burrpars1
   WITH VELOCITY = 100mm/s
   WITH ...
```

where the meaning of `GRINDMOVE` and `DEBURRING` has been added at a level below the user level of robot programming, i.e. tightly connected with the motion control. Such application features should on the other hand be encapsulated and separated from the general purpose motion control system. More on this in following chapters.

2.2 Spot welding

Spot welding is one of the most successful applications for industrial robots. As the application has become more mature, the performance demands have become clearer. The time needed to weld a spot, including the move to the spot and the closing of the weld-gun, is of major importance for the applicability and economical pay-off.

Consider the welding of some part of a car. The spots to weld are the black spots in Figure 2.3. The distance between the spots is around 50 mm, and a typical time for the motion is around 0.3 seconds, which is significant compared to the welding time. When the welding of the spot left to the `spotx` is just finished, then the statement `WELDMOVE WeldGun TO spotx` is executed to move to `spotx` and weld as fast as possible. The statement `WELDMOVE` on the user level will be interpreted and a compiled procedure will be called. It is the actual coding of this procedure that is the problem considered. It has to be built on existing primitive motion commands, to use information from the basic servo loops, to handle signals from the tool, and to include other types of

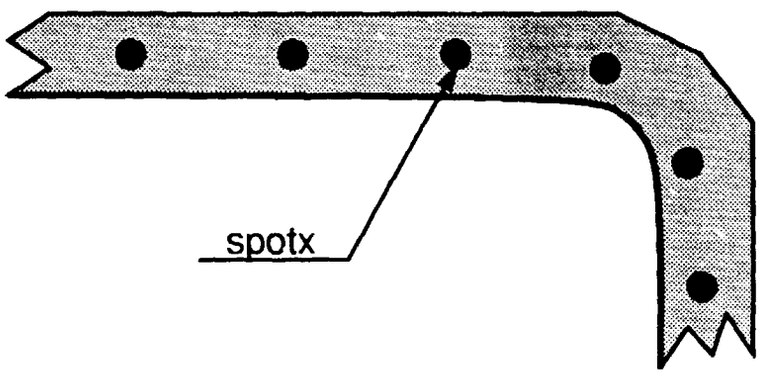

Figure 2.3 Spot weldings around a corner in a car.

application knowledge. Some of the typical characteristics will now be given, and it should be easy to imagine that it would be a tricky task to implement the feature by modifying the basic motion control system. The desirable user level primitives are on the other hand not designed for this type of fast timing, which motivates implementation in the intermediate level of the system.

The short distance between the weldings usually result in short motions for each joint. In this application this means that the major limiting factors are the maximum jerk and the maximum acceleration. The joints will not get close to their velocity limits, and the manipulator dynamics can be considered constant with respect to the joint angles during the motion. Application knowledge, based on heuristics or formal methods, has to be included to optimize this type of motion.

The timing of the events must also be considered. When the move has been completed, then the robot controller orders the weld-gun to close by asserting some signal to the welding equipment. After the weld-gun is closed, the welding itself is controlled by the welding equipment. Since some time passes, after the signal has been set, until the weld-gun is almost closed (i.e. until the motion has to be completed), it

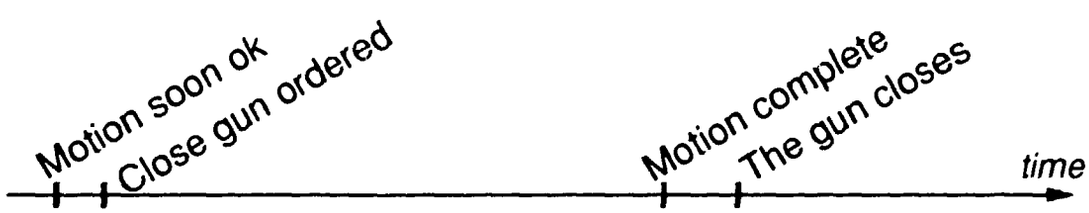

Figure 2.4 Timing for stop of motion and start of welding.

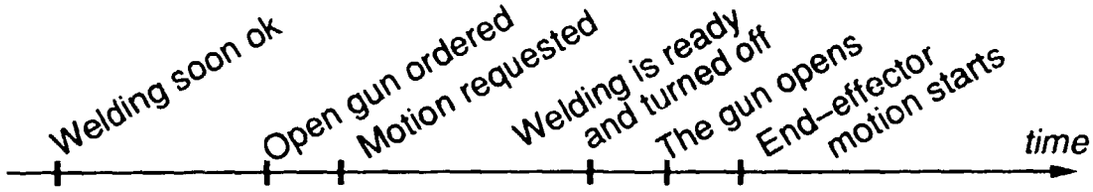

Figure 2.5 Timing for stop of welding and start of motion.

is desirable to signal "close and weld" some specified time before the motion is completed, as shown in Figure 2.4. Similar tricks can also be applied at the start of the motion, where it is required that the welding equipment generates the "welding OK" signal a certain specified time before the welding is estimated to be complete. (Welding is event driven and depends on the welding current and other signals monitored by the welding equipment.) In Figure 2.5 the estimated "welding OK" event is denoted *Welding soon ok*, and the *Open gun ordered* and the *Motion requested* events are scheduled so that the last three events in the figure occurs as close to each other as possible. The performance and timing requirements of this application require that the signaling and scheduling of actions must be performed within the motion control system, or tightly coupled with it.

2.3 Arc welding

Following a path using an external sensor signal is common in arc-welding and gluing applications. Path tracking performance is a key issue for improved productivity, but the optimal parameters depend on the application and the task. As the sensor signal has to be used continuously during the motion, it should be clear that execution of the path tracking algorithm has to be integrated into the motion control system. It is suggested that application procedures should have continuous read access to data in the motion control layer.

Consider for example seam-welding of thin sheet-metal (thin implies high speed), tracking the seam with a laser scan sensor. If the pieces to be welded together have ragged edges, then the raggedness will be equivalent to noise in the path tracking control loop. Good parameters for the path tracking depend among other things on the noise, i.e. on

the raggedness. The tuning must be requested from the user level of the control system, and involves recording of sensor signals during slow path tracking and then computation of the parameters.

Sensors can be incorporated in two different ways in typical commercial robot system currently available. Either sensors are incorporated on a user programming level, affecting the flow of the program but not the behavior of the robot during motions, or sensors are utilized in a predefined way during motions. In the latter case, it may be possible to set some parameters, but the type of sensors (e.g. force, torque, vision, or laser scanner) that can be used and the algorithms for the sensor based control can not be influenced by the user. New types of sensor based motion control can then only be added by the robot manufacturer after considerable reprogramming of the motion control system.

Assume a system where new algorithms for path tracking can be added at a motion control level that is open to the experienced application engineer. Such an application level of the system will of course be application specific, but the tuning of the path tracking in this case will be task specific. The tuning of the path tracking control must therefore be accessible from the user level of the system. A part of the robot program on the user level can then look like the following, where -- means comment:

```
-- Auto tuning:
EdgeData = RecordSeam(CalibPath)
OptSpeed = OptimalSpeed(EdgeData)
OptTrack = Tracking(EdgeData, OptSpeed)
-- End; Optimal parameters computed.
```

where `RecordSeam` (which is a measurement procedure including motions and standard path tracking), `OptimalSpeed()` and `Tracking()` are application features added to the underlying control system. A new feature compared to existing robot program executives is that application features added can sample or record signals in the motion control system, and export the data to the user level of the system. Note also that the procedures `OptimalSpeed()` and `Tracking()` can be given an alternative implementation. One example is in application research, where software packages available on host computer can be used. The variables `OptSpeed` and `OptTrack` can then be used in the user program as follows:

```
LaserSensor.On
MOVE effector TO StartPos
   WITH V=searchspeed
   UNTIL LaserSensor.EdgeDetected
Weld.On
TRACKMOVE effector TO EndPos
   WITH V = OptSpeed
   WITH SENSOR = LaserSensor
   WITH TRACKING = OptTrack
...
```

where TRACKMOVE (special version of MOVE accepting the SENSOR and TRACKING parameters), SENSOR, and TRACKING are application features added to the underlying control system. The encapsulation and implementation of such features will be described later in the thesis.

Other arc welding features, perhaps even more interesting, could be control of the arc welding equipment, integrated with the robot controller. For example, a laser scanner can be used both for path tracking, control of welding equipment, and weld quality supervision.

2.4 Assembly

Assembly cycle time is the key measure of the performance for an assembly robot. It is often too long a cycle time that is the reason for using fixed automation or manual work instead of robots, resulting in less flexibility and manual work that is monotonous. The first example deals with the peak performance cycle time (i.e. reduction of the time required to mount one piece or component), while the next deals with the continuous performance cycle time (usually over half a minute or longer time periods).

Peak performance

In many industrial assembly applications, the part of the cycle time that is spent in excess to the theoretical minimum time is mainly due to slow approach of precise positions (due to robust servo tuning fixed for the robot). The work cell is typically designed with the assembly taking place

in the middle of the working area, and parts are picked up from feeders and magazines in the periphery of the area, see Figure 2.6. Motions between different stations in the work cell are performed at maximum speed. Before a position can be reached accurately, the robot either has to do a smooth deceleration, or make a fast deceleration and then a slow approaching motion. The dynamic effects from the high speed motion will otherwise make the accuracy deficient. It is not practically possible today to have the servo so accurately tuned that a fast motion can end directly at the proper location. Such a tuning would be dependent on pay-load, actuator and gear temperature (which depend on the task, time, and the room temperature), etc.

Control problems with time varying unknown parameters are sometimes solved with adaptive control[26]. This means, however, that the servo parameters will vary during motion, and depend on recent mo-

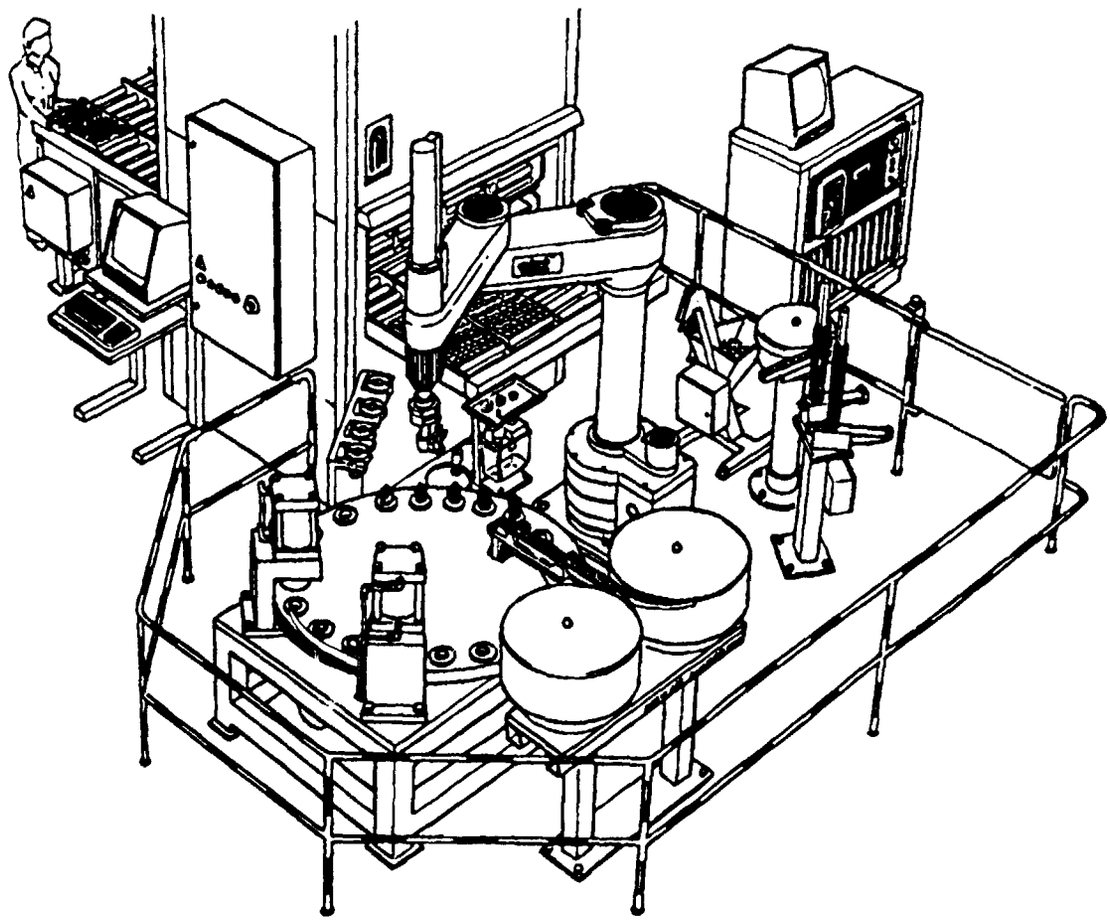

Figure 2.6 An assembly work cell, from [22] with permission from IVF.

tions. Hence, adaptive control for industrial robots requires special care, as repetitive accuracy is very crucial. One solution is to use auto tuning[26] of the servo in one or a few critical locations of the workspace. The request for auto tuning should be issued from the user program, typically when the robot has to wait anyway, e.g. for new parts. This kind of auto tuning is called *dynamic calibration* here. It is related to static calibration of positioning errors and kinematic calibration [25]. The instruction in the user program requesting dynamic calibration can look like:

```
Gripper1.Close    -- Holding bolt now
MOVE gripper1 TO AssemblyPos
BoltTuning = CALIBRATE(ShakyPath)
```

The computed control parameters `BoltTuning` can then be utilized in move instructions like:

```
MOVE gripper1 TO InsertBoltPos
    WITH SPEED=Vmax
    WITH TUNING=BoltTuning
```

There will of course be several tunings for the different work-pieces and for the different locations. User access to auto tuning and use of the task specific parameters should be supported by the control system software.

Overall performance

Time optimized assembly operations sometimes have a problem: The thermal load of the motors gets too high. The problem is that the "time optimal", i.e. torque demanding, motions sometimes need energy saving modifications. The nominal torque demanding motion can be developed with formal methods [10], or interactively by an experienced robot programmer, as well as the modifications can. Assume that we can detect for which joints there is a risk for overload, and that we program modified motions for those cases. How should then the proper motion be selected during execution of the robot program? The temperatures of the actuators are needed in the program to select the right motion. Adding thermal sensors on the actuators can be difficult and expensive. (Built in sensors used for protection of the motors are of on-off type and normally directly connected to the drive power for safety reasons.)

It would on the other hand be quite easy to reconstruct the motor temperature from the torque reference, if the software architecture allows

proper access to the signals required. This can be quite accurate if the room temperature is known. Such a feature should be implemented in the basic motion control, or possible for the experienced user to add. The system software should support use of low level information, as the thermal load, on high levels of the system.

2.5 Materials handling

The solution to the materials handling problem exemplifies how a concept of method passing, that will mainly be used in lower layers of the system, can be utilized also from the user level of the system.

External sensors are sometimes used to identify an object to be grasped, or to detect if the object has been grasped. In the case that it is possible to grasp an object without use of external sensors, enough information to identify the type or existence of an object is very often available in the signals that today are internal to the system. For instance, consider the following piece of code proposed for a materials handling application:

```
MOVE gripper TO GripPosition
   WITH ...
Gripper.Close
IDENTMOVE gripper TO DropPositions[ Id ]
   VIA TopPos
   WITH Id = IDENT(proc, possible_parts)
CASE Id OF
   NoObject : ...
```

where IDENTMOVE and IDENT are application specific features of the system. The algorithm proc for performing the object identification should be coded by an experienced application or control engineer. Note that the destination in the IDENTMOVE instruction is not known when execution of it starts. However, when the via-point is passed, the identity of the object should be known, and motion continues to the corresponding element of the position vector DropPositions[]. The algorithm for identification is passed as an executable parameter to the application level, where it gets access to the data structure internal to the motion control system.

Note that there will be parallel activities going on in the underlying control system, implied by an attribute of a motion instruction, i.e. implied by the user robot program. Parallel activities programmed or exposed at the user level are possible in other systems, and parallel computing within the underlying motion control system is very common, but this type of implicit parallelism seems to be a new idea in robot programming.

2.6 Summary

Some examples have been given of industrial applications which can be better solved if the control system allows a tighter coupling between the servo control level and a level for programming of application features. The examples are interesting and important industrial applications, and they also illustrate at least one important software issue each, namely:

- Programming of application specific real-time control strategies.
- Embedding high performance application features which are based on timing and signaling to and from external equipment.
- Export to the user level and utilization of data sampled in the low level motion control.
- System support of task specific tuning of the motion control system.
- Software demands for emulation of thermal motor load used for motion planning.
- Implicit parallelism caused by move attributes.

These issues give the flavor of the demands on the control system software for current applications. It is believed that these aspects will be even more important in future more demanding applications. This means that the control system must allow the advanced user to introduce new robot motion primitives on a lower level of the system then possible today. The principles for such a system will be developed in the thesis.

3

Levels of Control and Programming

The purpose of the first section in this chapter is to give an introduction to programming of motions. The applicability problems with current systems is described in Section 3.2, and a first coarse outline of an architecture is presented in Section 3.3. This outline will define three main levels of which the upper and lower levels are the traditional ones for programming and control of robots, and the middle level is introduced to solve the application examples in the previous chapter. Each of the three levels will then be refined in Chapters 4, 5, and 7. Section 3.4 discusses the use of internal control signals versus the use of additional external sensors, and Section 3.5 treats the choice of the system programming language used in the following chapters.

3.1 Computer control of motions

Historically, the individual joints of a robot were programmed and controlled directly. The abstraction level and ease of use of the system was increased by having a kinematic model of the robot built into the control

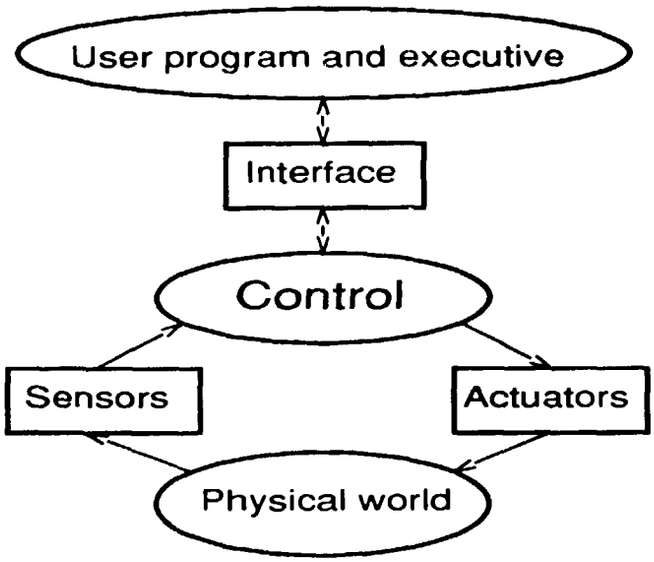

Figure 3.1 Programming and control of motions today. The executive is normally an interpreter executing the user program that specifies the motions. Execution of motion statements result in procedure calls and access of data in the "interface".

system. Robot programming was still manipulator oriented, i.e. the manipulator motions were specified rather than the task to be performed, and further increase of the abstraction level of the interface and the underlying control system has been desirable. A robot motion specified by some instructions in a robot program, as in the application examples in Chapter 2, is typically handled at different levels of the control system. The physical world is controlled by the algorithms implemented at a control level, which communicates with a user programming level via a software interface, see Figure 3.1. The interface can from the user program be viewed as a model of some parts of the physical environment, just like a reference signal to a simple control loop can be viewed as a model of the controlled output. The interface is, in the simplest case, some variables or procedures that allow change of set-point, and possibly change of parameters. The set-point and parameters in the robotics case could be the goal position for the end-effector and motion attributes like speed, end-effector dimensions, etc.

A current trend is to include more knowledge about the physical environment, and thereby having the interface to reflect the robot and its environment in more abstract ways like motions expressed in manipulation of the objects handled. This solution can be suitable for high level

motion description [24], but it has little to do with the feedback control of a robot in normal industrial cases. The reason is that the task, the tools, and the objects manipulated are fairly well known at the time when the robot program is written, or at least at the time when the motion is requested from the user program. A model of the environment is then only required to compute the manipulator oriented motion commands. The database containing such a model of the physical environment is called a *world model.* It would then be attractive from a software engineering point of view to separate the world modeling from the motion control system. Such a separation will be treated in Section 3.3.

One interesting approach for a motion control interface is the MDL language [13], which can be viewed as PostScript (used in modern printers) for robots. However, an intermediate motion description language does not necessarily lead to proper solutions for control of non-perfect robots, just like PostScript does not support printing by use of non-perfect printers. The MDL language is therefore based on so called modal segments [20], which makes it possible to select the control strategy quite freely with only a few basic primitives. The MDL language will be returned to in Chapter 4.

The issue how robot programs should be expressed on the user level is reformulated in Chapter 7. Until then, we can assume that the robot programming language used in Chapter 2, i.e. a notation similar to the one in [9], is used on the user level of the system. The robot programs can be created in a host computer, but are interpreted in the robot control system, where the motions are controlled by a number of computers. The hardware can for instance look like the experimental platform that will be shown in Figure 9.1, where micro processors and signal processors are used in the embedded control system. The structure of the executive software and the motion control software is the subject in remaining sections in this chapter, and in Chapters 4 and 5.

Relations to PLC and servo controllers

The most common way to program motions in the industry today is by use of sequencing controllers or Programmable Logical Controllers (PLCs). Programming in this case means to compose a number of blocks or elements containing the code to be executed into a data structure to be used for periodic execution. Executing the program means to scan

through the data structure, execute the executable code, and map input and output signals according to the attributes of the elements. This model of programming and execution introduces some overhead. High performance servo control is therefore often implemented in separate controllers normally supplied by the motor vendor. The corresponding PLC element then only contains a controller interface for change of set-point, parameters, etc. The demands on the PLC programming environment are quite different from the demands from robot programming which is treated in Chapter 6. On the other hand, robots often work together with equipment which is, or would preferably be, controlled by PLCs. Thus, there is a need to merge the two types of motion control, but PLCs are not well applicable to robot motion control.

Some "stand alone" servo controllers can control and coordinate multiple axes of motions. Such servos are not designed for a particular mechanical configuration and the axes are therefore assumed to be dynamically decoupled, which implies that the user has to implement quite a lot of algorithms and control strategies to achieve best possible performance when the system is multivariable and non-linear with non-ideal dynamics (friction, backlash etc). Robot manufacturers spend many man-years on this control software which is delivered with the robot itself. If we view that motion control software as the internal code in a PLC element, an attempt to structure robot control software can look like a minor issue, but the engineering effort to design that single element can equal the effort to design the rest of the entire PLC system, thus motivating the research presented in the thesis.

3.2 Applicability problems with current systems

The conclusion from the previous section is that robot control systems today have a structure according to Figure 3.2, even if some experimental control systems have a more sophisticated interface to the control algorithms. This means that the motion control part of the system provides a set of predefined primitives for the programming environment and that an industrial robot is adapted to different applications by adding tools, fixtures, etc, and by writing programs on a user level of the system. This is, however, not sufficient for the examples in Chapter 2.

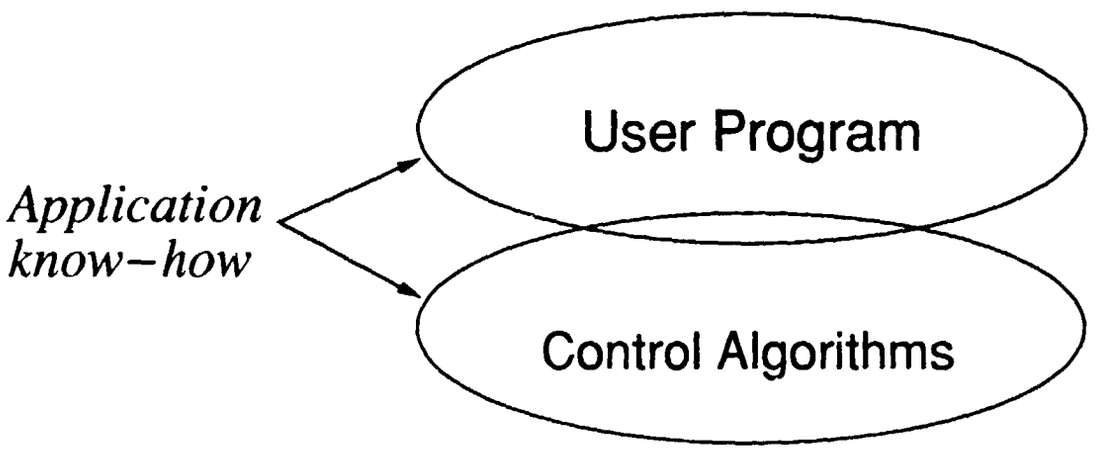

Figure 3.2 Main levels of motion control in (commercial or experimental) systems today.

Inexpensive production equipment combined with demands for high performance, or needs for unforeseen sensor based motion control, therefore result in a need to include application know-how also in the basic motion control system, as indicated in Figure 3.2. The problem with modification of the motion control is that it often requires a substantial engineering effort, or it can even be impossible for the user. A robot manufacturer for instance, does not want to reveal the control solutions developed. It is also a complex task even with access to the source code etc. The reason is that it is complex software with many coupled functions, that are based on mutual primitives, include timing and so on. The seemingly harmless task to include a new robot function may actually require a new control system including new real-time primitives and motion primitives. In practice, it is usually not that bad, but it may represent a major effort. There are also safety reasons for not letting users modify the motion control of the robot. Built-in fault detections can be very sensitive to modifications of the control strategy, resulting in decreased personnel and equipment safety, or spurious emergency stops. If systems should be open on the control level, careful attention must be paid to these issues.

The state of currently available robot systems is that some robot manufacturers have included special motion control features for a few major application, but sensors can only be used in some predefined way, e.g. for certain force control strategies. Application engineers have prob-

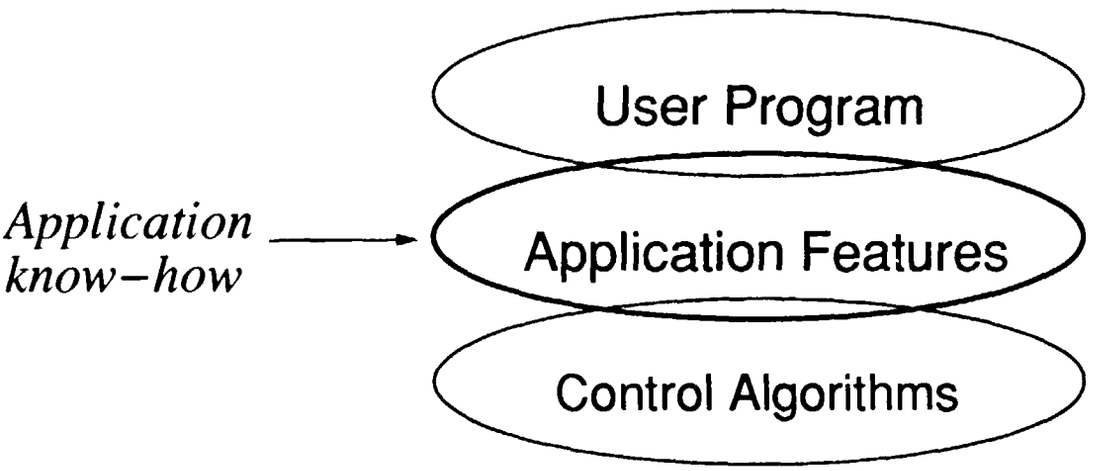

Figure 3.3 Proposed intermediate application **level**. Each level will be further refined into software **layers** later in the thesis.

lems using available robot systems for new applications, and robots are unnecessarily expensive for some applications. The problem is to find a way to support tailoring of the motion control for certain applications like those in Chapter 2, often including incorporation of additional sensors installed by the user.

3.3 A new approach

The problem stated in the previous section naturally leads to a solution with an intermediate software level for implementation of application features, see Figure 3.3. This intermediate level has not received the same attention as other software aspects in robotics like the higher levels, or the low level with explicit joint control. It has been neglected by statements of the type "just implement a procedure" or "implement another robot function". The advantage of the intermediate level is that application know-how does not have to be put in the basic motion control system. On the other hand, just proposing a structure is not a solution until the implementation problems associated with it have been solved. It actually turns out that the structure is not that easily achieved in combination with efficient real time computing. The implementation aspects and the considerations made in adjacent layers will be returned to in later chapters. Typical users at the different levels of the system are indicated in Table 3.1, which is a coarse outline of a more detailed description that

Level in Figure 3.3	Typical programmer	Full design in Chapter
User Program	Ordinary robot programmer	7 ; The User Programming level
Application Features	Experienced application engineer	4 ; The Application Level
Control Algorithms	Control engineer	5 ; The Control Level

Table 3.1 Typical users at different levels of the system.

will be given in Chapter 10. The curious reader can look at Figure 10.1 which is the final architecture corresponding to the outline in Figure 3.3, and the more detailed user table corresponding to Table 3.1 is shown in Table 10.1. Some general aspects of each of the proposed three levels now follows.

Separating the world model from motion control

Recall from Section 3.1 that there is a trend to include knowledge about the environment in the motion control system to increase the level of motion commands. Also recall that such knowledge is contained in a database called a "world model". By utilizing a method-passing concept which will be introduced in Section 4.2, the world model management and advanced robot programming language features can be separated from the motion control, which instead can be focused on the manipulator control. (The commercially available robot programming languages can be supported even without the method passing.) The need for more advanced robot programming capabilities instead leads to multi-layered software for execution of robot programs. High level concepts are implemented in an upper layer, possibly executed in a host computer. We will see later that the robot programming language, including its high level concepts, will be implemented in an executive software layer, which can be viewed as a run-time interface to the user level programming environment. The robot programming languages mentioned in [9], and others, can thus be supported as special cases of the architecture that is developed in this thesis.

The approach in the thesis is to include knowledge about the robot

itself only in the control level. Such knowledge is utilized both for high performance control and for best possible manipulator oriented motion descriptions. Therefore, maintenance of a so called world model is considered as a feature in the user programming environment or executive, which not necessarily has to be supplied with the robot. However, there are special cases when it can be convenient to have the model of the environment integrated in the motion control system. One example is motion relative to a moving work-piece whose position is measured by some sensor. The robot program can then be programmed for a non-moving work-piece, and a separately managed world model data base can be extended for the moving case, without affecting the robot program. However, a concept of method passing that solves these special cases without having world model management included in the motion controller will be introduced later in the thesis. The advantages will be improved generality, improved computing efficiency, and that controllers for simple applications can be kept simple.

The user programming level

Robot programs can be developed on-line or off-line as mentioned earlier. The output from the user programming is robot programs expressed in some Robot Programming Language (RPL). Even if the user programs with the help of some robot programming system with a language that is less complex than computer programming languages are, an RPL just as rich as computer programming languages should be used in the bottom. It turns out that major parts of robot programs consist of statements that are not robot specific [17].

The robot program is interpreted by some kind of interpreter or executive, as described in Section 3.1. Compilation is not used because the programs should be possible to develop and debug interactively and incrementally, and the interpretation overhead is normally not a problem since the mechanics is slower than the computer. There has been some attempts to standardize RPLs. However, standards so far (like [44]) have not dealt with parallel activities, exception handling, special control strategies, and modern computer science concepts in a proper way. They are therefore ignored in the thesis.

The user programming environment must be mainly composed by existing software tools to make an implementation feasible for one or two

persons. (The more advanced off-line programming systems available today for instance, have required more than 50 man-years to develop.) We will return to the internal design of this level in Chapter 7.

The application level

Application features can be of two different types. The first type, which is possible to implement in most systems today, is procedures or data that define operations or primitives that are common for a certain application. These primitives often handles external equipment or sensors that are connected to the robot controller via the customer IO. In an arc-welding application for instance, the welding equipment is controlled via general primitives (like `WeldOn` etc.), rather than specifying exactly what signals are to be asserted for the equipment. This is simply data abstraction. Such primitives can also include motions of standard types. An instruction like `clean_tool` in the arc-welding application would mean: Turn the welding off (if it is on), move to a predefined cleaning position, have the weld-gun cleaned by e.g. compressed air, and return to the original position. Only predefined types of motions are used, however, and this type of application features can be defined on the user level of what is called open systems today, and is not further commented in the thesis.

The other type of application features requires a tighter coupling with the motion control system, and should be implemented on the in-

```
MOVE grinder TO first_corner
    WITH SPEED=0.8*Vmax
    VIA above_first_corner
GRINDMOVE grinder ALONG right_edge
    WITH SPEED=0.15*mps
    WITH FORCE=grind_force
    WITH GRINDING=grind_pars
MOVE grinder TO above_end_pos
GrinderOff
```

Listing 3.1 A sample standard `MOVE` command, followed by a special motion command `GRINDMOVE` implemented in the application level of the system. The `GRINDMOVE` accepts the new `GRINDING` attribute. The last `MOVE` is the standard motion instruction again, and GrinderOff is defined on the user level as e.g. `DigOut[5]=0`.

termediate level of the system as mentioned above. This type of feature can employ special control strategies for dealing with physical effects like process forces, or new types of sensor based motion control that have not been foreseen during design of the base system. (There is no way the robot manufacturer can foresee all ideas the application engineers will come up with.) This second type of application features defines new instructions that can be interpreted on the user level of the system. It would be possible to change the meaning of a standard MOVE instructions to better suit a certain application, but a program that does not use any special features should perform in the same way, whether special tricks have been implemented or not. New names should therefore be given to the special motion instructions that are defined in the application layer that will be designed in the thesis. The user can then define a motion sequence looking like the code in Listing 3.1.

The normal user or programmer of the robot should only program the robot on the user level. The intermediate layer for application oriented programming is intended for e.g. experienced application engineers, application researchers, designers of manufacturing equipment, or control engineers working with customer support for the robot manufacturer. An ordinary computer programming language is to be used for programming on the application layer. The user program executive, the application layer, and lower levels of the control system have to be implemented with a compiled language to achieve type safety *and* computing efficiency. Chapter 4 will describe a design and implementation suited for efficient real-time execution.

The control level

The extensive research within robotics control have resulted in a number of promising control schemes (see e.g. [17]) which, however, have not been implemented in the systems that are commercially available today. One major reason for this is that the structure of the motion control has evolved from cost optimized solutions with the software structure reflecting the (sometimes old) hardware structure, which makes the implementation quite hard. The design that will be developed in Chapter 5 defines an application and implementation oriented structure for the standard motion control. Features for "pipelining" of motion commands are introduced, as well as a sensor interface for external sensors. That interface is

then used for sensor-based motions, and for simulation of such motions for off-line programming purposes.

3.4 Software sensors

Recall that industrial robots are rather precise machines, which implies that signals in the control system reflect the state of the robot arm rather well, but not perfectly. A basic idea in most of the application examples shown in Chapter 2 is to make use of such information already existing in the system. The application layer and a well designed software architecture makes this possible. It is, however, often not clear to people not familiar with control systems, how much information that exists in the servo for an industrial robot. The purpose of this section is to clarify this. Even researchers within the robot control field have used more sensors than would be feasible in many industrial applications, but algorithmic solutions utilizing internal signals have been developed in a few cases, for example in [3].

Consider one link of a robot, i.e a single servo. Assume we use a quite simple control strategy with a PI type velocity controller, and a P type position controller. This is shown in Figure 3.4. It is basically the type of control that is used for the jointwise control of most commercially robots available today. The system can be given desired compliance and damping by only using two P regulators (i.e. $K_i = 0$), but the I-part is introduced to take care of unknown disturbance forces. The output of the inner regulator can be interpreted as an acceleration reference to the controlled system if we have $K_i = 0$ or if we have no disturbance forces. With disturbance forces and with a properly tuned K_i, the output from the control system is a torque (or force) reference to the servo drive unit.

Some information used for dynamic control can be useful also outside the controller according to the following examples. The first example is the most important one; that principle will be used to solve the deburring application problem.

- Recording of the signals e_p and θ during motion over a surface will give the profile of the surface along the path. The deviation of the profile from the nominal one is simply e_p as a function of θ.

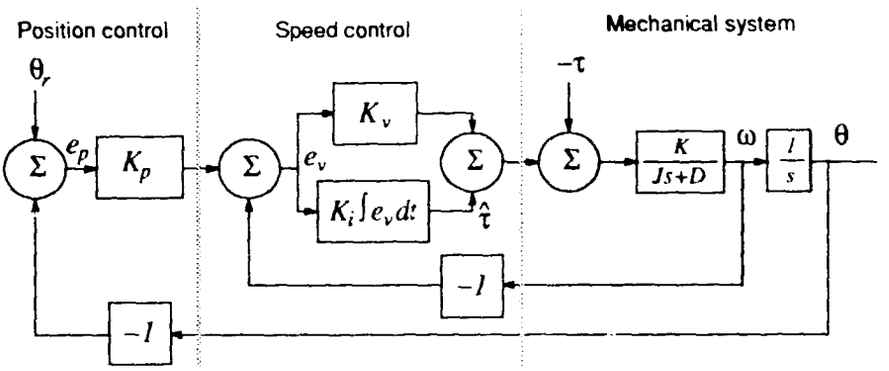

Figure 3.4 Simplified servo control.

- The $\hat{\tau}$ signal is an estimate of the Columb friction in the joint during slow motion with constant speed and no contact with the environment.
- The friction coefficient of an object, or the existence of an object, can be computed from the $\hat{\tau}$ signal, as for previous case and with the internal friction already estimated according to the previous item.
- The signal ω during acceleration and deceleration contains information about the inertia of the joint, which will give the inertia of the load if the inertia of the robot itself is known.

In a complete servo for all joints in a real robot it gets more complicated to identify or compute signals of interest, but the information is in the system, the problem is to get it out. Signals depending on friction or external forces can of course be practically hard to estimate accurately, but can still be feasible in some heavy applications. Note, however, that the first example using the position error during force control is not sensitive to friction effects. The problem of designing the software architecture for the motion control in such a way that safe and efficient real-time access of the control data can be provided, without exposing the internal data structure, will be returned to in Chapter 5.

The essence of this section is that when additional information about the physical environment is needed, then it is sometimes advantageous to use information already in the system instead of installing external sensors. From the user programming point of view it makes no difference, if a procedure called to get sensor data samples the signal from an electrical signal, or if sampling is performed inside the system.

3.5 System programming

The normal robot programmer only programs in a so called robot programming language, while the implementor of the system uses some computer programming language. The programmer implementing application features and sensor interfaces is an intermediate type of user. Should such users program in a more "simple to use" language than the system implementor does, or can the same language be used? The answer is that it is up to the user, but the object code must be possible to link together with the system software. Only one system programming language, selected according to the discussion in this section, will be used in the thesis. The most important criterion is that the language must be well suited for implementation of control systems.

The fact that the control system software reflects different physical properties of the robot and its environment makes an object oriented implementation feasible. A complication with the object orientedness is that it relies on internal states in the objects, which combined with the so called message passing (i.e. calling of methods, not to be confused with the method passing introduced in this thesis) can cause complicated sequences of events that are hard to debug in a real-time system. A complicated inheritance structure can also make it hard to realize what code that will actually be executed, and this deteriorates predictability in a practical sense. In spite of this, the author believes that with awareness of the complications and with proper use, object oriented programming is the best paradigm to achieve good modularization of the software. There is also a need to introduce better abstractions for real-time behavior, e.g. according to [8], but this is left for further research.

An implementation of the system presented in the thesis will use the object oriented paradigm as a base. The C++ language [42] will be used because the C language is the only language that is available for most micro processors and signal processors, and new processors normally come with a C compiler. C-code can be generated from C++, which supports the desired programming paradigms. There is also no garbage collection in C++, which makes the language practically useful for demanding real-time applications. Both C++ and Modula-2 (used for some prototyping) are supported by the real-time programming environment developed within the department [5].

3.6 Summary

Control and programming of robot motions have been described on a conceptual level, and has been compared with other types of motion control. The difficulties to solve the application problems in Chapter 2 with current control systems was pointed out. A new coarse system design with an intermediate level for programming of application specific motion control was then proposed as a solution. This coarse design will then be refined into a complete architecture in the following chapters according to Table 2.1

It was pointed out that many of the, usually internal, control signals can be useful when solving application problems. Both the use of internal signals and external sensor signals are reasons for having a a tight coupling between the low level control and higher levels that open to the user. Such a tight coupling is a key feature of the solutions presented in the thesis.

A so called robot programming language (RPL) is used on the user level of the system. A conventional computer programming language is used for the implementation of the control system itself. A third case is programming on the new intermediate level of the system. The choice of the same programming language and programming paradigm for both the system implementation and for the intermediate level programming was mentioned to give a background to solutions in the following chapters.

4

The Application Level

The design of the intermediate level for application specific motion control will be presented in this chapter. This level is open to the advanced user and contains the application features as was indicated in the outline in Figure 3.3 and Table 3.1. The software **levels** in the outline are refined into software **layers** in the final architecture that will be shown in Figure 10.1 and Table 10.1. This means that each functional **level** in the system is implemented by one or several software **layers**, and the main layer in the application level is the application layer. The software at this level of the system should be implemented in the compiled language used, cross compiled, and executed in the embedded robot control system. A possible hardware configuration will be shown in Chapter 9.

An important part of the specification of the application level is its interfaces to adjacent software layers. These interfaces will be described in general terms in Section 4.1 and 4.3. Some details depend on the solutions that will be presented in the Chapters 5 and 7, and will therefore have to wait until then. Some features required to solve the application problems in Chapter 2 are presented in Section 4.2, which leads to the software environment that is to be used for application oriented robot programming. Some real-time aspects are discussed in Section 4.4, and the application problems are solved in Section 4.5.

4.1 Interface to user programming

The interface to the user programming logically contains two types of interfaces, one used at compile time and one used at run time. The run time interface has been given a software layer of its own, called the executive layer, which encapsulates the robot programming language (RPL) used. The compile time interface is simply the interface of the application layer to the executive layer.

The executive software layer

The motion control system, including application features, must be accessible from the user programming level of the system. The user requests motions by executing instructions like `MOVE`. Execution typically means interpretation of program instructions or manually entered commands. Compilation of user programs is generally not used as mentioned in Chapter 3. The layers below user programming, however, form a compiled system to achieve efficient real-time computing. The interpretation of the user's `MOVE` instruction will have to result in a call to a compiled procedure which we name `Move`. Attributes to `MOVE` specified with e.g. `WITH` or `VIA` will be passed as actual parameters to the `Move` procedure in some specified way. Other parameters to `Move` could be e.g. the target frame, and the frame in which the target frame is specified (see [17] for a description of frames as coordinate systems are usually called). In a simple case a user program instruction can look like:

```
MOVE gripper TO drop_pos
   WITH SPEED=0.5*Vmax
   VIA top_pos,above_pos
```

which may result in the following calls performed by the interpreter:

```
path = make_path(gripper, drop_pos);
via_path(path, parsed_vias);
par = make_par(parsed_pars);
Move(path, par);
```

The syntax of the RPL used for the user program is not important here. Some RPLs are well known from the literature [9], while others are specific for different robot vendors like [2]. The thing that matters here is that the motion control system is accessed via compiled calls written in an ordinary computer programming language. The software implement-

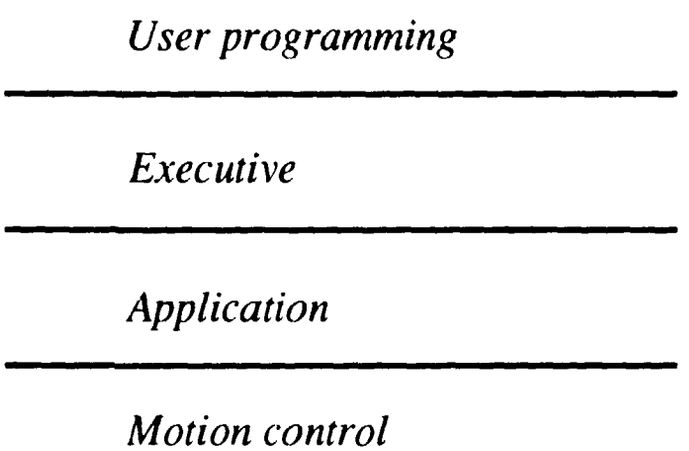

Figure 4.1 Software layers between user programming and basic control. Opposed to the functional description of the coarse design according to Figure 3.3, this figure describes the underlying implementation. The *application* layer implements the application features in Figure 3.3, except for the run-time interface to the features. That interface depends on the RPL and therefore belongs to the *executive* layer.

ing these calls, as well as interpretation of other RPL instructions, forms a layer of its own, namely the executive layer. This layer provides an interface to the user programming level, and separates the user's view of motion requests from the underlying implementation. The major part of the executive will, however, provide facilities comparable with those of any computer programming language. When the RPL is changed, the layers below the executive layer does not need to be changed.

The application layer

The motivation for the intermediate software layer presented in Chapter 3 was that the basic motion control should not need to be changed when the motion control system is given new features for a new application. Such a feature can be the `GRINDMOVE` instruction described in section 2.1. The application layer should first of all make the interface to the basic motion control system (see next section) available to the executive layer. It should then also be possible for e.g. the experienced application engineer to implement features required. The structure developed so far is shown in Figure 4.1.

Application layer interface to executive

The purpose of the executive layer is to provide a suitable user programming view of the control system, regardless of how it is implemented. The application layer must provide a procedural or object oriented interface to the executive layer. One of the advantages with the proposed structure is that the application layer can be used to give the motion control system an interface that is tailored to special executive demands. Assume for instance that we want to make the control system directly compatible with the RIPE [35] programming environment. The motion control system is then modeled by the following abstract base class, written in C++ [42] and quoted directly from [35].

```
class Robot : public Transport
{ protected:
    point home, current;
    // <... some more private data...>
  public:
    // Abstract class so contructor is empty
    Robot();
    virtual ~Robot();

    virtual int move(point loc,
                     int motion_attributes,
                     double speed=0.0);
    virtual int move_rel(point delta,
                         int motion_attributes,
                         double speed=0.0);

    // <... 15 more virtual member functions...>

    virtual void where(point cur_loc,
                       int coordinate_type);
    virtual int report_status();
};
```

If we want an interface to an ABB IRB-6 robot control system for example, we then inherit from the `Robot` class to a class `IRB6` in which we implement all the virtual methods (for that robot) of the base class. A general purpose robot programming environment (that calls the vir-

tual functions) can then remain unchanged when the robot is changed. This standard interface can be given to any commercially available robot system (since the functionality of these systems can be expressed by an abstract data type [35][1]), but then built on top of the executive and RPL provided by the robot vendor. The system presented in the thesis can be interfaced directly, but even more important, application features can be added. Continuing the RIPE example, a new abstract base class called `GrindingRobot` can be defined according to:

```
class GrindingRobot : public virtual Robot {
  protected:
    GrindingPars grind_pars;
  public:
    GrindingRobot();
    virtual ~GrindingRobot();

    virtual int GrindMove(point loc,
                          int motion_attributes,
                          double speed=0.0,
                          GrindingPars gpars=NULL);
    virtual int GrindMove(path gpath,
                          int motion_attributes,
                          double speed=0.0,
                          GrindingPars gpars=NULL);
};
```

The executive interface for an ABB IRB-6 grinding robot can then be defined through multiple inheritance from class `IRB6` which is derived from class `Robot`, and from the abstract base class `GrindingRobot`. This is shown in Figure 4.2, which perhaps not shows the best object oriented style, but this solution is compatible with RIPE and provides proper functionality and reuse of code (the implementation part of the `IRB` class might be pre-compiled and not possible to change). Another solution would be to say 'a robot has bahaviors', instead of saying 'a grinding robot is a robot' as shown in the figure, and have a separate inheritance hierarchy for behaviors.

Thus, we see that an object oriented interface to the motion control system, including application features, is quite practical. The RIPE object oriented interface is one example of how the motion control system

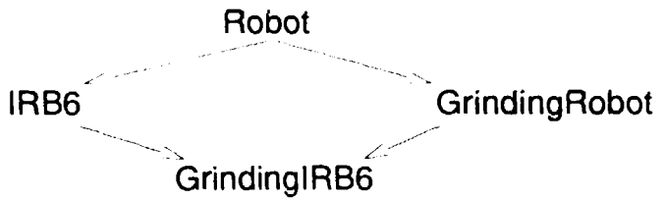

Figure 4.2 Both the properties of a certain robot (IRB-6) and the features for a specific application (grinding) are gathered through multiple inheritance to an IRB-6 grinding-robot class.

can be encapsulated. However, it would need to be extended with the features, like method passing, that will be presented later in the thesis.

The executive provides a run-time interface, i.e. the RPL, for the user programming level. Such a run-time interface or language can be based on some fundamental primitives for motion control, like the modal segments [20] in the MDL language [13], which uses a basic motion specification in terms of reference signals, gain scheduling vectors and time segments which are applied to a state space model exposed to the user. The advantage with that approach is that generality is achieved with a small number of primitives, but it is not believed to be the best solution. But anyway, assuming we would like to have a system based on the MDL approach, then note the following feature with the proposed architecture. We can use the application layer, not only to encapsulate application specific control features, but also to change the interface to the underlying motion control system so that the so called modal segments can be used. This is provided that the standard motion control features are general enough, which they will be according to the next section.

The MDL language, which is only one of over hundred RPLs, was mentioned above because it is very general with only a small number of primitives. It was claimed, however, that it is implementable within the ORC architecture. A more traditional user level language is, however, more feasible for industrial use, and then combined with programming in the application layer, where the expressiveness of the system programming language together with the architectural features can be used. Improved control performance and computing efficiency can then be achieved, at the price that new motion primitives cannot be included at run-time, which is not a problem for industrial applications.

4.2 Method passing

The motion control system must provide the following features for the application layer to improve performance and flexibility compared to traditional systems:

1. The data in the **path** specification (refer to the piece of code in the previous section) should be allowed to be functions of states and time.
2. Functions should also be possible to give as control parameters.
3. An optional third argument containing so called actions should be possible to pass to any motion procedure. Actions are not restricted to pure functions, with all arguments passed as input data and the output being of a specific type, the actions can be scheduled as parallel processes communication with the calling process.

A special type of function is a table look-up function, which together with a general version of the `Move` procedure can provide the same functionality as the MDL language does. Items 1 and 2 can be incorporated in a ordinary abstract data type interface, while the third case needs special software support. A call to the basic motion control level from the application layer can then look like (in C++):

```
ML = new MotionLayer; // Once at startup
...
ML->Move(path, par, act);
...
```

The **par** parameter could in principle be as general as the **act** parameter, thus making the latter unnecessary. It has, however, been decided that the **par** parameter may only contain functions that are free from side effects, while the **act** parameter can have side effects on a supplied data structure but cannot effect the states in the standard controller. The risk that control stability, built in fault detection, proper real-time scheduling, etc. is deteriorated by included application features should thereby be minimized.

Software mechanisms like call-back routines and inheritance are needed to achieve a proper structure of the robot control software, and particularly of the application layer. However, such concepts are not supported by the programming language in the multiprocessor case. The

actions will instead be used as a fundamental concept to achieve a proper structure and efficiency. This motivates a separate treatment of that concept.

Actions

It has been claimed that robots are nothing more than abstract data types [1]. This is basically true, because even if classes are used to define the interface to the robot, it is basically only abstract data types that are needed. This means that inheritance and other object oriented features can be used for convenience when different robots share some properties, but if we consider a single specific robot, an abstract data type is sufficient. We then have some methods (like `move`) we can use to access the data that is internal to the system. However, if the robot is considered to be an abstract data type (in our compiled system), the methods have to be known at compile time. The problem is that when the basic motion control is compiled (by the robot vendor), the application specific control strategies are not known. Thus, robots cannot always be viewed as abstract data types, if the software also should encapsulate general motion control, application specific motion control, etc.

The natural extension of the abstract data type paradigm to cope with the need for adding more methods (in the object oriented sense), is to use inheritance and object oriented programming. All methods of the basic motion control system could be inherited, and additional features could be added in a subclass, but this is not a general solution in the multi processor case. The base class can then have an implementation part (also containing some interprocessor communication code) running in processor A, and an interface part defined for the program in processor B. The processors can be of different types, for instance one M68030 micro processor and one DSP32C signal processor as in the system that will be presented in Chapter 9. The processors then have totally different executable code, different sizes of standard types, different addressing of memory, etc. Even if the processors were of the same type, there would be problems with different address spaces.

Consider development of the application layer program for processor B. A standard object oriented implementation would mean inheritance of basic motion control (implemented on processor A), and adding application features as new methods in the derived subclass. The problem,

which has been named "the method passing problem", is the following: Application specific methods must be possible to pass to the base class object (in another processor) for execution with proper real-time performance and efficiency. Four different possibilities to pass executable code as parameters have been examined.

1. A interpreted language, like LISP, can be used. A textual representation can then be passed, and the method is executed by calling the `eval` function.
2. Methods can be composed out of a small set of operations required for typical known applications. The set of operations is predefined, and they are composed according to ordinary data parameters.
3. Modify the compiler to make it possible to interleave different types of cross compiling according to new compiler directives.
4. Combine existing compilers and tools into a two stage compilation strategy. Some code processing programs might be required to interface standard software.

Solution **1** does not fulfill the hard real-time requirements, and a language like FORTH is not rich enough. Alternative **2** is not flexible enough. The third alternative leads to programming and maintenance (updating for new processors and language extensions) requiring too much manpower. Alternative **4** will be the approach for implementing actions and function arguments in the sequel.

The method passing problem has also been recognized in process control of batch processes. So called recipes contain a formula (i.e. the data part), and a procedure (i.e. the method) that should be passed to the process control system for execution. This problem has normally been solved by approach **2**, but since batch processes normally are slow compared to the control system, approach **1** could also be used.

Implementation of actions

Actions are relocatable procedures that are defined at an upper layer of the system, and passed as a parameter for execution in the next lower layer. Actions are needed because otherwise the execution and hardware structure of the control system does not match a layered application oriented structure. Consider for instance the deburring application example in Chapter 2. A `GRINDMOVE` command was introduced in the user pro-

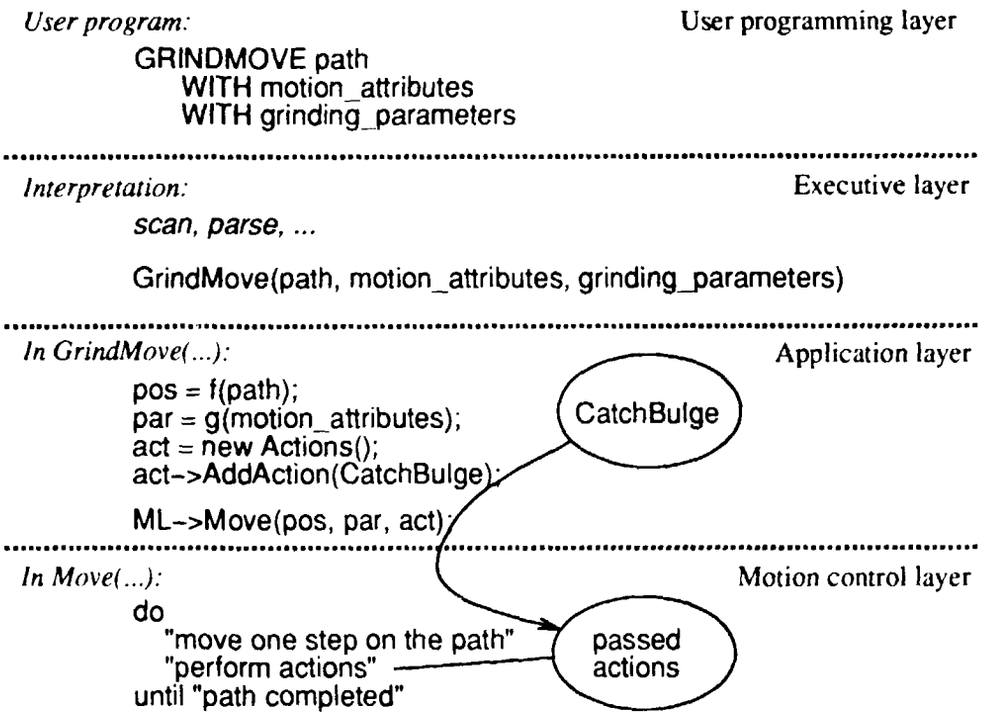

Figure 4.3 Execution of GRINDMOVE at different levels of the system. An action is passed to the bottom layer where it is called at "perform actions".

gramming level, and no additional sensors were to be used. The key problem is to detect a bulge during the motion, without slowing it down, and utilizing a general purpose (i.e. without features for grinding predefined) motion control system. The bulge detection and identification must then be implemented as an action. The principles of the execution of the GRINDMOVE instruction are illustrated in Figure 4.3. The action is called CatchBulge in that figure. Parts of the application layer code will be presented in Section 4.5.

Actions are to be (cross)compiled into relocatable executable code. Most C compilers (and thereby C++) can be given directives to produce code with PC relative addressing of executable code, i.e. no linker is required to modify the addresses. Additionally, data must be addressed in a relocatable way. This is achieved by having all variables allocated on the stack, i.e. no global or static variables in the C++ program. All actions are given a data structure of a fixed type via an argument to the procedure. The address of the data environment will then be passed via the stack, and all data addressing will be relative to that address.

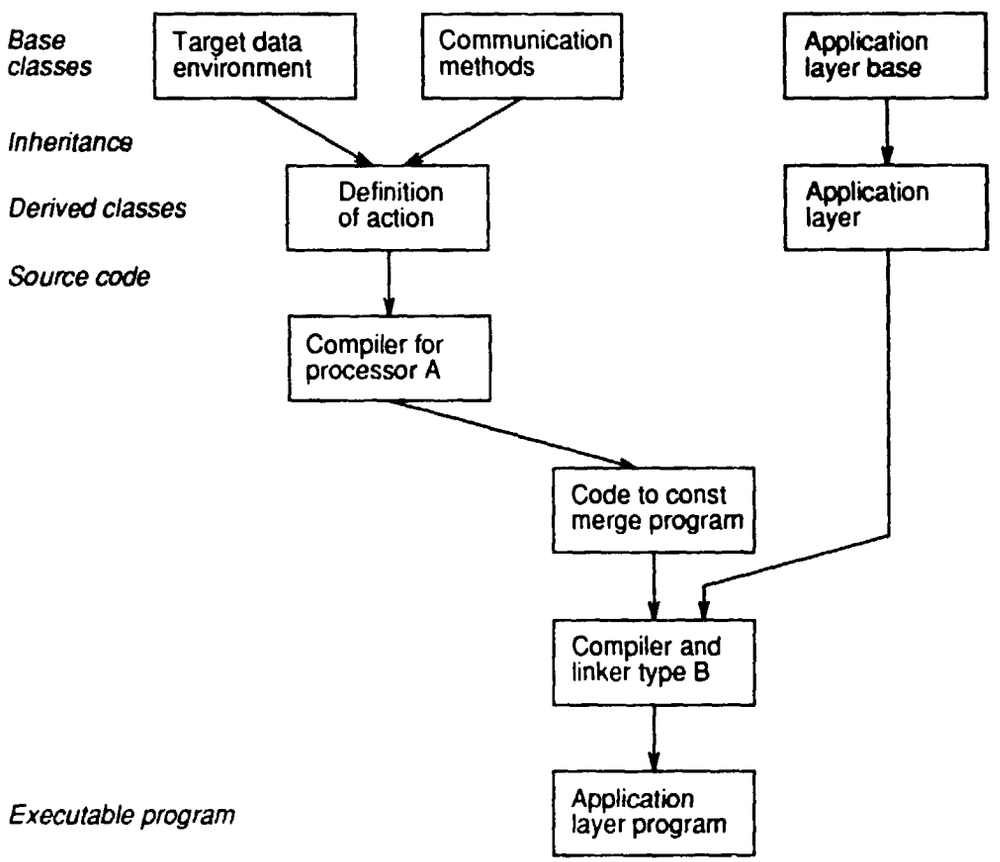

Figure 4.4 Generation of the application layer executable program, including actions to be passed at run time. The actions are to be executed in processor A which can be a DSP, they are defined in (and passed from) the application layer program which is to be executed in processor B. Thus the executable code for the action is treated as data in the processor B, which can be an M680x0. The entire code generation strategy shown in the figure is executed in the host computer of type C.

An action can therefore be treated as data on the upper level, and as executable code on the next lower level of the system. The system generation is illustrated in Figure 4.4. The data environment passed to the actions must of course contain pointers to procedures for communication with the upper level, data IO, etc. The aim of the software solutions in the thesis is that the execution in the control system tailored for a certain application should be as efficient as if the entire system was hand coded for that application. Further implementation details are given in Chapter 5.

4.3 Interface to basic motion control

The interface to the underlying motion control must contain methods for accessing the robot arm control as well as external joints for e.g. fixtures or feeders. The arm control system is normally provided by the robot vendor, while additional servos of almost any type must be possible to incorporate into the motion control. The definition of the motion layer interface below provides some functionality for external axes, but methods for customization of kinematics and dynamics to include external axes are not included. Management of customer IO on the servo level is also not included. The interface is, however, sufficient for motion requests including the external servos. Note the different overloaded versions of the move procedure, which are further commented below.

```
class MotionLayer {
  public:
    enum mv {ready, spawned, illegal, fault, act_stopped};
    // Optional arguments default to NULL => default used.
    mv Move(JointSpace motion, Params pars=NULL,
                               Actions act=NULL);
    mv Move(JointSpace motion, FuncParams pars,
                               Actions act=NULL);
    mv Move(CartSpace motion, Params pars=NULL,
                              Actions act=NULL);
    mv Move(CartSpace motion, FuncParams pars,
                              Actions act=NULL);
    mv Move(FuncSpace motion, Params pars=NULL,
                              Actions act=NULL);
    mv Move(FuncSpace motion, FuncParams pars,
                              Actions act=NULL);

    Params GetDefaultDefault();
    Params SetCurrentDefault(Params pars);
    Params GetCurrentDefault();

    enum mode {off, standby, block, brake, run, track};
    mode* SetMode(mode new_mode, JointSet joints);
    mode* GetMode(JointSet joints);
```

```
    jlag ResetRef(JointSet joints);
    clag ResetRef(DOFs dofs);

    mode* Reset();

  protected:
    \\ States, modes, etc...
};
```

The class definition above is a simplified definition of the interface to the underlying motion control, and is to be included when compiling the application layer program. The basic motion control is separately compiled and its implementation is normally invisible to the user. Base classes for the arguments to `Move` will also be defined, but are not shown in this chapter for brevity. Three types of motion spaces *times* two types of parameters (constants or functions) result in six versions of the `Move` procedure, which are named the same by use of overloading. A return from a `Move` procedure call can be either when the motion is ready or interrupted in some way, or can be immediately when the motion has started. Suitable behavior is requested via the `act` parameter, and the result is obtained via the returned value of type `mv`, which is the enumeration type defined. Motions can be performed in joint space or in Cartesian space. A third alternative is what has been named "function space", which means that the components in a Cartesian space motion are allowed to be functions of some variables, like time or path coordinate. Problems like specification of motions relative to moving objects, or motions relative to a fixed object but with a varying end-effector, are then solved in a uniform way that has not been seen anywhere else.

Parameters are also allowed to be functions, which explains the type `FuncParams`. The reasons for having different types for function objects and data objects are that efficiency can be improved, and the implementation of the function objects have some specific problems, as actions have. `Actions` contain a predefined data structure and procedures that can operate on much of the data internal to the `MotionLayer`, so they can be viewed as methods of the `MotionLayer` class that are not known at compile time. This imposes some problems that are commented under the method passing problem below.

A number of different servo controller modes can be handled by the `SetMode` and `GetMode` procedures. `ResetRef` is something that is useful in sensor based search strategies. The implementation of the `MotionLayer` will be returned to in Chapter 5.

4.4 Real-time considerations

It is well known that additional unknown dynamics are often introduced into the controllers due to deficient real-time system design. The problem is not in the inner fixed control loops, which can be implemented according to the standard sequence:

```
"initialize"
LOOP
   "sample"
   "compute control output"
   "output control signal"
   "update controller states"
END
```

The problem is that the outer control loops for sensor based motion control and application features are closed around the inner control loops and other algorithmic computations, which have to be split into different real-time processes to get a modularized and practical design. Fluctuations in CPU load due to algorithms requiring more computing in special cases, multirate sampling, non-periodic processes, etc., combined with buffering in the interprocess communication, introduce unwanted time varying dynamics. Seam tracking by use of a seam-finding sensor can serve as an example. The basic position control is then part of the outer application specific control loop.

The experience is that a variety of real-time primitives are required to achieve both efficiency (e.g. data shoveling kept to a minimum) and proper dynamic behavior for the software. Some commercially available real-time operating systems and kernels have tried to simplify the programming by only providing a few real-time primitives. These primitives are then either on too low a level for some problems, or the primitives are on a higher level (e.g. pipes, messages, etc) resulting in more complex dynamics in some cases. The real-time kernel used in the current

work is developed within the department and assumes no specific concurrency model [5]. A number of different primitives like semaphores, events, monitors and messages can be combined to achieve the wanted real-time structure. Other suitable real-time kernels may exist, but more complex real-time operating systems like OS-9 and VxWorks are less attractive for low level control. Hard real-time problems are preferably solved, in our case, with small, simple and fast primitives that can be extended according to special demands. (Hard real-time means (here) that the "cost" for an additional execution delay is significantly more than proportional to the delay.)

Scheduling of actions

Execution of the actions described earlier requires some care. Actions are to be executed in the context of a process that is predefined in the system. This should prevent improper scheduling of other processes, for e.g. fault detection, due to errors in actions implemented by the user. In principle, hierarchical scheduling [11] would be a good solution to ensure proper scheduling. That would mean that a low level scheduler would be used for the actions, ensuring that the execution is kept within the time slot given from the standard motion control. That low level scheduler would run under the scheduler used for other processes in the motion control layer, but the following execution scheme is instead adopted for simplicity: Actions to be performed during execution of a standard method are passed to that method (i.e. to a move procedure) and collected in a predefined data structure like:

```
class ActionSet {
    friend class MotionLayer;
    ActionSet* actions;
    ActionSet* pre_actions;
    ActionSet* post_actions;
    ActionSet* cont_actions;
    // ...
}
```

The `pre_actions` are executed before motion starts, `post_actions` are executed when the motion is finished, and `cont_actions` are installed for execution also after the motion is completed. The `actions` that are to be

executed during the motion are simply executed in sequence according to the order they are passed to the motion command. Actions can therefore not interrupt each other, and actions cannot interrupt the motion control, as they are called as procedures in that context. The question whether or not the motion control should be allowed to interrupt actions is still open. It is clear that it must be possible in a timeout situation to ensure control of the equipment, but the mutual exclusion problem is eliminated if execution of the actions is interrupted only between actions. Which solution that is the best depends on the following details (given without comments): The execution rate for the scheduler, if a timeout counter in hardware (for times shorter than a clock tick) is available, if the (C library) routines `setjmp` and `longjmp` are implemented (or possible to implement for the processor used), how the compiler treats floating point registers, and if exception handling is supported by the compiler. A simple approach sufficient for experimental use is to assume short actions, but also to provide a procedure that can be called from the action when interruption is allowed.

Exceptions

Exceptions in robot control can be of two types [16]: System software error, and external state errors. The former type should be supported by the compiler and the run-time library and is not further commented. The latter type should, however, be supported by the control system. The action structure simplifies the implementation of these exceptions as the exceptions in the application layer can be defined as actions, and exception handlers are then the code that handles the result of an action. Exceptions exposed to the robot programmer can be implemented by use of actions in the same way. The way exceptions are used on the user programming level is then defined by the executive layer. Examples: It will be shown in the next section how an action can be used to detect a bulge in the deburring application presented in Section 2.1. The occurrence of a bulge is not considered as an exception, since it is an expected case which is explicitly taken care of in the application layer. In the arc-welding application on the other hand, loss of the tracked seam would preferably be defined as an exception in the application layer, and passed to the user level in a way defined by the executive layer. Returning to the deburring example, the new motion primitive `GRINDMOVE` which

was used in Section 2.1, will call the standard `Move` procedure (supplied by the basic motion control) from the application layer. Any exception which is predefined for the standard `MOVE` is only passed to the executive by calling `ExceptionHandling` as will be shown in the next section. (The actions corresponding to the predefined user exceptions are included in all `Actions` objects by the constructor for that class.)

4.5 Solving application problems

We will now return to the application problems stated in Chapter 2. A few of these problems have been partly solved in e.g. robot control systems developed at ABB Robotics, and probably also by other manufacturers. The software and controller design in such a commercially available control system requires an effort of several (more than 10) man years to develop. The application features that have been implemented are, however, not well encapsulated. The implementation of such a feature therefore requires several months of programming, tuning, and retesting of the basic motion control system. This has to be done by the robot manufacturer's control engineers, and the result is that only features for major well-known applications get implemented.

With the system developed in this thesis, motion control application features should be possible to implement by e.g. the experienced application engineer. Less effort should also be required due to better encapsulation of control features. It will now be indicated how application features can be added according to the principles developed so far in the thesis.

Deburring

A grinding tool is moved along the edge of a casting in the deburring application. The nominal profile is known, see Figure 2.1, but rather than following the nominal path exactly, force control is applied in order to make the edge of the casting smooth. The purpose of the application layer is to encapsulate things like the special handling of remaining bulges that should be ground down. Strategies for doing this extra grinding should be programmed in the application layer, while the parameters of a particular task have to be specified by the ordinary robot programmer.

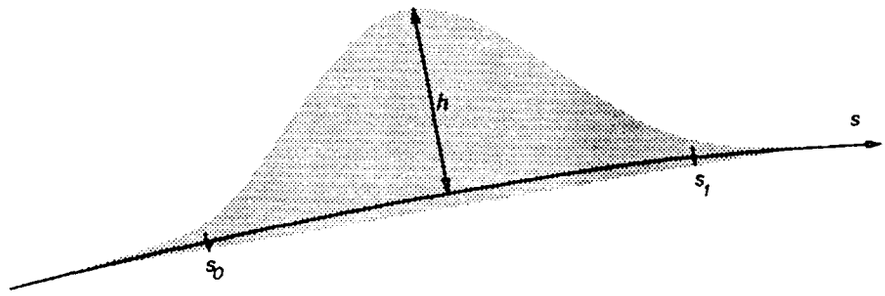

Figure 4.5 A remaining bulge after deburring.

These parameters are collected in a structure that is type compatible with the DEBURRING parameter (please refer to Deburring of castings section in Chapter 2) which are passed down from the executive of the system. The parameters are then utilized in the application layer as explained below.

A simple grinding strategy would be to apply a grinding force that is proportional to the deviation from the nominal (i.e. programmed) path. A more advanced solution is to compute a modified path reflecting the actual desired profile of the edge, and use the modified path for the detection of bulges. (The modified path can be estimated from the programmed nominal path and the actual position during force control.) This strategy allows larger tolerances for work-pieces and fixtures. The modification of the nominal path is not shown below, but use of this more advanced strategy implies that bulges are detected **after** they have been ground once.

Assume that a remaining bulge can be characterized by the start and stop path coordinates s_0 and s_1 and the height h as shown in Figure 4.5. The detection and recording of the bulge is done in the `CatchBulge` procedure in the program below. The maximum deviation between the modified path end the actual path equals h. After some checking that s_0, s_1 and h are within reasonable limits, an additional grinding strategy can easily be computed since the entire profile of the bulge is known from internal sensor signals. This is done in the procedure `ComputeBulge` below. Note that all computation on the application level is done while the underlying motion control moves the robot. At a path coordinate s_2 (close to s_1), the nominal motion is interrupted, and the bulge is ground down, according to the procedure `GrindDownBulge`, by moving the tool back and forth over the bulge (like a human worker would have done).

Grinding is then resumed at s_2. The central part of the application level code (written in C++) would then be as follows (neither handling of multiple events for one MoveSignal (e.g. a bulge at the end of the path), nor support for multi processor execution according to Figure 4.4, is covered for clarity):

```
// Declarations:
enum ActIndex {OK, BULGE};
ActionResult
Finished(MotionData Environment) {
    ...  // Def of procedure.
}
ActionResult
CatchBulge(MotionData Environment) {
    ...  // Def of procedure.
}
...

// Application feature initiation:
Act = new Actions;
Act->AddAct(Finished, OK);
Act->AddAct(CatchBulge, BULGE);

// Procedure implementing the grinding strategy:
GrindMove(Path BurrPath, ...// Pars
  ...
  do {
    Move(BurrPath, GrindPars, Act);
    Wait(Act->MoveSignal);
    ActResult = GetResult(Act);
    event = ident(ActResult);
    if (event == BULGE) {
      s0s1hEtc = ComputeBulge(ActResult);
      s2 = StopMove();
      GrindDownBulge(s0s1hEtc, s2);
      BurrPath = PathLeft(BurrPath, s2);
    }
  } while (event == BULGE);
```

```
  if (event != OK)
    ExceptionHandling(event);
  else
    return; // Motion complete
}
```

where text after // are comments, and where `ActionResult` before the procedure definitions denote (as in the C language) the type of the data returned by the function.

Recall from the motion control interface above that a motion was requested with a call from the application layer, invoking a motion procedure on the motion layer. The generic form is: `Move(Pos, Par, Act)`. Depending on the type of motion we have different versions of the `Move` procedure overloaded with respect to the first argument. In this case we do the following call: `Move(Path, Par, Act)`, or whatever is suitable for specifying the nominal path. As mentioned above, `Par` contains control parameters used to achieve the desired dynamical behavior in the motion control layer. In this case the force control parameters, path speed etc, are put in `Par`. The third parameter `Act` (actions) is used to achieve desired logical behavior not preprogrammed in the motion control system. In this case we want to monitor the Cartesian position error, which will be the input to the "deburring controller" implemented in the application layer. The `Act parameter` will contain a request for the Cartesian position error as a function of the path parameter s which is to be supplied too. In this way, it should be a straight forward programming task to implement the deburring controller.

Spot welding

In the spot-welding application we execute robot program instructions like `WELDMOVE WeldGun TO spotx`, wanting highest possible performance for the usually short motion. Let us examine how the `WELDMOVE` instruction is executed at different levels of the ORC structure. For brevity, we only consider the timing when welding starts according to Figure 2.4, and not the complete timing in Figure 2.5.

In the application level, tuned for spot welding in this case, the application specific parts of the instruction should be executed or transformed to calls to the application independent motion control level. The robot controller has been integrated with the spot welding equipment at the

application level. The tool can be installed from the user programming level via an INSTALL instruction, implemented in the executive layer as it does not request any motions. Arguments to an INSTALL instruction will then be the identities of the physical IO channels used, and some characteristics of the particular equipment (e.g. timing). When executing the WELDMOVE instruction on the application level, the timing is known, and the demand for special control for short motions is also known. Assuming there is a special control feature for short motions in the motion control layer, the central part of the application level code can look like:

```
// Code executed when installing the tool
Gun = new WeldGun(... // IO channels etc.
...
// Procedure for optimal spot welding:
WeldMove(Pos, ...
    ...
    if (ShortMotion)
        ShortMove(Pos, Par, Act);
    else
        Move(Pos, Par, Act);
    Wait(Act->MoveSignal);
    Gun->Close();
    Gun->Weld();
    Gun->WaitWelding();
    Gun->Open();
    return;
}
```

where MoveSignal is a semaphore, and the arguments to the ShortMove procedure have the meaning as for the move procedure, which has been explained earlier.

In spot welding, the pay-load is constant and known for the application and the mass and inertia parameters of the load are sent as parameters in Par to the motion control. The Act parameter will contain an action that will evaluate the condition $t \leq T_d - \tau$, where t denotes the time, T_d the time when the motion will be completed, and τ the reaction time of the welding equipment. The variable T_d is computed in the motion control layer, and τ is passed as a numerical value from the application level. When the condition evaluates to true, the action will signal the

semaphore `MoveSignal` (possibly a remote semaphore in another computer). Return from a call of a `move` procedure is done regardless if the motion is completed or not. The call `Wait()` for the `MoveSignal` can be called immediately if waiting is desirable, but if some computing should be performed during the motion, this can be done before calling `Wait`. In this way the application level programmer can achieve this without dealing with tasking etc. It is also easier to utilize the computing power of a multiprocessor control system.

Arc welding

The following primitives need to be implemented in the application layer to achieve proper path tracking performance:

1. `RecordSeam(path)` moves along the `path` using low performance path tracking.
2. `OptimalSpeed` and `OptimalTracking` are design procedures for computing optimal path tracking parameters.
3. The types of the variables used in the auto-tuning code in Section 2.3 should be defined (not shown).
4. The `TRACKMOVE` and `TRACKING` keywords should be defined, i.e. it has to be possible for the user to request motion using the optimal path tracking parameters computed.

An alternative design would be to have only one procedure for the entire path tracking controller design, with data invisible to the user. The reason for the implementation presented here is that the design stages should be open to the user. If someone with extensive application knowledge is not satisfied with e.g. the `OptimalSpeed` procedure, it should be possible to replace that part only. In a development stage this can be done without recompiling the application layer by implementing the function in an interactive computing package running on a host computer system. The possibility to connect the robot controller to programs in a time sharing environment is an example of tools required for development of advanced industrial robot control systems.

Assembly

The solution to the peak performance problem stated in Chapter 2 was to introduce auto tuning controlled (i.e. requested) from the user level program. How this is implemented in the application layer depends on what features that are available in the basic motion control system. We have three different cases:

1. Auto tuning is available in the basic motion control system. The tuned parameters can be accessed from the application layer.
2. Variables of interest for the tuning can only be recorded (e.g. via actions), but programs for tuning are supplied (possibly only object code) with the standard application layer software by the robot manufacturer. The underlying motion control can then be simplified.
3. No direct support for auto tuning, but the system structure presented in this thesis makes it possible. Actions for recording of variables are implemented in the application layer, as well as a controller tuning algorithm. This algorithm can also be run on a host computer if the robot controller is connected to a communication network.

It should be a fairly straight forward programming task to implement the auto tuning feature in the executive layer, possibly except for the algorithm itself which requires more extensive knowledge in control theory. It is then up to the application engineer programming the application layer, what primitives that should be added to the user level.

Another part of the assembly application problem was to make use of the maximum continuous torque of the motors. The maximum peak torque and superior tuning gives good peak performance, while letting the motors of the robot work close to the maximum continuous torque gives good overall performance. The continuous torque is limited by the thermal load of the motors, which is basically a first order low pass filtering of the difference between the environmental temperature and the square root of the integral of the squared torque signal. The model can easily be calibrated by running the robot to emergency stop caused by thermal overload, which is detected by sensors of on/off type built into the motors and connected directly to the emergency stop hardware. The executive layer must make it possible to use the thermal load estimate in a convenient way on the user level, e.g. by simply supplying a predefined

variable or procedure as for any sensor. The application layer software must handle the updating of the estimate, which can be achieved in the following ways:

1. An action that does recording and prefiltering of the torque signal can be passed to the motion control layer. Filtered torque signals are sent to the application layer with a certain sampling rate, all the time from start of the system.
2. The passed action can also do the computing of the thermal load, which increases CPU load in the motion control, but decreases the amount of data that need to be sent to the application layer.
3. Thermal load estimation is built into the standard motion control.

The user program can remain the same independent of which alternative that is selected, but alternative 3 is judged as the best engineering solution as this is such a basic feature (even if it is not found in commercially available systems).

Materials handling

In the materials handling application it should be possible to use the `IDENTMOVE` command with the `IDENT` attribute. This is implemented in the application layer as the following steps:

1. Request a proper type of motion from the motion control layer towards the first via-point (there has to be at least one). Recording for a fixed time (say 0.2 seconds) of torque references and internal sensor signals is requested via the `Act` parameter described earlier.
2. When recorded data has been received, objects are identified (if possible) by applying the procedure obtained as a parameter from the executive level, on the recorded data.
3. The complete motion request to the proper final frame is sent to the motion control system. If the final motion specification does not arrive to the motion controller in time, the movement is slowed down and eventually stops.

No special features other than recording of signals are required in the motion control layer and below for this application. There are also no special algorithms defined in the application layer, it is mainly an extension of the executive so that the `IDENTMOVE` in the user program can be

executed. The reason for this is that the identification procedure is task specific, so the definition must be done on the user level of the system. The real-time execution of the procedure, however, is to be performed in the application layer. The conclusion is that action passing is needed not only from the application layer to the motion control layer, but also from the user programming level to the application level. The latter type of actions can be defined, compiled and executed just as the application actions. Which user instructions that accept this type of actions is defined in the executive software layer.

4.6 Summary

The design of the intermediate level for application features has been made in this chapter. An executive software layer was introduced to encapsulate the robot programming language used. That layer then provides an interpreter for the user programming level, and compiled procedures in the underlying motion control system are called during interpretation of motion instructions in the user program. The set of motion procedures available can then be extended by the advanced user, without changing the standard motion control system. This is achieved by having a software layer, called the application layer, on top of the basic motion control software. The possibility to extend the set of motion primitives, rather than dealing with low level effects on the highest level of control, can be compared with a skilled human worker that performs the work efficiently without having to think about the details of how to do it.

The interfaces between the layers have been briefly described, and the implementation is preferably done in an object oriented style. The main implementation problem is to achieve a tight and efficient coupling between the application layer and the underlying motion control layer. This is solved for the multiprocessor case by so called actions. Actions are a type of parameters consisting of executable relocatable code that is passed from a higher to a lower layer, where they are called and can access otherwise internal data and control signals. Some more features, such as function parameters, have also been presented, as well as solutions to the application problems stated in Chapter 2.

5

The Control Level

This chapter presents the design of the bottom control level of the system as shown in Figure 5.1, which will result in the lower software layers in the final architecture that will be summarized in Figure 10.1 and Table 10.1. A partitioning into three software layers for proper encapsulation of dynamic and real-time properties of the control algorithms is presented in Section 5.1. The upper of these three layers, which provides the interface to the application layer, is then treated in Section 5.2. The interface must of course support "actions" as described in Chapter 4, but also other demands have been recognized. These novel aspects have not been found in the literature, and are introduced and treated below.

Recall that these lower layers of the system must be designed for efficient real-time execution in micro processors and signal processors.

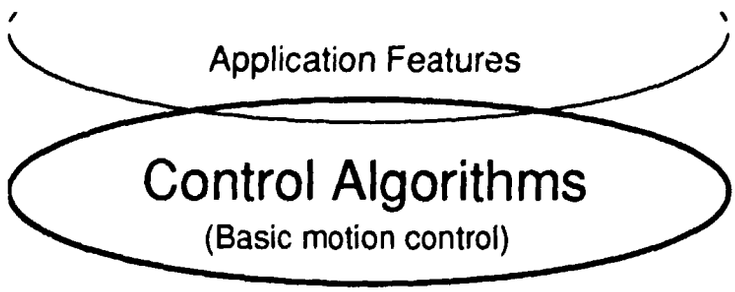

Figure 5.1 Lower half of Figure 3.3. The structure of the *basic motion control* will be developed in this chapter.

Robot motion algorithms have therefore traditionally been split into two types. One type is algorithms that can be used at run-time and thereby also for motions which depend on sensor inputs. The second type is algorithms that require substantial CPU-time (before the motion can start) with the computing power normally provided. Such algorithms have only been possible to use in an off-line context, but features for precomputing and buffering (called pipelining) of motion commands presented in Section 5.2 will allow more powerful algorithms to be used for on-line execution. A special issue is then how sensor signals should be managed, since the required signal is not available in advance.

Off-line programming also requires special features for incorporation of sensors. The motion control system should provide a simulation mode for off-line programming purposes, because the basic control software written by the robot manufacturer is kept secret for practical and business reasons. The software design for simulated motions combined with pipelining of motions and combined with incorporation of external sensors is a major issue in Section 5.2.

Section 5.3 presents the arm control layer, and describes how an algorithm for path velocity control can be rewritten to suit incorporation of external axes. Both external axes control and parts of the manipulator joint control can be distributed to the individual servos. This part of the software belongs to the lowest software layer, and is described in Section 5.4. Some implementation details will be discussed in Chapter 9.

5.1 Background and basic design

The motion control system includes software for all servo controlled motions in the work cell. The control system has to provide servo control for both the joints of the robot, and for axes external to the robot. The difference between the control of the robot joints and the external axes is that a multi-variable servo controller for the robot joints is supplied by the robot manufacturer, while the control and configuration of the external axes must be possible to select also by others, e.g. by a control engineer not working with the basic arm control. Commercial systems today only offer very restricted reconfiguration and tuning of external axes, and experimental systems do not treat external axes in a way that allows separate and user friendly configuration of them. The software

design and control structure must support solutions to such practical demands.

Classes, in the object oriented programming sense, are used for different purposes in the design of the control system. Classes will encapsulate features of physical objects, real-time and multiprocessor aspects will be encapsulated, and computational entities are collected into package-like or module-like classes. Some of the class interfaces model the interface to a software layer in the system. Mirroring the model of the physical world, as shown in the lower half of Figure 5.2, in the controller/process interface gives a feasible motion control structure. The resulting motion control structure is shown in the upper part of Figure 5.2. The servo controllers for the individual joints form the *Local control* layer, and the fixed structure control that is tailored for the robot arm is captured in the *Arm control* layer. The upper layer for the general motion control is called the *Motion control* layer, which interfaces to the application layer described in Chapter 4. These three layers will be described in the next three sections. Only one arm is considered for simplicity, but multiple arms can be incorporated on different levels of the system depending on the degree of control interaction required.

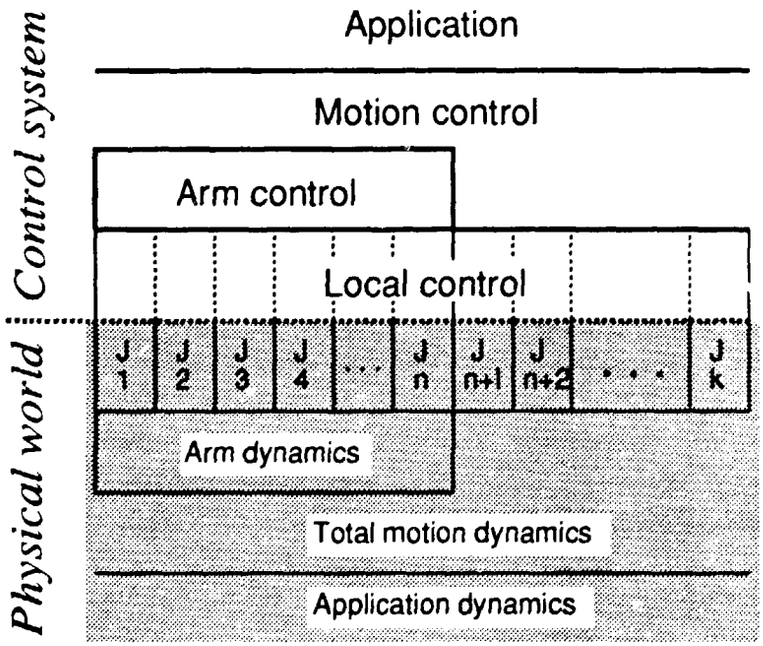

Figure 5.2 The control software layers correspond to process dynamics. J1 to Jk denotes joint 1 to joint k, each having a local control interface as in Figure 3.1. Only one arm is considered for simplicity.

5.2 The motion control layer

The motion control layer contains the general purpose motion control algorithms for all incorporated servos. It may also provide execution of the parts of the arm control algorithms that do not fit into the more static real-time scheduling used in the arm control layer. Such aspects will, however, be left as implementation details, and the rest of the section treats the interface aspects to the environment and to the application layer. A concept for high throughput of motion control computations, similar to pipelining used in CPU hardware, will also be introduced.

Services provided for the application layer

The services available to the programs outside the general motion control, i.e. to the application layer software in this case, can be divided into two types:

1. Services for performing motions.
2. Services for acquiring knowledge about motions.

Systems today are designed for type 1, while the second type of services is not supported, at least not by commercially available systems. Such acquiring services can be to return information about:

- Suitable parameters for a certain motion.
- Predefined kinematic parameters and inverse kinematic solutions.
- Dynamic performance in certain locations.
- Simulation of a sequence of motions.

The acquire services can be utilized in the robot control programs on upper levels (i.e. in the application layer, executive layer, or on the user programming level) of the system to improve performance and flexibility of the robot and its programming environment. The need for such knowledge has also been noticed from the field of off-line programming. Robot manufacturers do not want to reveal their control solutions and software models for the robot due to confidentiality reasons and software maintenance reasons (improving the control system should not impose updating of software tools distributed all over the world). Note that services like the kinematics are included in one specific software layer, even if can be viewed as a public service in the system. The reason is that the

the layered architecture being developed, specifies layers of programming and not the underlying implementation.

The acquire services listed above are handled in two different ways, depending on if it is for simulation purposes or not. Simulation of the robot system is proposed to be achieved by instantiating a motion control object that controls a dynamic model of the robot instead of the real robot. An extra argument to an overloaded `MotionLayer` constructor implies that the motion control object controls a virtual robot instead of a real robot with its drive power electronics etc. Instead, additional computing power for the simulation might be required, but the entire control software is the same. Whether or not the virtual robot (i.e. the simulated one) and the real robot can be run simultaneously depends on the real-time performance of the system, but it is possible in principle.

The other type of acquire services is available via special member functions, or they can be requested via parameters to ordinary member functions, like `Move`. This looks the same for both simulated and real robot motions.

Special member functions are used to provide access to the kinematics, dynamics, and other algorithms of interest for motion planning on higher levels of the system. The idea is that **the same executable code should be used as for the motions control of the robot**, but the execution is of course made in the context of the calling process (or in the context of a call-server process in the multiprocessor case) with a lower priority than the actual motion control.

Another kind of parameter acquisition is used to improve real-time performance of the system. The background is that when more powerful algorithms are used to achieve improved motion performance, and computing power is limited, computations necessary before the motion can start, may delay the motion more than improved performance can speed it up. However, much of the computations can often be performed in advance, i.e when the motion specification is known but before the motion may start. The standard approach is then to do all precomputations in a robot program as an intermediate stage after the user program is written, but before execution starts. The problem is that robot motion instructions are often interleaved with conditional expressions which depend on sensor input. Positions may also be taught in or modified by the robot operator after the program execution has started. The impli-

cation is that run-time support for precomputations is necessary for e.g. optimum time algorithms.

Features introduced to solve the precomputing problem shall as much as possible be hidden for the normal robot programmer. Software solutions similar to hardware solutions like pipelining and cache memories (used in modern CPUs) will therefore be used. Recall from Chapter 4 that one version of the `Move` procedure in the interface to the motion control looked like (omitting default values for clarity):

```
mv Move(CartSpace motion, Params pars, Actions act);
```

The parameters `Params` supplied with a motion request contains the data members `causality` and `mv_descriptor` declared as:

```
class Params
{ public:
    enum causality {done, during, save};
    causality computation;
    int mv_descriptor;
    // ...
}
```

The value of the variable `computation` means the following:

`during`: Computations and motion are performed in the same call. The motion starts as soon as possible, and remaining computation are performed during the motion.

`save`: As much as possible of the computations for the motion are made in advance, and the result is stored in a simple precomputed-motion data-base as element number `mv_descriptor`.

`done`: The motion can start immediately, as the preparatory computations have already been made. Such a preparation can for instance be computation of the minimum time trajectories, and the result can be retrived by using the index `mv_descriptor`.

Use of the wrong move descriptor (compare with file descriptors in UNIX) for a motion, or if the descriptor points to data that has been overwritten by a more recent call of `Move`, must not result in disaster. Start and target positions for the motion are therefore checked to be the same (as those of the corresponding `Move` of type `save`) before parameters of type `done` are used. In case of mismatch, the `during` case is simply imposed, which means that the motion must be fully specified also when it refers

to already computed and stored data.

Making use of the caller's knowledge about motions

Two cases have been identified where it should be possible for the application layer to supply extra information in addition to the motion specification itself. This type of knowledge from the calling layer answers the following questions:

1. To what extent will the motion be subject to change by external sensors at run time?
2. Can the robot dynamics be assumed to be constant (i.e. not depending on the joint angles) during the motion? If so, we call the motion *short.*

The application layer may of course need the motion layer services described above to conclude if a motion can be considered to be short, but the conclusion will depend on the application (and possibly on the task) so the final information flow goes from the caller of `Move` to the motion control layer. Utilization of both types of knowledge will now be described.

The short and simple solution for the second question is that the motion control layer should contain an additional member function for short motions, named `ShortMove`. The `ShortMove` procedure can compute trajectories in a simplified way considering special control strategies for short motions. Please refer to the spot welding application example in Chapters 2 and 4.

The rest of this section treats Question 1 and incorporation of sensors. Motions, as specified in the call of `Move`, depend on sensor information to one of the following extents:

- Completely known, i.e. does not depend on sensor information.
- Almost known, i.e. only minor modifications according to sensor information will be made at run time.
- Partly known, i.e. the motion basically depends on sensor information, but some nominal path exists.
- Not known, i.e. the motion is entirely specified by sensor signals.

Note that in the motion control layer, the task is to create references to the underlying arm control and local control. This is called *planning and*

generation of nominal trajectories in the sequel, and the nominal trajectories do **not** depend on internal sensor signals used in the underlying arm control and local control. This assumption simplifies the data flow in the system. The nominal trajectories can then, however, be modified into actual trajectories utilizing internal sensor signals, which will be described in the arm control section. The sensor dependency discussed here considers only external sensors, i.e. sensors added for a certain application or task (sensors can still be internal in the sense of software sensors, see Section 3.4).

The application layer program passes the information about the degree of sensor dependency via one additional enumeration data member which is added to the `Params` class above according to:

```
enum mv_spec {completely, almost, partly, not};
mv_spec motion_known;
```

The meaning of the `mv_spec` enumeration constants is explained by the following examples:

`completely`: Most motions occurring in the user's robot program are completely known at the time the instruction is interpreted. All services available from the motion control (like optimal trajectory planning in advance) can be employed.

`almost`: The motion is requested from a user level instruction immediately after the starting position has been found by a sensor based search strategy, or a path tracking sensor is used to follow a path with only small deviations from the nominal path. Proper trajectories can be computed in advance, but full performance should not be requested (to leave space for control actions due to sensor signals) and too fine tuning does not make sense.

`partly`: A path tracking sensor can be used to track an almost unspecified path. Specifying a nominal path can be good for:

- Influencing the path tracking to search in preferred directions.
- Specification of a motion termination condition. A motion from one location to the same location in a specified time will for instance result in a specified time for the path tracking motion.
- Having both a specified and an actual motion provides data that can be used in actions for e.g. supervision.

A partly known motion is otherwise treated as a **not** known motion.

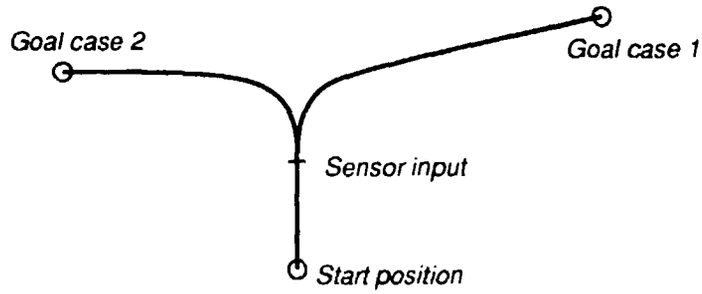

Figure 5.3 Path and trajectories not known in advance. The goal position depends on the input from a sensor.

not: The typical example is manual move of the robot arm by use of a joystick. None of the optimal trajectory planning schemes, like minimum time, can be used. Instead, *trajectory planning and generation* schemes useful at **run-time** must be used. Note that this also applies to a situation like the one shown in Figure 5.3.

Solutions to the run-time trajectory planning and generation problem have not been found in the literature. (Do not confuse this with e.g. the solutions in [17] where the trajectory generation is done at run-time, but not the trajectory planning.) Such trajectory generation is, however, necessary in practical applications, and at least one major robot manufacturer therefore has considerable competence within this field. Technically, motion increment buffers for a distance at least as long as the distance required to stop is maintained. The buffers are then scanned through and used together with a reference model of the robot. Joint synchronization algorithms including extensive heuristics are applied.

Incorporation of sensors – system aspects

We have so far considered the fact that motion may depend on external sensor signals supplied by the robot user, but how should sensors be managed in the control system software? Some principles and specifications for a sensor interface will be given in this section, considering the following aspects:

- Object oriented modeling of sensors.
- Application specific or task specific control and filtering of sensor signals.
- Sensor signals in precomputation of motions.
- Sensor signals for simulated motions.

It is quite natural to model sensors with a general base class, like the class `Sensor` shown in Appendix A, and with derived subclasses for each special type of sensor. Each sensor class must specify three different behaviors to support pipelining and simulation of all motions:

- *Real* sampling is the normal case. The data is read from a sensor interface for external sensors. Some hardware handlers may be predefined, but scaling and possible transformation have to be implemented by the sensor engineer.
- *Dummy* sampling must be provided so that precomputation (e.g. computation of minimum time trajectories) of almost known motions can be carried exactly the same way as for completely known motions. The dummy sampling typically returns a worst case value (possibly depending on time, positon etc.) for the sensor signal that will affect the motion.
- *Simulated* sampling must be defined to allow sensor based control of purely simulated motions, i.e. virtual sensors should be possible to use with the virtual robot arm. The behavior of each sensor (given by the application or task) is defined, compiled, and passed to the sensor object in the motion control layer, in the same way as actions are.

Note that dummy sampling must be possible also in the simulated case, and that real sampling, dummy sampling, simulated real sampling, and simulated dummy sampling might be executed simultaneously in different real-time processes (with priorities in the order mentioned). From the sensor engineering point of view, it is required that all behaviors for a certain type of sensor are specified in one subclass (to simplify incorporation of new types of sensors) and the system has to automatically ensure that all required objects are instantiated. This is solved in the following way.

A motion layer object is dedicated to either control of the virtual servos (simulation), or to control of the real servos. The proper case is selected at instantiation time via an argument to the constructor of the motion layer object, which then instantiates a `MotionSensors` object for the proper case. The object of type `MotionSensors` is the manager for all sensors, either for the simulation case or for the real control case. The sensor manager remembers which case it handles via an internal

flag, but the remaining sensor software system will be designed in a way that the flag is only needed when new individual sensor objects are instantiated. The following two overloaded constructors are defined for the `MotionLayer` class:

```
class SimulationParameters;

class MotionLayer {
  // ... see earlier definition...
  MotionSensors *sensor_env;
  MotionLayer() { // Creates object for real control.
    // Create the real sensor interface:
    sensor_env = new MotionSensors;
    // ...
  };
  MotionLayer(SimulationParameters SimPars);
    // Create sensor interface for simulation:
    sensor_env = new MotionSensors(SimPars);
    // ...
  };
  // ... and so on according to Chapter 3...
};
```

The simulation version of the constructor will replace the process interface, including drive unit interfaces, internal sensor interfaces, and the external sensor interfaces mentioned in this section, with their predefined dynamic models. This is implemented in a few constructors. All other executable motion control code remains the same, including sensor management and motion precomputation using dummy sampling of sensors. We can then concentrate on the normal real control case. The system is designed to provide the other modes automatically.

Before an external sensor connected to the control system can be used, the corresponding object has to be instantiated from the class defining the interface to the sensor. Such an interface for a specific type of sensor should be derived from a base class with features that must exist for all sensors (e.g. a method for returning the value of the sensor signal). Member functions declared as `virtual` in the base class (called `Sensor` in Appendix A), can then be (re)defined in a derived class specifying a specific sensor type, and by defining a virtual function as pure virtual

(`virtual f() = 0;`), the compiler checks that the implementation of that method has not been forgotten. The use of inheritance makes it possible to separate the sensor specific parts of the program from the standard software.

Because we always allow precomputations of sensor based motions using the so called dummy sampling, it must be ensured that both a dummy sensor object and a real sensor object are instantiated. This is achieved and checked at compile time, via a class `SensorPair` which is predefined in the system. By having all members of the `SensorPair` class declared as `private`, and the sensor manager declared as a friend, the compiler checks that the sensor pairs are only created by the sensor manager `MotionSensors`. Please refer to the C++ code in Appendix A for details. The system software solutions are quite tricky in some parts to simplify the parts that are subject to change when new types of sensors are incorporated.

Incorporation of sensors - user aspects

New types of sensors, that for control performance reasons must be incorporated in the motion control layer, cannot be introduced in the software by the ordinary robot programmer. However, an engineer with some programming skill can be certified to write sensor interfaces, following a methodology with seven steps shown below. The system will support incorporation of quite complex sensors with different operation modes, dummy behaviors, etc., but we will focus on the normal simple case here. As an example, a laser sensor interface follows (note that the complete code that needs to be written by the sensor engineer is given):

1. Define the data type returned by the sensor, e.g.
```
class LaserSensorValue {
  public:
    float distance;
};
```
2. Define a class specifying how sampling is performed. The only thing required by the compiler is that we have a `Sample()` method, and that we derive the specific sensor from the base class `Sensor`. In this case we define a constructor too, because we want to be able to specify the hardware connection at run time:
```
class LaserSensor : public Sensor<LaserSensorValue> {
```

```
  private:
    IO_spec connection;
    LaserSensor(const IO_spec HW_connection);
    LaserSensorValue *Sample();
};
```

3. Write the implementation part for the constructor:

```
LaserSensor::LaserSensor(const IO_spec HW_connection); {
  connection = HW_connection;
  DummyVal.distance = 0.05; // 50mm = middle of range.
  SetMode(single); // Sample when Sample() is called
};
```

4. Write the implementation part for the sampling:

```
LaserSensorValue *LaserSensor::Sample() {
  LaserSensorValue *val = new LaserSensorValue;
  val->distance = MotionIO(IO_spec, FLOAT, READ);
  return val;
};
```

5. Implement other features like special dummy sampling, other sampling modes, sensor driver process, prefiltering, exceptions, etc. Note that the constant dummy value could be set in the constructor above, and simulation sampling is never implemented in this layer of the system because it is unknown at compile time. No additional features are needed for this simple sensor.

6. Include the enumeration number, the textual name, and the class name in the static `SensorDefs` class shown in Appendix A. This defines the sensor interface to the upper layer of the system, i.e. to the application layer, which together with the executive layer provides the user programming level view of all sensors defined.

7. Compile the new code, and link it together with the original system.

It is around 25 lines of code that have to be written for a simple type of sensor. Quite strong inheritance and type checking is made at compile time, but a preliminary linking with a sensor interface debugging program would be a good idea. Computing efficiency and simple coding for new sensors, in spite of the precomputation and simulation aspects, is achieved at the price of a more complex system software, which is implented and tested once for all.

The sensor can then be installed from the application layer, e.g. as in the following:

```
ML = new MotionLayer;
// ...
sensor_descriptor LS; // For Laser Sensor.
LASER = ML->GetSensorType("Laser distance");
if (LASER>0) {
  LS = ML->InstallSensor(LASER, AnalogIn, channel);
  // ...
```

If the sensor should be used in an action for example, the sensor descriptor is the key to get access to the right sensor in the motion control layer data structure available to actions, function parameters, and paths specified by functions of sensor values.

5.3 The arm control layer

The arm control layer was introduced to separate the optimized arm control algorithms, and the fixed and efficient implementation of them, from the general purpose motion control and from the local control of individual joints. The arm control actually contains what is normally meant by robot control. Algorithms like "computed torque" and "hybrid force-position control" are still not much used in real industrial applications, but one aim with the system structure developed in the thesis is to ease implementation of such algorithms.

Functionality – specification and problems

The arm control software layer is encapsulated in a class. The features provided for the motion control layer are basically the following:

- Kinematics and inverse kinematics for the robot arm,
- Dynamic parameters for the robot arm.
- Setting of control parameters and modes.
- On-line trajectory time scaling for torque limited path following [18] must be implemented and possible to activate. The *nominal trajectories* are then modified with respect to the actual torque. The modified trajectories are called *actual trajectories.*

- Nominal trajectories can be passed to the arm control, either as generated trajectories passed incrementally point by point, or as complete trajectories parametrized in some predefined way to be generated in the arm control.
- Acquiring of control data used in the arm control. For example:
  - Actual generated trajectories for a motion.
  - Internal sensor signals.
  - Control errors for position, velocity, and force (for force control) can be read (and preset in special modes) by the motion control layer.
- Feed forward signals to torque and velocity references can be set. For example, signals that are added to the torque reference for the joint drives can be useful when implementing force control strategies not supported by the arm control layer itself.

It is interesting to see that internal control and sensor signals, which are supplied to the motion control layer by the ar control or local control layers, can be accessed during execution of an action. The action is implemented in the application layer, but used via the executive layer from the user level of the system. This is used to solve some of the application problems in Chapter 2, for example the deburring application problem.

The software layers within the basic motion control level are application oriented, as opposed to other systems which usually are algorithm oriented. This has the following two structural implications:

1. Several of the algorithms used in the arm control often need to be extended to handle all joints in the work cell, i.e. also the external axes which are installed by the user. The algorithms should then be structured in a way that additional joints can be added in computational stages outside the arm control layer.
2. The design of the multivariable arm controller is complex if the complete dynamics including compliance, friction, backlash, torque ripple, unknown pay-load, and external forces on the end-effector is considered. This is a major reason why e.g. computed torque is not used in commercial robots today. The design could instead be divided into joint-wise controllers (to be implemented in the local control layer) and a multivariable part (to be implemented in the

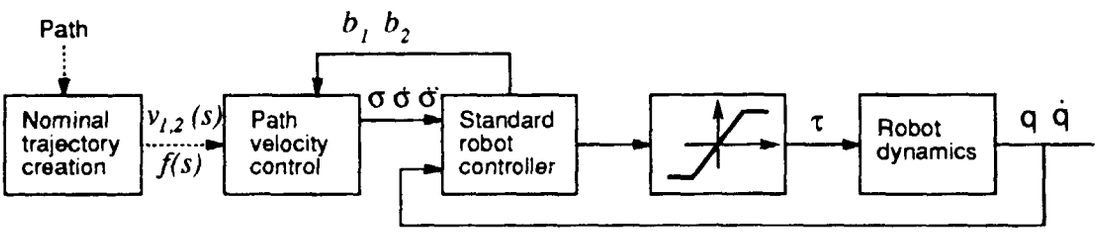

Figure 5.4 Path following according to [18].

arm control layer) considering some of the joint-wise effects.

Implementation of joint-wise control, useful for external axes or for robot joints according to item 2, will be treated in the next section, but separation of the joint-wise control and design of the multivariable controller is left for further research. An example how an algorithm can be rewritten according to item 1 is given in the rest of this section.

Introduction to on-line trajectory time scaling

The aim of the on-line (i.e. at run time) trajectory scaling [19, 18] is to change the time scale of the motion along a specified path, in such a way that required torques for the motion are within admissible limits. This fits very well into the proposed system structure, since the precomputations mentioned then do not need to (or cannot) take the actual torque into account, which would lead to an unfeasible data flow. The torque available for the motion varies because e.g. friction (in the robot or to the work-piece) varies. Some spot welding robots, for instance, need to be exercised several minutes after maintenance or cold start before they can continue performing optimized motions. Instead, the control system should automatically rescale the trajectories to cope with the time varying friction.

The time scale of the nominal trajectory is changed at run time by the path velocity controller shown in Figure 5.4, in such a way that the required torque matches the torque limits. If no joint exceeds its torque limit for the nominal trajectory, the actual trajectory will be the same as the nominal. If one or several joints need more torque than allowed, the motion is slowed down but the path is kept. The basic idea behind the algorithm is to parametrize the standard robot controller in the path coordinate s. It is assumed that the path is specified as a vector function $f(s)$, the nominal trajectories are specified as two scalar functions $v_1(s) = \dot{s}$ and $v_2(s) = \ddot{s}$ specifying the velocity and the acceleration

along the path. The path velocity controller then executes the path and velocity specifications by sending the path coordinate s and its first and second time derivatives $\dot{s}$ and $\ddot{s}$ to the standard robot controller in such a way that the required torque τ is kept within the limits. To do so, the controller is written on the form $\tau = b_1\ddot{s} + b_2$, and b_1 and b_2 are fed back to the path velocity controller. Please refer to [18] or [19] for a detailed description. Figure 5.4 is basically taken from [18], but the data flow for the path specification is explicitly shown.

Rewriting of the on-line trajectory time scaling algorithm

Some restructuring will now be made in order to make the path velocity control fit into the system, still maintaining the same basic algorithm. First an explicit time scale is introduced with the motion specification. In [18], the time scale is implicitly given by the functions v_1 and v_2 which are derivatives of the path coordinates s. This means that when the functions are integrated up to the final value of s, round-off errors will be integrated and the motion will not last exactly as long as might be specified in the call to `Move`. A time correction term can easily be added to the path velocity controller, and a small percentage of overspeed can be allowed to let the controller catch up with the specified time.

The second modification is that the motion specification (including the nominal trajectories) is not any more considered as a precomputed matrix with columns s, v_1, v_2 and f, and with rows corresponding to steps along the path. A time column t was introduced above, and the nominal trajectories and the path can just as well be generated at run time based on e.g. external sensors. The columns therefore form a time signal Σ. The reason for having a dashed line for the path information in Figure 5.4 was that the motion specification was considered to be pre-planned

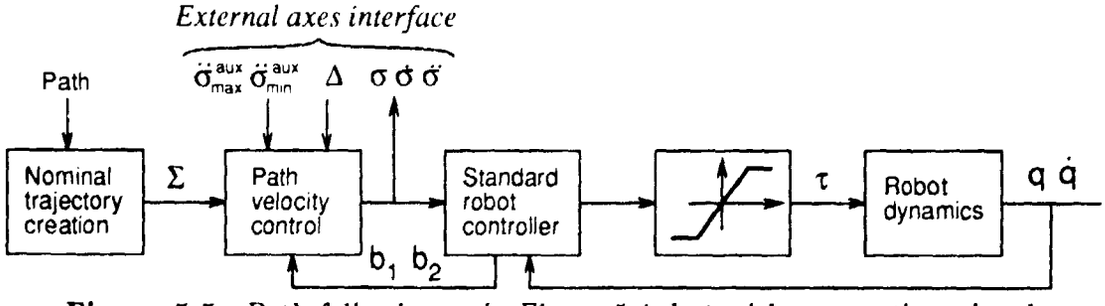

Figure 5.5 Path following as in Figure 5.4, but with proper time signals and an interface for external axes

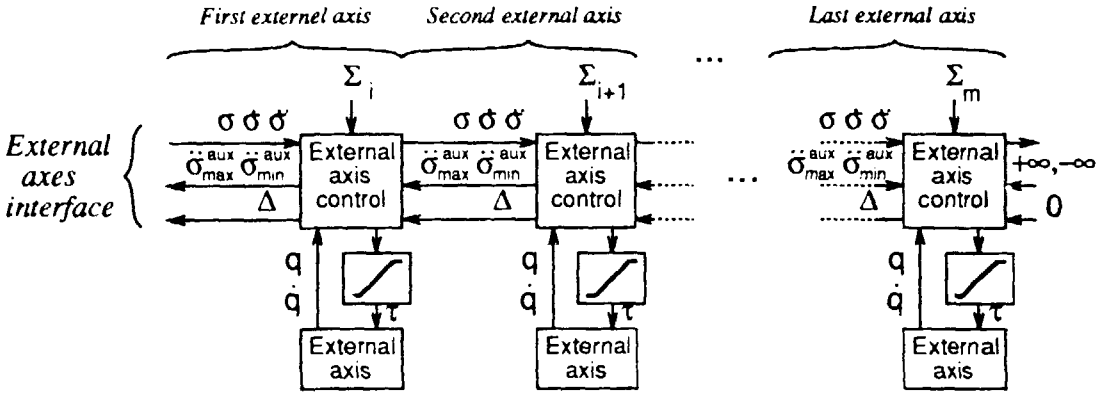

Figure 5.6 External axes can be incorporated in a "bit-slice" style in the proposed system.

nonlinear functions [18]. The signal Σ on the other hand is a proper signal as shown in figure 5.6. Precomputations are still allowed according to earlier sections, but the controller should work with signals giving a more or less constant real-time data flow allowing higher utilization of the hardware.

A third modification can now be introduced to solve the external axes control problem we were looking at. Additional motion reference signals Σ_i are supplied to each axis i. Σ_i is a vector signal consisting of the same values of t and s as sent to the arm controller, but also of the scalars v_1, v_2 and f_i specifying the nominal motion for axis i. Exactly the same values of the path coordinate reference σ and its derivatives that go to the arm controller also go to each external axis controller. Each external axis controller then computes the allowed range for $\ddot{\sigma}$ according to [18], but in stages as shown in Figure 5.6. The time lag Δ is also computed as the maximum of the input Δ and the time lag for the axis itself.

It has been shown how one algorithm can be modularized to fit into a structure suitable for industrial use with external servos connected to the robot controller. It is even straight forward to incorporate multiple arm synchronized with the same path coordinate s. It is believed that many other algorithms can be modularized in the same way, but when not possible, other features in the system (like motion in function space) will always make it possible to program and perform a motion, although a larger user or application programming effort will be needed.

5.4 The local control layer

The purpose of the *local control* layer is to support controller implementation in two ways:

- A complex multivariable arm controller can be better modularized if axis specific features are put in separate modules, as mentioned in the previous section.
- The local control software can be decentralized to each individual joint, and possibly be put in hardware together with the drive units and the internal sensor measurement system. Such hardware modules can be relocated away from the control computer to improve flexibility of the system and reduce cabling and hardware cost.

A suitable structure for the local control will be examined. The structure consists of well known basic elements, and can be viewed as an example of what the local control layer is intended to contain. Interface aspects due to hardware distribution or multiprocessor are ignored for simplicity of the example.

Assume the arm control has been designed for a rigid robot, but some of the joints on the robot have gears with backlash. It then seems like a sensible engineering principle to cope with the backlash effects, which are joint-wise, in the joint-wise control, i.e. in the local control layer. The arm control layer software (and layers above) can then be the same as for the high cost version (without backlash) of the same type of robot. This corresponds to the object oriented way of modeling shown in Figure 5.2, but what can the control structure inside the local control object look like?

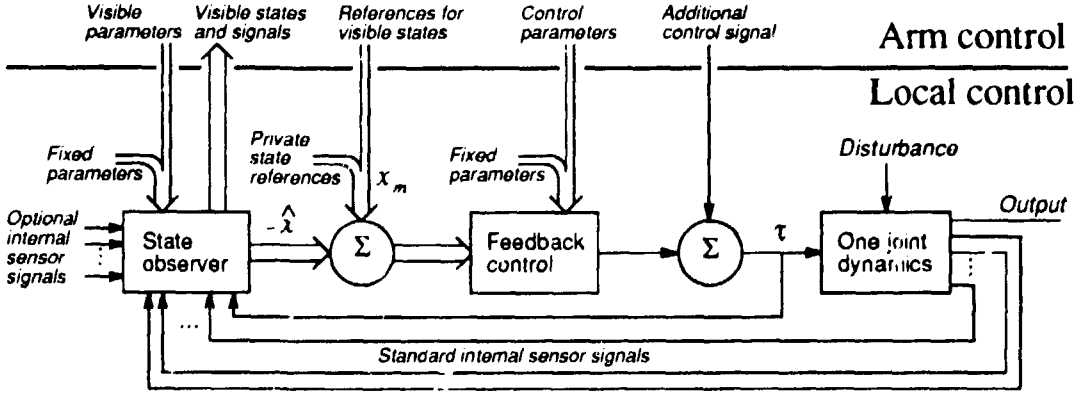

Figure 5.7 General structure of one local controller.

A general control structure of an axis controller in the local control layer is shown in Figure 5.7, and Figure 5.8 shows the same structure with a simple controller of the type commonly used today as an example. Torque limits are not included for clarity, and all signals are shown as if they where continuous time signals. Major parts of the observer and controller are normally implemented digitally, but A/D conversion is then considered to be included in the state observer, as having part of it implemented with analog circuits (e.g. together with an anti-aliasing filter) sometimes is a good way of achieving high bandwidth disturbance rejection. The D/A conversion can also be put in different ways, depending on how much of the controller that is analog, and depending on the drive system which can be purely digital or purely analog.

Both standard sensor signals and optional ones are connected to the state observer, which also includes filters etc. An optional sensor for local control can be a force sensor used for force control of a single joint. Some of the states in that block are output, i.e. $-\hat{x}$, to be used by the feedback control after the reference x_m has been added. Only the part of x_m that is of interest for the arm control can be given reference values from that layer. The same applies to the motion control layer if the joint is an external axis. Parameters are also divided into one changeable and one fixed part. The additional control signal allows a combination of the local control and the arm control, or an alternative control strategy can do all the control in the upper layer, possibly based on external sensor signals not available in the local control layer.

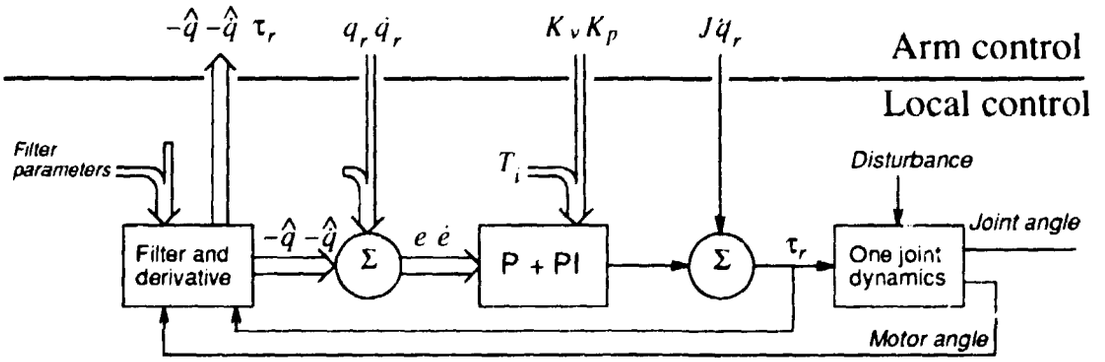

Figure 5.8 Local control of a type commonly used today with a P-type position control and a PI-type velocity control. The notation of signals should be obvious for those that are familiar with robot control, or please refer to [17].

The optional internal sensors, see Figure 5.7, are statically defined in the system. Parameters from the layer above can, however, affect how, or if, the internal optional sensors are utilized in the control. The sensor signals can as an alternative be filtered and sent up to higher layers for further processing. The reason for such a solution would be to simplify the hardware by using the ports for internal sensors also for external motion control sensors. An example with a sensor serial bus will be given in Chapter 9.

Reconsidering the example above with the rigid robot with backlash, the local control should provide signals q and $\dot{q}$ that can be used in the arm control layer as if the robot did not have backlash, and the local controller should compensate for the backlash effects in such a way that highest possible performance is achieved. The performance will of course never be as good as if the robot did not have any backlash, but note that it might be possible to get around the effects of the backlash in certain applications, i.e. the application software layer (see Chapter 4) can be extended while keeping other layers unchanged.

5.5 Summary

The design of the standard, i.e. application independent, part of the motion control system has beed developed in this chapter. A partitioning into three software layers was obtained from an object oriented view of the hardware and dynamics structure of the physical robot system. A design of the interfaces and the internal structure of the layers has also been made. The design is based on encapsulation of motion control features at compile time and flow of motion data at run-time. Both the general analysis and design of the motion control system, and the specific solutions (for incorporation of sensors, pipelining of motion commands, etc) are contributions from this chapter. The solutions to the software problems will improve the usefulness of the standard motion control as seen from levels above, but also support better ways of programming within the layers.

6

Robot Programming Concepts

The purpose of this chapter is to introduce some of the concepts that are currently used in robot programming on the user level. The presented concepts can be found in the literature or in practical use in the industry, but the intention of the collected descriptions and comments given in this chapter is to give the reader a proper view of the problems that will be tackled in Chapter 7. The main references for the material are [9] for robot programming languages, [17] for off-line aspects, and [2] for on-line features.

Section 6.1 describes some ways of classifying robot programming. A classification according to where the programming takes place is then used in Section 6.3 treating on-line programming, and in Section 6.2 covering off-line programming. Section 6.4 finally presents the current style of *world modeling*, which will be extended in Chapter 7.

6.1 Classification of programming concepts

Many preferences by robot programmers for one or the other robot programming method stem from the way different methods are combined in today's commercial systems. For example, some programmers prefer off-line programming (explained below) because some important feature is not supported by the teach-in interface they have access to. Others prefer on-line programming because their system provides an easy to use programming interface for on-line programming. Even major books in robot programming like [9] and [17] have such preferences, but some factory floor aspects (mostly to the favor of on-line programming) deserves some comments. This section tries to classify robot programming in four different ways to serve as a background when different concepts are further examined later in the chapter.

Level of abstraction

The following levels of abstraction for robot programming languages and systems are conventional (and will be reflected in the final software architecture):

- **Task-level programming** is the highest level of abstraction allowing instructions specifying what to do without specifying how the task should be performed. The task can be either a complete task like `assemble relay-X`, or the task can be a subtask like `fetch bearing from box-B`. True task-level programming does not exist yet but is an active topic of research. Two main problems are that very extensive models for the robot and its environment are needed in the system, and planning and scheduling of motions utilizing such models have not been accomplished yet[17]. Another problem which is addressed in this thesis (Chapter 10 and [38]) is that it is unfeasible to deal with low-level physical effects in high-level planning.
- **Object-level programming** also utilizes a world model consisting of objects described by e.g. frames, but a full world model with obstacles etc. is not used, as the system does not plan the paths. Instead, paths are implicitly or explicitly given by specification of relations between objects. The system computes the necessary motion to achieve the object relation, and performs the motion according to a defined strategy like straight line motions in

Cartesian space. Systems supporting a more implicit type of programming are ROBEX and RCCL. A ROBEX instruction can look like: `MOVE/box,underside,AGAINST,table,top`
Most robot programming languages described in [9] support object-level programming, usually in an explicit way like:

```
MOVE ARM TO box
CLOSE HAND
AFFIX box TO ARM
MOVE box TO table
OPEN HAND
UNFIX box FROM ARM
MOVE ARM ...
```

where the ARM object is predefined in the system, and the frames for box and table objects have to be specified. All objects make up the so called world model. The instructions AFFIX and UNFIX are used to maintain the world model when the objects are manipulated. Systems with database management of the objects, supported by the user interface, have been developed to simplify robot programming in assembly applications [40, 41].

- **Manipulator-level programming** focuses on manipulator motions rather than on the objects that are manipulated. Instructions can still refer to objects in the working space, but the objects are simply named frames and no full world model is maintained. Motions in joint space can also be used, as well as other manipulator specific parameters like acceleration and speed as a percentage of maximum for the manipulator, and type of manipulator control used.

An object oriented language would be appropriate for robot programming. The fact that object-level programming support software objects belonging to the world model does, however, not imply that object oriented programming in the computer programming sense is supported by the RPL. In fact, the object-level RPLs described in [9] does not support object oriented computer programming, and an advanced RPL like SIL has quite recently been updated for object oriented programming.

The use of manipulator-level programming, which is the most common method used in practice today, is sometimes considered as a problem in robot programming as it hinders general purpose task-level programming algorithms. The use of object-level programming maintaining a

world model has two advantages. It provides a base for sophisticated software tools for off-line and task-level programming, and the flow of program execution is better separated from the description of the environment. Separation of the world model from the program itself is particularly interesting for so called intelligent robots acting in a dynamically changing environment. A parallel process can then utilize sensor information to update the world model, which is accessed by the robot program executive.

While much of the research in robot programming aims at increasing the level of abstraction, there are industrial needs to support low level manipulator-level programming. One reason is that manipulator constraints have to be considered to achieve a high utilization of the comparably expensive mechanical device. Another reason, which we will come back to in Section 6.3, is that manipulator-level programming is a quite natural method for on-line programming.

To conclude the level of abstraction issue, different levels are suitable in different situations. A software architecture therefore has to support combined use of manipulator-level and object-level robot programming. The approach in the thesis is to use a manipulator-level style for the motion control interface as shown in Chapters 4 and 5, and to provide such features that the object level can be added according to the "separation of the world model" issue in Chapter 3. Task-level programming should then be put on top of the object-level programming.

Programming languages

Most languages used for robot programming resemble, or are, computer programming languages, which is natural since major parts of the programs consist of instructions that they have in common with computer programming. Some languages are, however, quite robot specific, and RPLs can therefore be divided into three categories [17]:

- **Specialized manipulation languages** have been designed specifically for robot programming, but are not expressive and rich enough to be considered as a general purpose language. One such language is AL (Assembly Language) which was developed at Stanford already in the 1970s, and another is the ARLA language [2] which has been developed by ABB Robotics. Many of the specialized manipulator languages, like ARLA, have evolved from simple teach-in

programming and are being extended until they resemble computer programming languages.

- **Robot library for an existing computer language** is feasible in the way that a popular and well supported computer programming language can be used also for robot programming. However, additional syntactic and semantic support for robot specific programming is sometimes desirable, and robot programs are normally interpreted while most computer languages are compiled. Examples of this approach are RCCL [24], PASRO [9], and HAL [36].
- **Robot library for a new general purpose language** well suited for robot programming can be technically feasible but requires a substantial development and maintenance effort. Examples are KAREL and SIL which are both Pascal like languages, the latter with several LISP-like features.

The main problem with robot programming languages is that each robot vendor provides a robot system that must be programmed in their specific language, and sometimes a programming environment supporting programming in that particular language has to be used for the programming. This causes significant problems for factories using robots from different robot vendors. One way to get around the problem is to use an off-line programming system in which programming is independent of the native language for each robot, and then have the system to generate code for the actual robot used. Another approach is to standardize the robot language. Such an attempt is IRDATA, which is intended as a low level language that is to be generated from higher level robot programming systems. While IRDATA is very good in its intentions, it is nowadays quite old fashioned since modern computer programming features are not supported. Other standardization activities are going on, and can be appropriate for some applications, but are of minor interest here since the new robot control and programming solutions are not fully considered.

User interface

Robot programming and supervision can be performed via a user interface comprising one or several of the following alternatives:

- Free text edit input in a usual computer programming style.

- A structured editor or programming tool supporting the programming language can be used.
- A graphical user interface, possibly with 3D world modeling.

A well designed system will use all three alternatives in an appropriate way. Fancy graphics is not always the best alternative. Use of the physical world can for instance be more appropriate than a graphical view of it in some cases.

Use of mechanical robot

The key type of classification of robot programming is by use of the mechanical robot, i.e. on-line or off-line programming as mentioned in earlier chapters. One opinion within robotics is that *how* the programming is carried out should be decoupled from *where* it takes place. The approach in this thesis is the opposite. The programming methods should be specially designed according to where the programming takes place, i.e. on line or off line, or it will be a compromise that cannot suit both cases. The importance of the on-line and off-line programming methods has therefore motivated separate descriptions of them in the next two sections.

6.2 Off-line programming

Off-line programming means that the mechanical robot is not occupied during programming, which instead takes place in a host computer. Note that even if the control computers of the robot control system are used, the programming is considered to be off-line in this thesis. Off-line programming is the subject for almost all research in robot programming. One research goal is to increase the level of abstraction towards task-level programming, while others look at suitable object-level systems which should provide a base for future task-level programming, and the trend has been that modern approaches focus on computer programming. A major problem for many advanced off-line programming systems is to get around limitations in the basic control system. One aim of the thesis is therefore to design the control system in such a way that it suits off-line programming, and thereby making available off-line programming systems more useful. Only the lower level of off-line programming, which

needs to be resigned together with the control system, is therefore considered here. The main motivations for off-line programming are:

+ The production equipment is not occupied during the programming.
+ A work cell can be designed, programmed, and simulated before it is actually built. The result of the simulation can be which type of robot that should be used, or how the equipment in the work cell should be arranged.
+ A uniform style of programming for robots of different brands can be achieved.
+ Major parts of the robot programs are often computations, which are better programmed in a computer programming style.
+ An off-line programming systems can provide an interface to task-level programming systems.

The problems with the off-line programming are:

- The accuracy of positions is deficient compared to teach-in programming.
- Simulation of the robot behavior will be inaccurate since the actual control software cannot be used because it is considered confidential by the robot manufacturer. The dynamics of the mechanics is also often unknown, but could possibly be identified.
- Transfer of the programs to the embedded controller often imposes restriction on the programming, since embedded controllers are often not flexible or open enough.
- Programming tends to be abstract since a computer representation of the robot and its environment is used.
- On-line modification of the program by the robot operator without computer programming experience is too hard when languages suited for off-line programming are used. The teach pendant user interface supported by current systems is not powerful enough.

Some of these problems have been addressed in research before, while others have not been noticed by computer science researchers and others. The current state can be summarized as:

= The accuracy problem has been approached by calibration methods, additional sensors, and by using teach-in for certain locations (refer

to [9] for integration of teach-in procedures).

= The simulation problem has been solved in earlier chapters of the thesis by designing the motion control system also for a simulation mode.

= The problems with embedded controller limitations is addressed in the thesis by designing an open and flexible system.

= The problem with abstract world models is partly solved with off-line programming systems providing a graphic user interface with 3D modeling etc. It can still be a non trivial task to define the objects in such a system, but can sometimes be derived automatically from CAD data available from the mechanical design.

= On-line modification of a program that has been generated by an off-line programming system has so far been restricted to the following cases:

1. Modification of positions via a teach pendant interface. The program itself cannot be modified, but modified positions can be transferred back to the off-line programming system.
2. Also the program expressed in the syntax used in the embedded controller can be modified, either via a structured editor (as in the ABB approach described in Section 6.3), or via a text editor requiring some familiarity with computer programming. In any case, the modified program cannot be translated back to the representation used in the vendor independent off-line programming system.

This problem will be addressed in the next chapter.

6.3 On-line programming

The major advantage with on-line programming is that it is concrete, i.e. abstract world modeling and computer simulation is not needed. The programmed motions will also be accurate since locations and frames can be defined via teach-in. Test and debugging is also often simplified by facilities like stepwise execution forward and backward, and adjustment of positions etc during robot motions while the true (i.e. not simulated)

behavior is studied. The disadvantages are basically the same as the advantages with off-line programming.

There has been a trend that embedded robot control systems have approached general purpose computer systems in complexity, i.e. robot manufacturers have tried to reinvent the computer in order to support robot programming [36]. The approach taken in the current research is to focus on the on-line programming needs for the embedded system, and otherwise make use of available tools for computer programming and off-line robot programming.

What are then the system demands for user friendly and flexible on-line programming? The answer depends, of course, on the type of user, but since on-line programming is a concrete style that can be used by the production technician with no or little experience in computer programming, that type of user is considered. A successful such system on the market is the ARLA language and programming environment from ABB Robotics. Some of its main intentions and features will now be presented to give a view of typical industrial requirements, at least for smaller work shops. The aim is then to design a system structure that fulfills these requirements and other demands for higher level off-line programming.

Interactive menu-based teach-in programming

The ABB way of programming robots can be characterized by the following:

+ All programming can be carried out via a hand held terminal which is called "programming unit". The programmer can stay close to the workpieces of interest during the programming.
+ The programming unit has a joystick for manual control of the robot, and for manual control of other equipment if feasible. Some fixed buttons for manual operation are available, e.g. open/close gripper, coordinate system selection, and a few more. Most features are accessed via function buttons, i.e. buttons that have different meaning depending on which buttons have been pressed earlier. The current meaning is displayed above each button.
+ Entering of new program instructions are done by pressing a button for positioning instruction, or another button for other instructions

(like computing, testing or branching), and then by pushing function buttons for type of instruction, its attributes, attributes to the attributes etc.

+ Program editing is activated through one fixed button and then done via function buttons.

+ Other features include start of program execution and modification of programmed positions while the robot is running.

In essence, a hand held terminal is possible since most features are accessed via function buttons, which also has the advantage that the unexperienced programmer always has only a few possibilities to choose between. No picture of the programming unit is shown here since a more modern design would probably look different with e.g. a larger graphical display, "track-ball" and pull-down menus instead of function buttons, voice input of characters and identifiers, etc. The important thing is that the robot is operated and programmed via a structured command interpreter and a structured editor, without abstract modeling of the environment since the physical environment is used.

Interactive menu-based teach-in programming has turned out to be preferable in most standard applications, but too primitive on-line programming environments is probably the reason for statements like the following which can be found in [17], where it relates to a sample application: "It should be clear that the definition of such a process thro gh 'teach by showing' techniques is probably not feasible. For example, in dealing with pallets, it is laborious to have to teach all the pallet compartment locations; it is much preferable to teach only the corner locations and then compute the others making use of the dimensions of the pallet. Further, specifying interprocess signaling and setting up parallelism using a typical teach pendant or menu-style interface is usually not possible at all. This kind of application necessitates a robot programming language approach to process description". It is true that only the corners of a pallet should be taught, but that is possible also in a powerful teach-in programming environment. Pallets are predefined in the ABB system. One attribute to the positioning instruction is PALLET, which can be selected by pushing a function button. Additional arguments will then be available from the function buttons, like pallet identifier and counters for row and column in that pallet. An object oriented RPL is preferable in a case like this, as will be further commented in Chapter 7.

6.4 World models for off-line programming

Examples of world modeling for off-line purposes can be found in [17], [9], [24] and others. The world model consists of objects which can be described by the frame concept in robotics. Apart from the base frame of an object, additional frames for different parts of the object can be given with respect to the base frame for that object. Frames can then be used explicitly in motion instructions, or they can be used implicitly by specifying equations with frames to the system which then computes the arm configuration that solves the equation. There are, however, also reasons for having more complete world models, as in the CimStation off-line programming system. The world modeling is best described by quoting the section "Objects and world models" in [17], which describes the CimStation.

Every simulated entity such as a workpiece, fixture, or link of a robot is represented by an object. The object data structure contains the model of the entity, several other attributes, plus room to add future data. A simple example of another attribute stored with an object is a label by which the object is referenced. Objects can be built from their models and stored in data base libraries for later use.

It is natural to group objects into structured objects in a tree structure. For example, an n-jointed robot is a structured object having a null model as its root, and having n + 1 subobjects ('link0' through 'linkn'). Subobjects are referred to by their path names, for example, 'puma1/link3'. The entire simulated world is a single structured object with a null model at the root named 'world'. Any time an object is

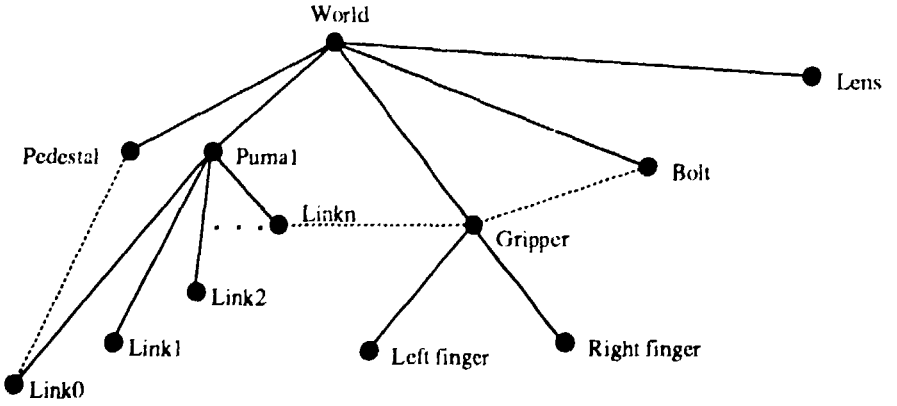

Figure 6.1 Example of the world model tree

moved, all of its descendants move with it, but a motion of a child-object does not affect its parent-object.

Figure 6.2 shows an example of a simple world model containing a robot, an end-effector, a part ('bolt'), and an object upon which the robot is mounted ('pedestal'). Dashed lines in Fig 6.2 indicate **affixments** *which are used to temporarily create a rigid connection between two objects. Affixments connect an end-effector to the final link of the robot. These affixments also enable a simulated robot to pick up a simulated bolt. In the situation shown in Fig 6.2, a command to move 'puma1/linkn', 'gripper', or 'bolt' would result in motion of the manipulator.*

In addition to the model attributes mentioned in the above quote, kinematics can be attached to structured objects, and paths (possibly generated from a CAD system) can be attached to an object.

6.5 Summary

Current methods and concepts used for robot programming have been presented in this chapter. The concepts have been classified according to level of abstraction, programming language, user interface, and according to on-line/off-line programming. The latter of these classifications is based on the physical use of the mechanical robot and its physical environment, and is therefore more fundamental than the others. Off-line programming was then described as in the literature, while the description of on-line programming is more based on products and methods. A description of world models, that are used in off-line programming today, finally concluded this introduction to the problems tackled in the next chapter.

7

The User Programming Level

The on-line and off-line robot programming concepts presented in Chapter 6 will be improved and combined in this chapter. This will lead to a programming environment where on-line and off-line programming each are supported by different software layers, and task level programming forms another layer on top of the off-line programming layer. These layers form the user programming environment as indicated in Figure 7.1, and interface to the (possibly application specific) motion control system via the executive software layer presented in Chapter 4. The layers designed in earlier chapters all reside in the embedded control system, but the

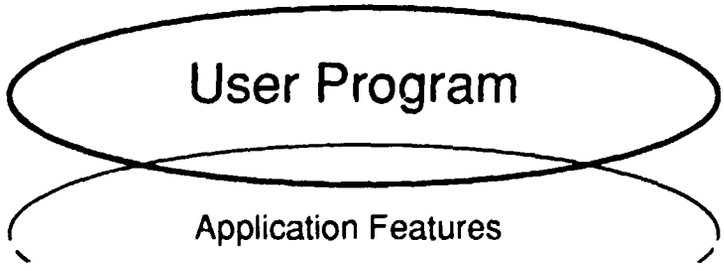

Figure 7.1 Upper half of Figure 3.3. The structure of the software for user programming will be developed in this chapter.

software components comprising the user programming level must be a mixture between host computer software and embedded system software.

The approach here is to make use of current off-line programming systems as much as possible. The enhancements that will be proposed in Section 7.1 about off-line programming should therefore be easy to implement. Section 7.2 then considers the on-line case, and shows that proper methods for on-line programming need support by both the programming environment and by the programming language to ensure consistency between states in the program and in the physical environment. Section 7.3 then presents a new way of combining the principles treated in the first two sections. The proposed principles for robot programming lead to a design of the top layers of the ORC architecture, which is finally presented in Section 7.4. Figure 7.4 in that section, showing the top of the architecture, will be included in the total architecture later in Figure 10.1.

7.1 Refined off-line programming

The intention to make use of currently available off-line programming systems does only include the most powerful systems. One such system which is both powerful and open is CimStation, see [17] and the references therein. That system uses a powerful RPL called SIL. CimStation and SIL will be used to represent off-line programming (OLP) tools in the sequel.

Solutions for lower levels presented in earlier chapters put some demands on the OLP system. It was shown in the materials handling application example that the action feature, otherwise used in the application layer, was useful also at the user programming level of the system. Generation of such actions will follow the principles shown in Figure 4.4, but the generated executable parameter should be included in the code defining the application specific motion primitive. This is believed to be possible to do with the quite open CimStation. The same applies to motions in function space presented in Section 4.3, but possible graphical support for programming of such motions is not known. The simulation features in the motion control system presented in Chapter 5, including the sensor interface designed for it, map well on the so called on-line mode in CimStation. That mode normally allows the real robot to be

run instead of the off-line simulated one. It should then be possible to use the on-line simulated robot instead, since it makes no difference for the OLP system, except for the simulated behavior of sensors which can be managed separately. Sensor behaviors can then be programmed in SIL or C++, translated to C and then to executable code, and passed to the control system as actions are.

A major problem for designers of OLP systems are to generate robot programs for, and interface to, different vendor specific robot control systems, which are often too primitive. The approach in this thesis on the other hand, is that each specific robot control system shall support or be compatible with off-line programming. The features introduced in earlier chapters seem to be compatible as explained above, but two other issues will possibly need some minor modifications of the OLP system considered. These issues will be considered in the rest of this section.

Object oriented robot programming

The object oriented programming paradigm has been selected for implementation of the system. Object-level robot programming, as described in Chapter 6, can be viewed as object oriented support concerning the physical properties of the objects within the workcell. This means that objects will have attributes reflecting their geometry, but are there cases when such objects also should have attributes corresponding to logical or computational properties? Consider a pallet for instance. Modeled in a off-line programming system, the pallet has some attributes needed to give a nice graphical view of it, some attributes that define the locations that can be used in the robot program, and some attributes that contain the states in a certain task. Attributes of the two latter types for the pallet can be:

1. The frame for one corner location of the pallet relative to a base frame, the frame for the diagonally opposite corner relative to the first corner frame, the grasping position of the part relative to its pallet location, and the number of rows and columns.
2. A representation for the state of the pallet can be the row and column for the next part.

There is a need to have the attributes classified in a way that suits later on-line access of the objects. Recall that a world model generated from

the off-line environment consists of a tree-like data structure as shown in Figure 7.1, where the nodes are objects. The attributes can be classified according to the following:

- **Model attributes** describe the graphical models of the objects. These attributes are only used by the graphical user interface for off-line programming. These attributes need not be passed to the embedded control system, except for making it possible to pass the (possibly modified) on-line program back to the off-line environment. The conclusion is that these attributes should be stored in an off-line model database, and they should be replaced in the on-line model by keys to that database. The keys cannot be used on-line, but the full world model can be retrieved later off-line.
- **Spatial attributes** describe the spatial properties needed for specification of the robot motions. Properties that can be both model and spatial attributes are considered to be spatial attributes. These attributes will be available on-line according to Section 7.2.
- **Soft attributes** are used to keep the software states associated with the object within the object. The use of free variables as RPLs used today leads to unreliable software. These attributes must also be accessible from the on-line system.

There can of course in some cases be a choice between different types of attributes, and between attributes and program flow. It is then up to the programmer to choose. It is not known how well these ideas can be supported in the CimStation.

Representation of robot programs

Robot programs, on-line or off-line, should be represented by syntax trees to allow structured and efficient access of the programs. Assume we have programmed a robot off-line and have a robot program written in the SIL language. We will then see in Section 7.3 that the program and its data need modifications before it is transferred to the robot control system. It will also be mentioned in Section 7.2 that programs can be modified in the on-line system via a structured editor. These different types of operations needed on the robot program can be efficiently implemented if the program is represented by its syntax tree.

It is not known if the SIL parser available in CimStation, or the

output from it, can be accessed by the user of the CimStation today, which would be nice for prototyping. A more general solution is to develop a (public domain) formal definition of possible nodes in syntax trees and semantic rules, and then use parsers for SIL and other off-line languages that produce syntax trees with the proper data format. A textual representation can also be agreed on, which would correspond to the IRDATA [9] standard, but updated to include modern computer programming paradigms.

7.2 Refined on-line programming

On-line programming should be possible to do via a programming unit that can be used close to the manufacturing process. It could be possible to connect a personal computer or a terminal to the embedded control system for more efficient programming in some cases, but all features must be accessible from the hand-held terminal, which will be called just a terminal below.

Some of the operations that must be possible to do have already been described in Section 6.3. A joystick and commands (i.e. buttons) for manual operation of the system are nicely implemented already in commercial systems today. Structured editors developed within computer science research show solutions that are applicable in writing and editing robot programs in an on-line environment, but such solutions have as far as known not been seriously applied.

A robot program created in an off-line programming system makes use of the world model, which is an abstraction of the real world. In on-line programming, however, the real physical world is there, which makes some simplifications possible. Frames that are not accessed in the robot program itself can first be neglected. Frames that are used by motion instructions form what is called *position equations* in RCCL [24]. A position equation can look like:

```
ARM*TOOL = Fixture*BasePlate*Beam*StartWeld
```

It is then normally only two frames of this off-line equation that are needed on-line, namely the frame `TOOL` and the frame defined by the right hand side product. These frames are called the tool frame and the goal frame. The goal frame will simply be a named location for the on-line programmer.

All motion instructions in the robot program will refer to a position equation, and each new left hand side will define another goal frame that will be passed to the on-line system, where they define the locations used in the on-line program, just as if it was programmed by teach-in. The intermediate frames of the right hand side of the equations appear physically, and are thus not required on-line. Instead of single frames, frame sequences modeling a path on the object can be retained for on-line use.

Objects with spatial features which vary with time, path coordinate, sensor signals etc. need special treatment. Assume that the off-line programming was done in the CimStation. Expressions for the these special features should then be written in the SIL language, and transferred to the robot control system according to the beginning of Section 7.1.

After these simplifications of the spatial part of the world model, only a minor part of it is directly accessible via the on-line user interface (remaining parts exist physically). However, it is probably a good solution to pass the entire spatial (i.e. not the model attributes) off-line world model to the embedded system which would only require a minor increase of memory (slow disc storage is sufficient). The complete world model can be useful for the off-line programmer calibrating the fixed parts of the world model on-line, and for support of the off-line programming style on-line, but such features should only be available under an "advanced feature" button in the on-line user interface and not used by the ordinary robot programmer.

In the case that the robot program is first created on-line, the ordinary robot programmer will define a number of taught locations which are used in the robot program. These locations can then be utilized by the off-line programmer in the definition of the off-line world model. The combination of on-line and off-line programming form a complete robot programming environment.

There is one exception to the simplifications of the world model above. It is sometimes desirable even in on-line programming to be able to specify that taught positions should be relative to some frame. For example, programming of welding of a part of a car can start with defining the base frame of the part, and by specifying that subsequent positions should be stored with values relative to the base frame of the part to be welded. A simple tree structure consisting of a root and some leaves is

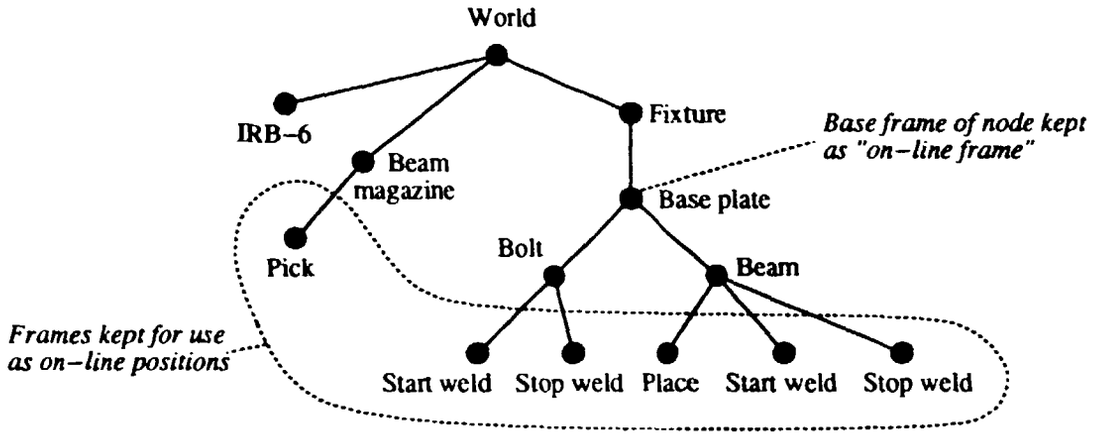

Figure 7.2 Part of off-line world model for an ABB Irb-6 robot welding a bolt on a plate, both fixed in a fixture, and welding a beam which is first fetched from a magazine and placed on the plate in the fixture.

then created on-line. The leaves of that tree still corresponds to the leaves of the off-line world model, but the root need not be the node closest to the leaves. The root can correspond to a node higher up in the structure as shown in Figure 7.2. Such a feature is available in the ARLA on-line programming environment [2], where it is called FRAME. The notion in this thesis will be *on-line frame*. On-line frames are nothing abstract, they provide a way of specifying that motions taught in one place should be possible to relocate when for instance the fixture is relocated. On-line frames can be specified during off-line programming by adding a tag to the world model node corresponding to the on-line frame, and will be returned to in the next section.

7.3 Combining off-line and on-line programming

Just like world models for on-line programming need to be simplified for on-line use, robot programs also need to be changed to suit the way the ordinary robot programmer programs the robot. Transformation of the off-line program with its world model to an on-line representation, and then transformation back again, must return the original program. The off-line world model is then reduced according to Section 7.2, and it should then be inserted as a global data declaration in the syntax tree obtained from a SIL parser. The following example about on-line frames tries to establish that the way on-line programming is performed requires

some special support from the language used. Languages designed for off-line programming do typically not meet these requirements, and special treatment of robot programs is therefore needed for combined on-line and off-line programming.

On-line frames

Any off-line generated robot program is represented by the defined syntax tree data structure. That structure needs to be transformed due to the on-line frames, which are currently used as follows in the ARLA system today.

Assume we have an off-line program with three move instructions according to:

```
moveto pos0
moveto base_plate.pos1
moveto base_plate.pos2
moveto pos3
```

This program will be translated to the following on-line program in the simplest case (note: different syntax but the same syntax tree, and positions are expressed differently because of the transformation of the world model).

```
MOVE ARM TO pos0
MOVE ARM TO base_plate_pos1
MOVE ARM TO base_plate_pos2
MOVE ARM TO pos3
```

If the `base_plate` is defined to be an on-line frame, the program corresponding to the FRAME concept in the ARLA language would look like (not in the ARLA syntax):

```
MOVE ARM TO pos0
FRAME base_plate
MOVE ARM TO pos1
MOVE ARM TO pos2
FRAME world
MOVE ARM TO pos3
```

The problem with this concept is that the FRAME instruction has a too weak syntactic and semantic support in the on-line language. It works well if the instructions are executed or programmed sequentially, but it does not work if the `FRAME base_plate` instructions is **not** executed

before the moves to pos1 or pos2 are programmed or executed. For example, the on-line programmer wants to insert an instruction **MOVE ARM TO pos1b** between the motions to the pos1 and pos2 instructions. The programmer selects the instruction **MOVE ARM TO pos1** as the current instruction by help of the structured editor. Two dangerous situations can then occur:

1. The programmer issues an "execute current instruction" to move the robot close to the pos1. This will normally result in a totally different motion that can cause significant damage. The reason is that pos1 should be relative to the **base_frame**, but the motion got executed relative to the default **world** frame (i.e. relative to the on-line frame that was currently active).

2. The programmer drives the robot directly to the new position **pos1b** manually by help of the joystick, and then issues an "insert instruction after the current one". The instruction is programmed by issuing a "move to the current location" instruction via the programming interface. The robot arm is manually moved away and the program is started from the beginning. The result will be a motion that is totally different from the one expected by the programmer. The reason is that the **pos1b** was programmed relative to the **world** frame (i.e. relative to the on-line frame that was active at the time of programming), but **pos1b** will be relative to the on-line frame **base_plate** at time of execution.

The error in both cases is that the programmer forgot to execute the **FRAME base_plate** instruction. The real problem, however, is that the system relies on external states that are not ensured to be updated. Note that even if a functional language were used, the states would still be present because of the physical states in the environment and the allowed single instruction execution by the on-line operator. The problem was not present during the off-line programming where the entire world model is separated from the program.

The problem in a more general sense is that a practical on-line programming environment must allow the programmer to select any motion instruction (and many other types of instructions) and have that instruction executed, forward or backward. On the other hand, some instructions (like the FRAME above) require initialization etc. which then must be required to be done before execution, and other instructions (like indi-

vidual instructions within a computational procedure) may be illegal to execute in this way. The full problem has not been studied yet, but the use of on-line frames can serve as an example of program transformations needed for practical robot programming.

One solution for the FRAME instruction would be to have each motion instruction tagged with the present on-line frame, or actually tagged with a list of on-line frames since there could be an hierarchy of them. This solves most of the problems in a way that is convenient for e.g. "pick-and-place" applications, where pick is performed in one frame, and place is performed in another. However, in many other applications there is a need to have regions in the program, i.e. parts of the robot's task, where motions are ensured to be expressed relative to some base frame. A welding sequence should then not get cluttered with information about the base frame, and the system should ensure that the base frame is properly set when new instructions are inserted. The following solution for the FRAME instruction is therefore also proposed.

During transformation of the world model to on-line use according to Section 7.2, all frames that have been tagged by the off-line programmer to be used as on-line frames are found by traversing the world tree. For each on-line frame, the frames used by the on-line program in that subtree are expressed with respect to the on-line frame, and the syntax tree for the off-line program to be translated to the on-line representation is traversed in the following way: A **FRAME** instruction is inserted before the first instruction that references a position that belongs to the on-line frame subtree, and an **ENDFRAME** instruction (restoring the frame that was valid before the **FRAME** instruction) is inserted after the last reference for the on-line frame. The program above will then look like:

```
MOVE ARM TO pos0
FRAME base_plate
   MOVE ARM TO pos1
   MOVE ARM TO pos2
ENDFRAME
MOVE ARM TO pos3
```

It should be possible to insert the **FRAME** clause by use of the on-line syntax-sensitive editor, just like other compound statements are created. Regardless if the **FRAME** block was created on-line or off-line originally, the program transformed back to the off-line environment looks the same.

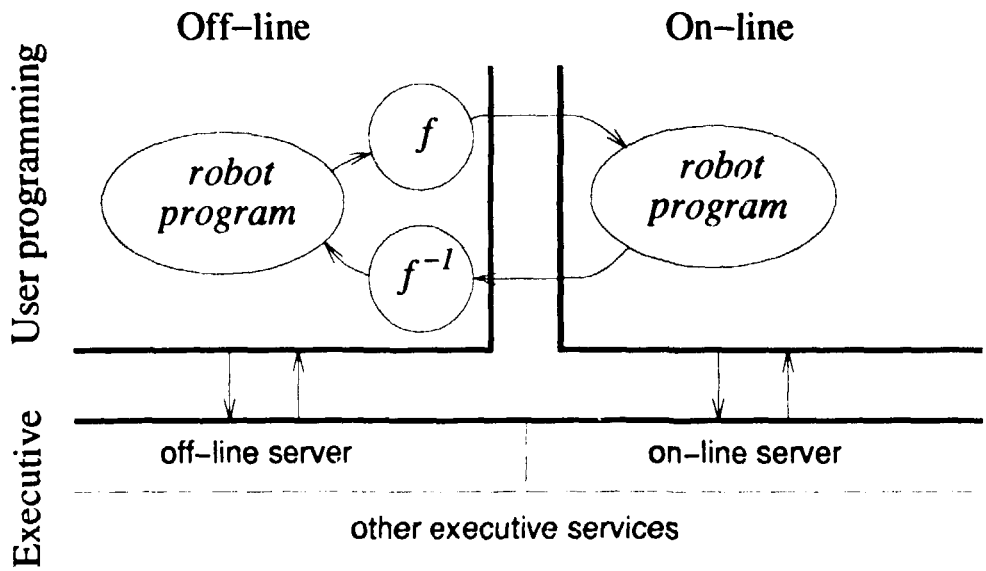

Figure 7.3 An off-line robot program needs to be transformed to an on-line representation for proper interactive on-line use. The transformation is denoted f, and its inverse f^{-1} transforms an on-line program to its off-line representation. The off-line server typically has an upper "target system handler" part running on the host computer, and a lower part linked with the embedded control system software.

Note that even if the `FRAME` statement looks like `WITH` in Pascal, there is a fundamental difference due to the syntax sensitive programming environment, and due to the coupling with external states and properties of the workcell. The transformation of robot programs, and the connections with the executive software layer, are illustrated in Figure 7.3. Investigation of other cases requiring program transformation, as well as development of the exact algorithms, is left for further research.

Tool frames

Tool frames can be subject to simplifications in the same way as goal frames are. For instance, if a left hand side of a position equation looks like `ARM*Holder*Grinder`, a `Grinder` tool frame to be used in the on-line system would be calculated (off-line) as `Holder*Grinder`, i.e. the `Holder` frame is not needed in the on-line system.

The tool frame is conceptually on the same level as the goal frame, which is explicitly given in the motion instruction. Another approach, used in the ARLA language, is to have a separate TCP instruction defining the tool frame to be used in subsequent instructions. That approach

leads to the same type of problems with external state dependencies as those for the FRAME instruction, though less dangerous since tool frames do not differ that much. The on-line programming environment will, however, provide a TCP or "tool frame" command defining the default tool frame to be used in subsequent motion instructions. The default tool frame is normally called ARM, as in the FRAME example programs above, which corresponds to no tool being mounted.

Reconsidering the `Grinder` tool frame example above, things are a little bit more involved if we want to model that the grinder gets worn down during grinding. A simple solution is to define a set of tool frames, one frame for every interval of grinding time, and have the program to keep track of grinding time and select for each new work piece which tool frame to use. A somewhat more sophisticated solution is to define one frame, but containing functions of grinding time. Motion in function space (refer to Chapter 4) would then be requested from the motion control system. An even more sophisticated solution is to define an action (refer to Chapter 4) that keeps track of the wearing by making use of the torque signals (i.e. tool forces). Such a feature is preferably encapsulated in the application software layer, where the action would also be designed to raise a user exception when the tool is worn out.

Recall from Section 4.4 that the first argument to the `Move` procedure contains the motion specification, which can now be specified to include the frames START, GOAL, TOOL, and WORLD. The START frame was needed in Chapter 5 for safety reasons in the "software pipelining" of motion computations, the GOAL frame and the TOOL frame have been discussed in this section, and the WORLD frame in the so called on-line frame from Section 7.2. Each frame contains a type flag that tells in what way the frame is represented. It is mainly the orientation part which can be represented in many ways, but a pure joint coordinate representation can also be used, particularly for joint space motions. The `Move` procedure was overloaded with respect to the first argument, i.e. with respect to the motion specification. Three different types were used to indicate Cartesian space, joint space, or function space. Note that the types used for overloading express the type of the motion, i.e. in what coordinate system the robot should move, and should not be confused with the types of the individual frames in the motion specification.

7.4 Towards a practical programming environment

The main topic in this chapter has been to combine on-line and off-line robot programming in a practical programming environment. As mentioned earlier, most of the research in robot programming aims at task level programming, or even automatic programming from CAD data, by increasing the level of abstraction and by employing AI related techniques. Such solutions are to be put on top of currently existing off-line programming system, e.g. implemented as tools in an off-line programming system. Such tools will be useful for the system developed in the thesis, since current approaches for off-line programming are joined.

The upper levels of the proposed system structure are shown in Figure 7.4. A normal "host computer"/"embedded controller" boundary have been marked in the figure, but there is nothing in the proposed structure that prevents e.g. off-line programming to be implemented in the embedded control system (which will partly be the case anyway because of the support for simulated motions in the embedded controller), or the on-line programming software to be run in the host computer. The aim of the system structure is to support the most practical way of using robots, but also to provide an open system that can be used in alternative ways.

The off-line programming software uses a library of routines comprising a "target system handler", which is the host computer part of the executive layer. These routines are normally linked together with the off-line programming software, but can also be linked directly with user programs written directly in an available computer programming language. Another way of using the system is for fixed automation, i.e. reprogramming of the equipment is less supported. Software libraries for the interfaces to the motion control layers and to the application layer should be available for the experienced user. Some application features can possibly be put in the application layer, but the executive layer in the embedded system is replaced by a single new instruction. That instruction is the entire robot program (or only servo program if all servos are supplied by the user as external axes) which is automatically called at start of the system.

An interesting aspect is that the proposed system structure with its programming tools used for robot programming can be used for control

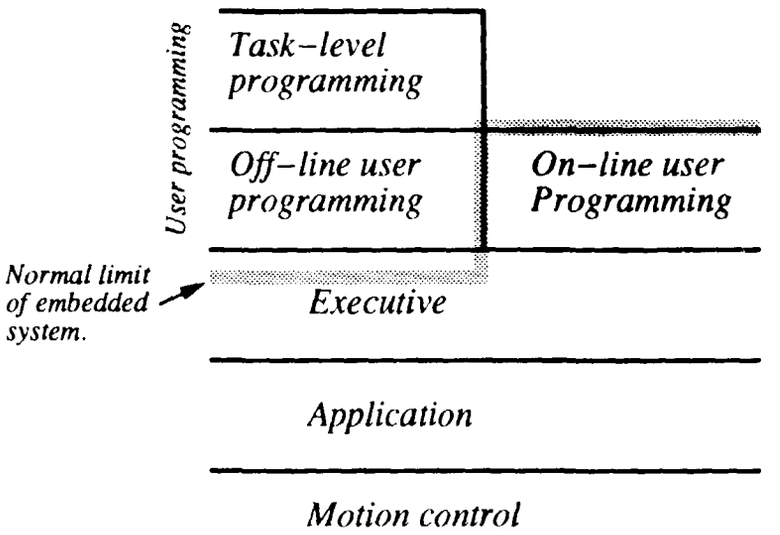

Figure 7.4 Upper layers of the proposed system structure. The *task-level programming* layer, the *off-line programming* layer, and a "target system handler" part of the executive is normally implemented in the host computer system, while lower layers are implemented in the embedded robot control system.

and programming of a production cell, or part of a production cell, which then is to be controlled by an industrial process control system (refer to Chapter 3). Robot control systems and process control systems are designed to solve different types of control problems, and offer different types of programming, but some problems are in common. For instance, how shall parallel activities in the production cell be programmed? With tasks defined in the robot programming system, or should the robot programming system only cope with robot specific features? The problem of integrating the two into an uniform approach for programming of manufacturing equipment is left for further research. The first step towards a solution is to implement the control and programming principles proposed in the thesis, which then offer a system with the flexibility needed for the further research.

8

Open Robots for Control Experiments

An important part of the experimental platform that will be presented in Chapter 9 is the commercially available robot systems that have been reconfigured to allow control experiments. The modifications will now be briefly described.

8.1 Opening up an ABB IRB-6/2 robot system

The main goal of the reconfiguration of the control system is to be able to do experiments in control, where the experiments can be done on several levels ranging from basic servo experiments to overall programming of the robot functions. The modification of the robot system was approached with the goal to keep the mechanics and the power electronics, and also as much as possible of the existing safety system, and only add parts necessary to create general interfaces to our own computers. The descriptions and drawings of the modifications included in [12], together with manuals available from ABB Robotics, form a complete description on how to do the reconfigurations.

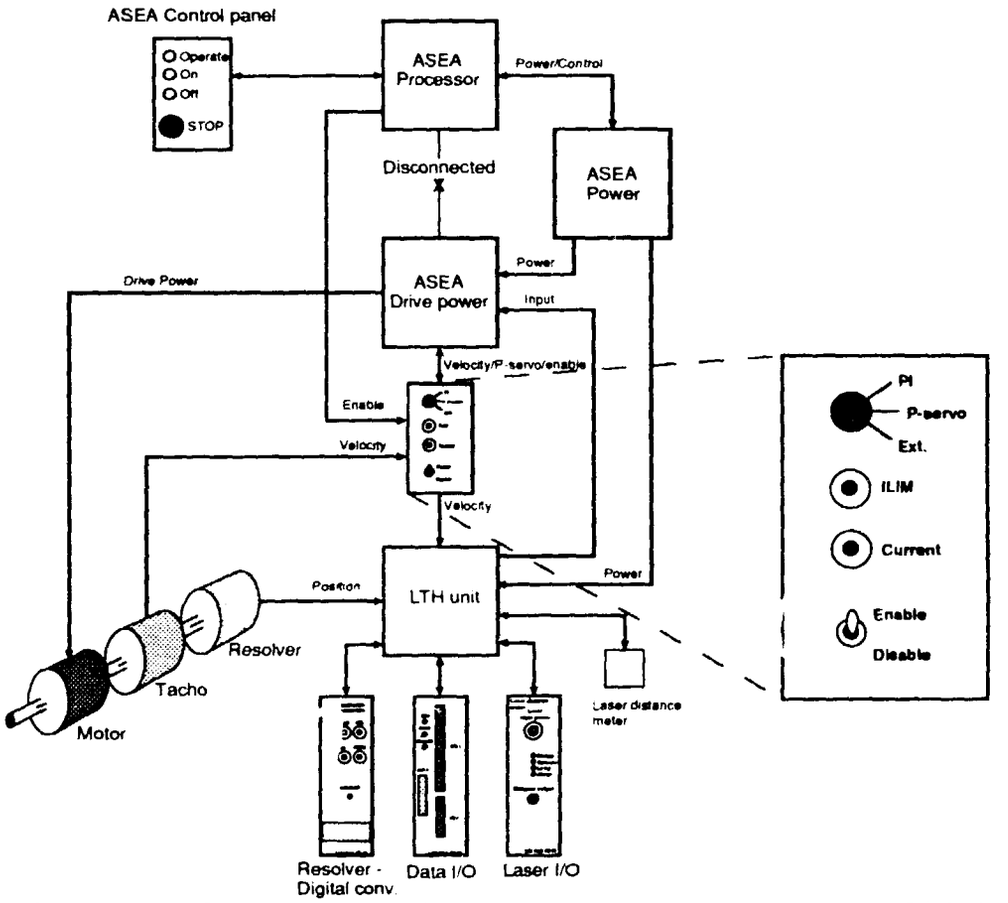

Figure 8.1 Overview of modifications. There is one **Resolver to Digital converter** (RDC) module for each joint, but only one axis is shown for clarity. The analog control signal inputs is also located at the RDC modules, where analog signals for joint speed and position are available, mainly for test purposes. The joint signals are normally accessed via the **Data I/O** module, which comprises a 16 bit (plus address and control signals) parallel resolver data bus. An interface for a laser distance sensor has also been included.

An overview of the resulting design can be seen in Figure 8.1. The original control computer is kept to make it easy to change the system back to its original shape, and to keep the changes in the safety system small. However, the original functions to take care of the measurement are not used. Instead, we have built our own sensor interface, and we have done it in a way that makes it possible to keep the sensors and cables already available inside the robot. The sensor system for measurements of robot joint angles is based on resolvers. Our solution principle is to drive and read the resolvers in our own hardware based on existing commercial chips for resolver-to-digital (R/D) conversion. The solution

principle can be seen as part of Figure 8.1, which also shows the solution principle for driving the motors. The user can select to use the original analog PI-speed-control available on the drive units by selecting the PI mode shown in the expanded part of Figure 8.1. If the user wants to implement also the *local control* layer, the Ext mode should be selected.

In conclusion, major problems were to find the right points to cut into the system while still keeping some of the functions available, and then to design and implement the additions needed. If the modifications are carefully done, the system can easily be converted back to its original shape.

8.2 Opening up an ABB IRB-2000/3 robot system

The IRB-6 robot which has been reconfigured according to the previous section is sufficient for research concerning the computer scientific problems in the current research. When it comes to implementation of the control software layers, however, a more modern 6 DOF industrial robot like the IRB-2000 from ABB Robotics is a better testbed.

The IRB-2000 is equipped with AC motors, and because we want to be able to implement all of the control layers, the interfacing to the S3 control system used has to be at a lower level compared to the IRB-6 interface. The system is simply cut at the drive unit interface, which means that not only all motion control must be implemented in our computing hardware, but also the AC motor current references must be computed in such a way that the desired torque is achieved. The AC motor control should be computed with a rate of at least 1 kHz, but the system has been designed to run at the sampling rate of 8 kHz to give some margin for future experiments. All computations are carried out in floating point, and programming in C++ also for the low level software has been made possible [33].

The same type of resolver to digital conversion as for the IRB-6 interface is used, but with the accuracy of 14 bits per motor revolution and 10 revolution counting bits in hardware. The use of commercially available R/D converters with an internal analog velocity signal and a phase locked loop makes it possible to get proper anti-alias filtering by tuning that loop. This is not possible with optical encoders, or with other types of resolver measurement principles (higher sampling frequency and

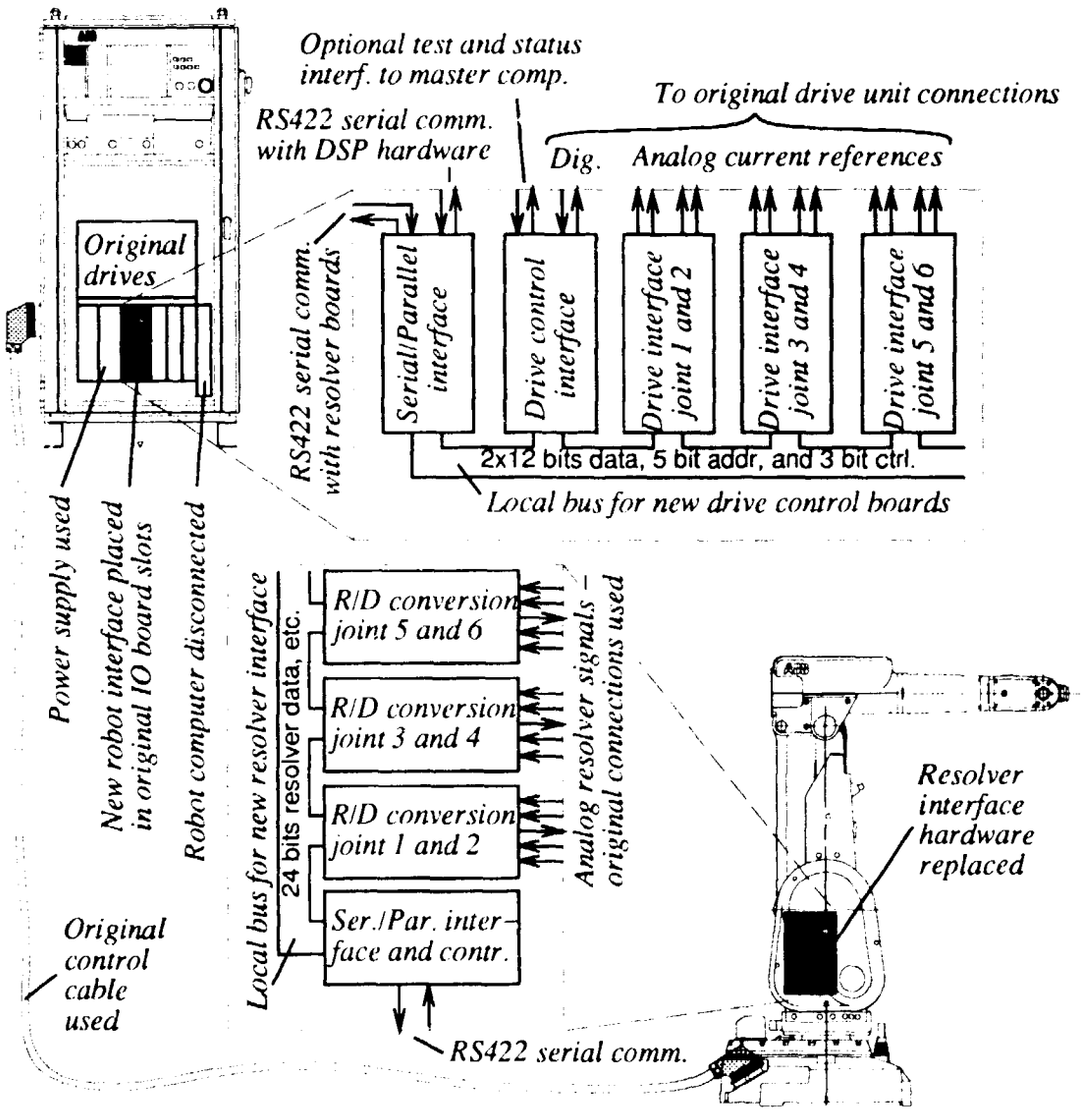

Figure 8.2 Overview of the modified IRB-2000 system. The main computer board in the control cabinet has been disconnected, and an interface to external computer and the interface boards has been added. The measurement board on the robot is also replaced to simplify the interfacing and improve performance.

the role-off of the process then have to be used instead). The 24 bit position data for the motors can simply be differentiated to get the speed; that signal has been filtered in the analog phase locked loop. The R/D conversion hardware is located on the robot which keeps the length of the wires for analog signals to a minimum.

The current references for each motor are output from the computer

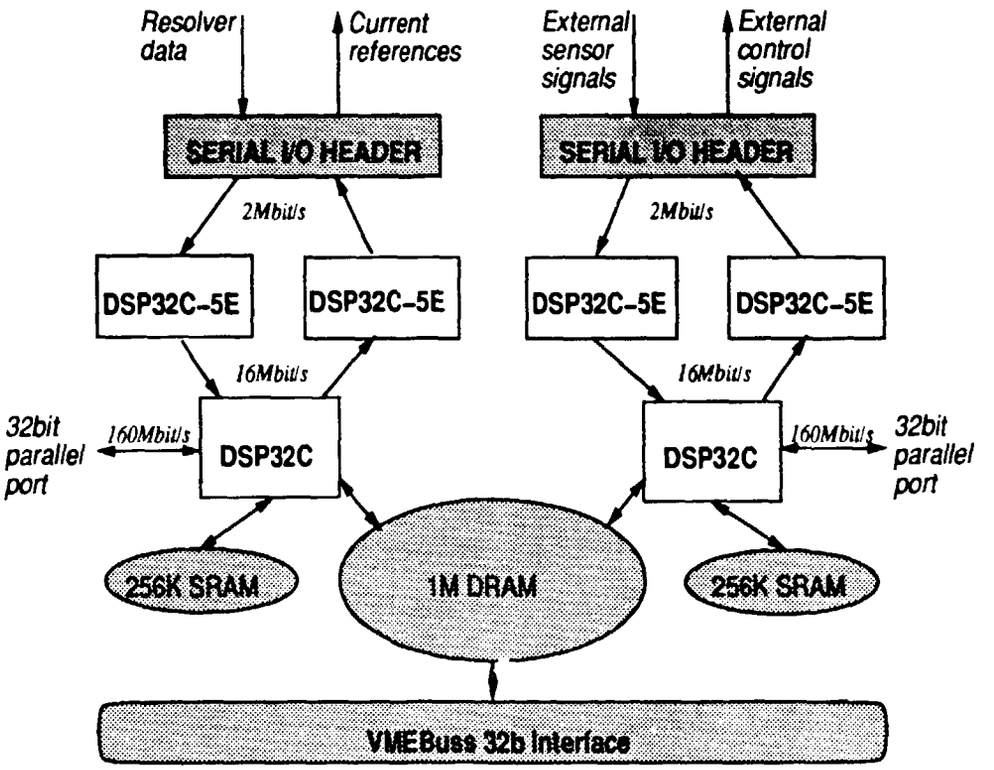

Figure 8.3 Signal processing hardware [6] with serial communication for simple interfacing with the reconfigured IRB-2000 control system. An additional parallel interface to the master computer for e.g. DMA transfer of data to and from all six DSPs via the VME bus is not shown for clarity.

hardware as two 12 bit values for two of the phases of the motor (the third phase is generated in the drive units). D/A conversion in hardware presently being built provides the current references that can be connected to the original drive units. The two 12 bit current references are transferred together as a 24 bit word from the computing hardware, i.e. the same number of bits as for the resolver data.

The interface of the modified IRB-2000 system was designed according to the following (see also Figure 8.2 and Figure 8.3).

- The robot system communicates via synchronous serial communication. A bit transfer rate of 2 Mbit/s makes it possible to transfer 32 bits of data with a sampling frequency of 8 kHz for each joint, if up to eight joints are used. The hardware protocol used allows simple interfacing to the serial ports of most DSP types, (i.e. no additional circuitry except for line drivers is required).

- The serial communication signals between the computer system (i.e. the computers that replace those of the original system) and the interface hardware, as well as between the interface hardware and the robot, are connected via IEEE-422 line drivers. The measurement board on the robot is replaced, but the original cables are used.
- Data is transferred in 32 bit words. A word going out to the robot system contains the current references, the address of the joint, a "current references to be used" bit, a "read the resolver for the addressed joint" bit, and parity. An addressed resolver sends a word containing the 24 bits position data back to the computer interface on another serial communication line.
- Additional control and status bits for the drive units are connected directly to standard IO boards on the VME bus. Those signals are used by the master computer for initialization and fault detection.

The experimental IRB-2000 control interface has been designed recently and is presently being built, and a full description of the modifications will be available. The DSP board used provides additional IO DSPs (of type -5E shown in Figure 8.3) which are mainly useful for:

IN Unpacking resolver data (extracting angle data and joint number), error checking, conversion to floating point, scaling to SI units, digital filtering from 8 kHz to a possible lower sampling rate used by the main processor, compensation for resolver and commutation offset (to get both the commutation angle for the AC motor drive and the true joint angle in spite of resolver offset), and data transfer to the main DSP of the cluster.

OUT Hold circuit implementation and interpolation schemes can be useful when the new control signals are computed less frequently. The 8 kHz rate is then used for computation of current references (from torque reference and commutation angle received from the main DSP), offset compensation, conversion to fixed point, and packing of the current references together with the joint address and other bits to be output to the robot control hardware.

9

Experimental Platform

Experimental verification is of great importance in the current research. The experimental platform will now be briefly described, including some more comments on the implementation of the principles presented in earlier chapters. Section 9.1 sketches the hardware structure used. Section 9.2 gives a brief discussion of real-time aspects, including programming of signal processors in C++. Debugging is another important aspect. Section 9.3 examines the characteristic problems with debugging of real-time control systems, and proposes a way of utilizing host computer software for evaluation of dynamic properties of the embedded control system software.

9.1 An experimental robot control system

The experimental platform for robot control consists of the reconfigured IRB-6 and IRB-2000 (under development) systems, a VME based board computer systems, and a host computer system consisting of Sun workstations and file servers, see Figure 9.1 (only the IRB-6 part of the plat-

form is described in this section for brevity). Signals from the internal sensors of the robot to the VME system go via the sensor interface described in [12] to IO boards connected to the VME bus. Apart from an analog interface, position data from the resolver measurement system is read via a 16 bit parallel interface to the resolver measurement system. The access time to internal sensor data is then 2 μs. Communication between the VME computer and the host computer system goes via ethernet as between the workstations.

The master processor in the VME computer is a M68030 with a floating point co-processor. A six DOF joystick can be connected to a serial port of any of the Motorola computer boards. For implementation of powerful control algorithms with possibly high sampling frequency, the board with six floating point signal processors used for the IRB-2000 can be used also for the IRB-6. Ports on the board make it possible to connect sensor signals directly to the DSP board.

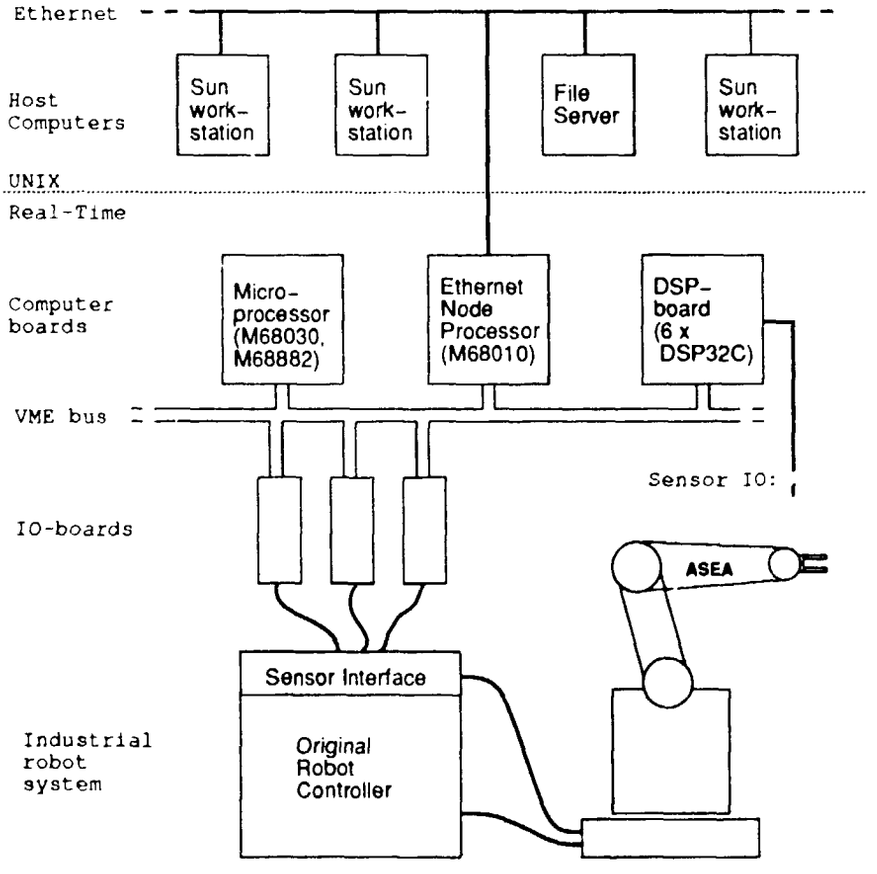

Figure 9.1 Overview of the experimental platform for one robot.

9.2 Real-time aspects

A real-time kernel developed within the department is used for the Motorola 680x0 boards [5]. No specific concurrency model is assumed, and real-time primitives are built on top of coroutine and interrupt routine facilities. Sampling rates presently supported ranges up to 1 kHz for the M680x0 boards. Programming has been done in Modula-2, but C++ and a new version of the kernel will be used in the future.

Real-time in the proposed architecture

A real-time operating system is preferably used in the embedded part of the executive layer if host communication is not used, or local file management etc. is needed. In the layers below the embedded executive, only real-time kernels are to be used. Waste of computing power on too general primitives and too heavy real-time processes is then limited. Another approach, used for the current research is to use a real-time kernel also for the embedded executive, and always have the system connected to a host computer which provides operating system services like file management via a server process in the target system. In the host computer layers, an operating system suitable for a time sharing environment is normally used. The UNIX operating system is used for the upper host computer layers in the current experimental setup, and a real-time kernel with light-weight processes is used in the embedded system. File management is provided by the host computer via computer communication facilities. The special aspects arising when implementing the lowest levels in the signal processors is the next subject.

Real-time control using DSPs

A digital signal processor (DSP) is optimized for repeated operations of the type $a = b+c*d$, which are frequently used in convolutions and matrix operations, i.e. the most common computations in signal processing and control algorithms. The processor DSP32C from AT&T has been selected for the experimental platform mainly for the reasons of simple low level programming (C-like assembly language and data directed programming [30]), compatible performance, and that the DSP, the VME board, and the software required was provided by a single source (AT&T).

It is of course desirable to use the same programming language for

DSP programming as for programming of the rest of the system, and the software objects map well on the computational blocks in the control system. The C++ compiler *Cfront* used on the Sun workstations has therefore been adapted to produce C code for the DSP, making it possible to use the object oriented paradigm also on the lowest levels of the system. Real-time primitives have been developed and encapsulated in C++ [33], making it possible to connect C++ procedures to hardware interrupts for example. DSP32C serves interrupts in a quick interrupt fashion. This means that the floating point registers, the four stage pipeline, and some other states in the DSP, is automatically stored in shadow registers during a single machine cycle (80 ns). This allows very quick interrupt routines performing for instance sensor input and buffering in less then 0.2 μs, including overhead. Clock interrupt handling resulting in no task switch will also be very rapid. The problem is, however, how to do a context switch including the floating point registers. The only shadow register accessible is the shadow register for the program counter, so the only possibility is to modify the code that will be executed after returning from the interrupt, in a way that the pipeline is emptied before the context switch. Two problems still exist; programs in ROM, and latency effects when branching due to the pipelining (one or more instructions after the branch instruction are executed before the branch occurs). Due to these drawbacks, static scheduling into interrupt driven procedures is used.

Critical parts of the algorithms sometimes have to be implemented in the C-like assembly language of DSP32C to fully utilize the computing power of the DSP. The C++ compiler has therefore been extended to allow easy and structured interfacing of inline assembly code. This feature is also useful when implementing new real-time primitives. It has been made possible to write interrupt routines for the DSPs in C++, and enable/disable of all the different interrupts have been implemented. These basic real-time primitives have been encapsulated in C++ classes but implemented in assembler and, provided by our own extensions of the C++ compilation, the code is inlined at assembly level for maximum performance. The C++ compilation can easily be adapted to other DSP brands as well, except for the feature (added by the author) that data can be passed to and from inlined assembly code. That feature relies on extensions of the underlying AT&T C compiler [7].

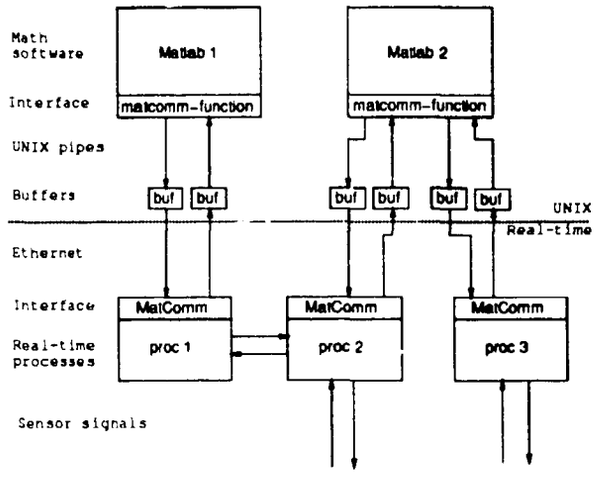

Figure 9.2 Sample connections of Matlab processes in the host computer with real-time processes in the target system.

Utilizing host computer software in real-time control

As an example of how host computer software can be utilized as a tool also for the lower levels of the proposed system structure, a Matlab interface for real-time control analysis has been developed. One or several Matlab processes running in the UNIX system can dynamically be connected to one or several real-time processes in the target system, as shown in Figure 9.2. The interface routine to Matlab is called `matcomm`, and appears as an ordinary function in Matlab even though it is implemented in C. A simple interface called `MatComm` is provided for the real-time processes in the target system. Data matrices can be sent and received, and features for managing communication errors are included. A full description will be given in [37].

9.3 Debugging of robot control systems

Debugging of robot control system software share the problems with software debugging in general, and particularly the problems with debugging of hard real-time systems. In-circuit emulators and logical analyzers are practical tools, and often necessary, when trouble-shooting also includes the hardware or when the timing of the software must be maintained. The approach taken in the current research is to use commercially available reliable computing hardware, and to manage without additional special

hardware for debugging. The approach to conventional debugging will first be described in this section, followed by an additional method for debugging of control systems. The proposed method has for some reason not been considered in the ongoing research on debugging methods within computer science. The software solutions are also appropriate to build into a commercial system.

Approach to debugging

Strongly typed compiled languages supporting abstract data types are used to ease development of reliable and efficient software. The use of C++ as a language that is supported for almost all computers (as C code is produced) makes it possible to initially test and debug the sequential and algorithmic parts of the software in a host computer environment. Standard debugging tools can then be used. The next step is to test the algorithms with the computing accuracy used in the target system. The M680x0 processors used in the experimental platform are used also in some of the host computer workstations, but the DSP software requires simulation (i.e. emulation) on the host computer. Each DSP manufacturer does provide simulation software for this purpose. The nice thing with the AT&T software is that the DSP simulator can be run under a powerful high level symbolic debugger, which also supports execution of host computer programs and also execution of other target system hardware like the M680x0. The name of the debugger is *rtpi*.

The *rtpi* debugger (Real-Time Process Inspector) [27] can be used both on the C++ code level and on assembly level. For instance, it is possible after single-step in the C++ code, to do single-step in the assembly code, and then continue on C++ level. Up to six DSPs running in parallel can be simulated from the same debugger. Rtpi also supports execution of some (or all) programs in the target system, but the device driver has not been modified yet to support the special communication with the DSP board in the target system used. Target system debugging is instead performed via a set of self-written debugging commands used from the workstation but mainly executed in the master computer (M68030) of the VME system. (Most of that software should be written anyway to provide real-time communication between the master and the DSPs.) Communication via DMA into the DSPs internal or external memory hardly affects the execution timing as DMA only imposes a

minor cycle steeling from the DSP.

Regardless of the software debugging tools available, when it comes to finding the hard errors in the control software, tracing of control signals during true real-time control is necessary. How to do that is the last topic of this chapter.

Debugging by use of control signals

The function of control software is often hard to verify by looking at conventional debugger traces of control signals. The reason is that the software implements dynamic systems which interact with the process dynamics, which implies that the time histories of the signals are of major importance. The tools required for debugging then resembles tools used in system identification or monitoring, i.e. the interface (see previous section) to Matlab and its graphics and identification tools form an important debugging tool. An error can be either a programming error or a control design error showing up only in special control cases.

The software that makes it possible to connect real-time processes in the target system with host computer software has been extended with an additional software layer providing the following features (some features not debugged yet):

- Simple export of variables to be available for logging. See the `Submit` call in the example below.
- Variable selection and logging parameters selectable from the host.
- The control loop only needs to do an `Update` call each sample.
- Continuous trace (no samples lost) mode and different trig conditions implemented. Both soft triggs from the host computer and hard triggs defined as procedure parameters possibly accessing hardware IO can be used. Post-triggering and other features normally found on logic analyzers implemented.
- Code optimized to keep the peak CPU load to a minimum and to keep control timing undisturbed.

This type of software is appropriate even for a commercial system, which would possibly have a connection for a memory and communication board not to make the base system more expensive. The control code only needs to call the `Submit` and `Update` member functions in class `Logger` according to the following:

```
#include Logger.h;
#define LOOP for(;;)
...
  float s, PathErr;
  Logger Plot;
  ...
  Plot.Submit(s,       "Path coord");
  Plot.Submit(PathErr, "Grinding dev.");
  ...
  LOOP {
     // Control...
     Plot.Update();
  };
```

where the strings supplied in the `Submit` call define the name used at the host computer side. The system clock time is automatically submitted.

An industrial aspect is that it should be possible for control engineers to analyze certain control problems, e.g. with sensor based control, without visiting the site of the application. Together with some computer communication facilities, the experimental software developed for simple debugging can be seen as a prototype for solving also this future industrial problem.

10

The Software Architecture

The internal structure for each of the three main control system levels programming, application, and control according to Chapter 3 have been presented in Chapters 4, 5, and 7 respectively. These structures together form the complete proposed architecture for industrial robot control, which is described in Section 10.1. Section 10.2 relates the proposed solution to other approaches.

10.1 The Open Robot Control (ORC) architecture

Different parts of a total system structure for robot control systems have been presented in earlier Chapters, and a possible hardware configuration was given in Chapters 8 and 9. The different parts can now be put together into an architecture which is called the **Open Robot Control (ORC)** architecture, which is shown in figure 10.1. Recall from the development in earlier chapters that the architecture has been designed for connection of external sensors to many of the software layers belonging to the embedded control system, and the software solutions are quite

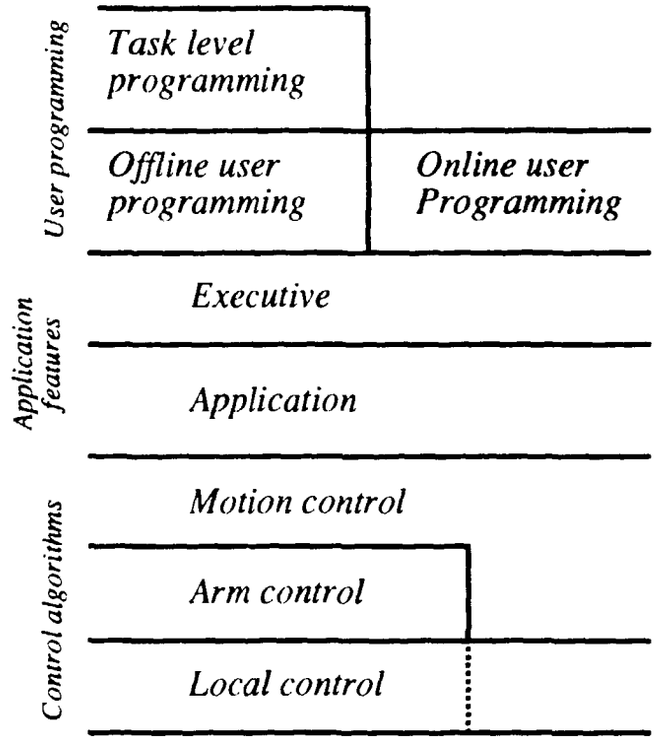

Figure 10.1 The Open Robot Control (ORC) architecture.

different in each of the layers. The ability to include sensors in the ORC architecture therefore adds an extra dimension to it. The same thing applies to the software tools. It was shown in Chapter 9 how software available on the host computer could be connected to the embedded control system. Such software can, for instance, be used for debugging of low level control, or for analysis of external sensor based control loops defined by the application engineer. Software tools can also be used for computational support as was described in the solution of the arc welding application example in Chapter 4.

The main properties of the ORC structure is summerized in Table 10.1, which can be compared with the outline in Table 3.1. A system designed according to ORC will be an open system where more advanced users have access to more advanced features and reprogramming facilities. The most important layer is perhaps the application layer, which provides a new degree of freedom for configuration of robot systems for certain applications. The software layers also encapsulates different properties of the robot control system, which bounds the software changes when the robot, the application, or the programming facilities changes. A very

ORC layer	Encapsulates	Typical programmer	Typ. exec.	
Task level programming	Automatic programming from design data	Implicit from work–piece design (not possible today)	Interpreted	Host comp.
Offline user programming	Programming without use of robot	Robot programmer with computer experience		
Online user programming	Programming with use of robot	Production engineer or robot operator		Embedded system
Executive	RPL and control system interface	Computer programmer and exp. application engineer	Compiled	
Application	Application specific motion control	Experienced application engineer		
Motion control	General control of workcell motions	Control engineer		
Arm control	Arm specific motion control	Robot control engineer		
Local control	Control suitable for impl. in distributed hardware	Servo control engineer		

Table 10.1 Users and properties of software layers in the ORC architecture.

hard constraint in the design of the architecture has been computing efficiency. This means both that powerful hardware is well utilized in hard applications, and that simple control or programming principles that is feasible in some simple application allows a minimum of hardware. In other words, there is no big overhead connected with features that are not used. The real-time demands for the different layers span from no real-time in the top layer to hard real-time in the bottom layer.

10.2 Discussion

A system structure like ORC can be compared with e.g. the OSI, or more specifically the MAP, layered system structure for computer and machinery communication in the sense that the structures try to encap-

sulate low level details from higher level aspects. However, a technical comparison is not feasible since it is hard to compare robot control with computer communication, and also because ORC is primarily a structure for the functionality and programming of the system, rather than an attempt to specifying the details in each software level interface. It would be a mistake to currently develop a detailed standard for e.g. the language to be used at the user programming level. The primary goal is to use a structure of the control system that aids in making robots more applicable, particularly for new applications that might put new requirements on the robot programming language.

The central part of ORC is the application software layer which in the simple case provides a set of robot functions. Seen from the outside it consists of data and procedures. The functions may even be nicely encapsulated like in the RIPE (Robot Independent Programming Environment) system [35]. When considering the internal implementation, why is an architecture proposed? One could instead consider the possibility of trying to find a complete set of well-defined procedures, i.e., some form of generic set of robot functions. These functions could then form a hard shell (no reason to get inside) library, where the internal implementation is hidden and optimized. In fact, several of the currently available robot control systems seem to have this structure. This is a major reason for the difficulties to slightly modify a function, to include a new sensor etc. Instead, the internal implementation should be easily accessed. An architecture makes this access possible and hopefully efficient.

The field were system architectures have been most extensively studied is within control of vehicles and telerobots. The most significant difference between such robots and more conventional industrial robots is the ability to cope with unforeseen changes in the environments of the robot. This is important also for more autonomous industrial robots that should be possible to use in fully automated, but still flexible, factories. The ORC architecture is designed for robots that shall be useful together with people, but a comparison with architectures for more autonomous robots is appropriate for the discussion.

Architectures for robots in changing environments

A major problem for robots in space, and for autonomous robots and vehicles in general, is to maintain a model of the dynamically changing environment and to replan (in real-time) the motions according to environmental changes. A comparably large number of sensors is then required, as well as advanced sensory processing and world model updating. Special manipulators can also be required to position sensors in places where the unknown parts of the environment is best observed. The need for an architecture to support system design, control and programming have been noticed, as for industrial robots, but the problems with a changing environment is very much reflected in these architectures. Differences in programming methods and applications are other reasons why these architectures do not provide much help for practical industrial robot applications.

The most well known architecture is the "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)" [4], which has some background in automated manufacturing research, and show some similarities with the ORC architecture. The NASREM structure is shown in Figure 10.2. The horizontal partitioning into sensory processing, world modeling, and task decomposition is not opposed to ORC where the software very well can be partitioned in this way, but it has not been found necessary to include this aspect in the ORC structure. On the lowest level for instance [21], sensory processing (G_1) means reading and filtering the internal sensors, world modeling (M_1) contains the kinematic and dynamic model of the robot, and the task decomposition (H_1) includes the control algorithm itself. The vertical partitioning into hierarchical layers roughly corresponds to the ORC layers in the following way:

- The local control and arm control layers in ORC corresponds to level 1 in NASREM.
- The motion control, application, and executive layers in ORC approximately corresponds to the primitive and E-move levels in NASREM. The executive layer have to be somewhat sophisticated to interface to the task level in NASREM.
- All programming layers in ORC corresponds to the TASK level in NASREM.

- Higher levels of NASREM, which primarily deals with planning and job assignment, are not explicitly represented in ORC but features that are relevant for programming of a work cell are included in the task level programming layer in ORC.

Clearly, support for application oriented programming of industrial robots (as defined in the thesis) is not supported by NASREM. In fact it is believed to be supported only by the ORC structure. Major parts of NASREM instead tries to support high level planning, job assignment for many cooperating robots and devices, and real-time knowledge-based solutions, while ORC supports (without specifying any particular solutions) such principles that are relevant for the work cell in the task programming layer. Complete factory automation planning systems etc

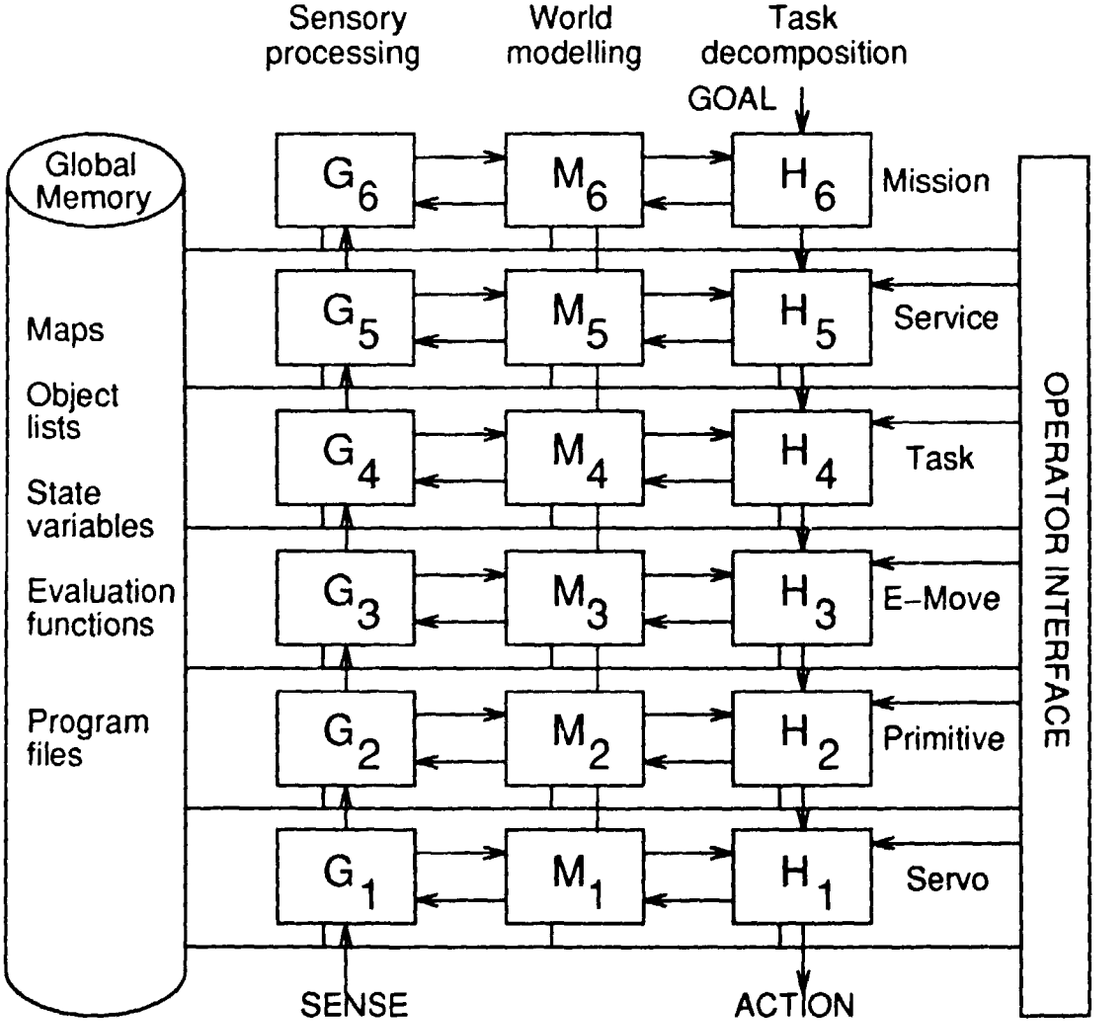

Figure 10.2 The NASREM control system architecture for telerobots

are not supported in any other way than making the local control system flexible and powerful enough to fit into such systems, as explained in the section about task level programming and executive agents below.

Abstractions and architectures

An interesting question about any control system architecture is what type of abstractions or hierarchies it is built upon. From software engineering we are used to concepts like data abstraction, functional programming, object oriented programming, etc. Such concepts have mainly been introduced to cope with software complexity. Considering a complete multi-layered real-time control system with hard real-time problems in lower layers and demands for AI related features in the top layer, it is clear that first of all we need the concepts from software engineering to be able to implement such a system in a practical way, but more concepts are needed in hierarchical real-time architectures like ORC or NASREM. These concepts are a number of hierarchies which have been proposed by different researchers, and collected in [39] according to the following:

- *Frequency hierarchies* [4, 43] are based on the common real-time principle that the real-time processes in a lower layer run more frequently than those in the next higher layer of the system.
- *Data abstraction hierarchies* [29] are closely related to the data abstraction ideas in e.g. object oriented programming. A lower software layer provides an abstract machine for the adjacent higher layer.
- *Representational abstraction hierarchies* [4, 43] is normally used by AI people as the method of building an abstraction by suppressing or ignoring information.
- *Deresolution hierarchies* [34, 31] is often used in motion planning. Two layers can do functionally the same computations, but with a higher resolution on the lower level. Deresolution is related to the previous hierarchies, but is not the same.
- *Subsystem hierarchies* [29, 4] are based on grouping the control of subsystems, e.g. control of individual joints, to control of the composed system, e.g. the arm driven by the joints. This approach is often combined with data abstraction.

- *Competence hierarchies* [14] are built by composing simple behaviors of lower layers into more competent behaviors on a higher level of the system. For example, vibrations in a robot gripper caused by a simplified control, can be utilized on a higher level for an advanced "non-stiction" assembly operation.
- *Temporal extent hierarchies* [28, 43] are designed so that higher levels manage behaviors over longer stretches of time. Note that the higher level considering a longer time period can still be computed more frequently then lower levels.

The ORC architecture developed in the thesis is based on, or supports, all of these abstraction hierarchies. The frequency hierarchy is one of the fundamental ones, and it is even more for the NASREM structure, which maps well on real-time implementation and computing efficiency. However, while a frequency hierarchy is suitable for the pure algorithmic parts of the system, many other features are hard to implement in a pure frequency hierarchy. An architecture shall support practical use of the system, which normally implies that several other hierarchies must be used, but a part of the software that belongs to a higher level in one of these other hierarchies might very well need fast enactment.

One example of such a contradictory demand between the frequency hierarchy approach and the competence hierarchy approach is the solution of the deburring application, please refer to Section 2.1. The detection and handling of the casting bulge in that case should be defined in the application layer of ORC, while the execution of that code has to be performed in the lower motion control layer. The contradiction between different architectural approaches needed for industrial robots have been solved in ORC by introducing *actions* (see Section 4.4). It has then been possible to find suitable abstractions for all the interfaces between adjacent software layers.

An attempt to a uniform approach for the design of software architectures is behavior abstraction [39, 15], but a further discussion is outside the scope of this thesis.

Task level programming and executive agents

Considering the issue how the ORC architecture can support planning and task level programming, one of the few systems that have proved to work is considered. The AI laboratory at University of Edinburgh has developed a complete assembly system called SOMASS [32, 23]. The system plans and executes assemblies in a special type of blocks world, namely the Soma world. SOMASS has been demonstrated to work well despite a number of possible sources of failure. The uncertainties that may cause assembly failure include part tolerance, physical characteristics such as friction or stiction and the like.

The following quote from [23] is central for the purposes of the thesis: "The interesting point about SOMASS, for our purposes, is that it takes a particular, and somewhat unusual, approach to the activity orchestration problem. The conventional view in assembly robotics has tended to be that the planning component of the system should anticipate and deal with various possible reasons for assembly failure, this has, in practice, proved computationally and intellectually intractable. SOMASS, on the other hand, takes the position that the planner should concentrate on those aspects of the problem that can tractably be expressed in symbolic form, leaving the execution agent to cope with the specifically manipulative difficulties of the assembly problem. Since the agent is hand-crafted, most of the consequences of the uncertainties in the parts and their manipulation are dealt with by the human programmer who has years of experience of object manipulation to call on when diagnosing and repairing failures in the tacit skills of the executive agent".

The fundamental standpoint in the above quote is shared. However, the main interest here is not planning or activity orchestration, but rather the executive agent or what is called the software layer for application programming. The hand crafted executive agent embodies the skill of the human operator and describes his knowledge of the physical situation and its uncertainties. The research goal here is this type of programming, to structure it and thus to improve efficiency. This also illustrates how research in structures for physical robot functions provide a link to higher level ideas like planning.

11

Conclusions

An application oriented view of programming and control of industrial robots has lead to a layered control system software. The layers are defined by the proposed architecture called ORC (Open Robot Control). The architecture defines different levels of programming for different types of users. The upper layers of ORC provide so called user level programming, i.e. programming for the ordinary production engineer or robot operator. The middle layers make it possible for the advanced user to introduce application specific motion control strategies, and also to modify the way the ordinary user may access any of the available primitives. The lower layers for implementation of the motion control are designed to support hardware utilization at run-time, combined with flexible reconfiguration for industrial applications.

It is important to structure the control system in layers that look independent from a user point of view. It should be efficient to work locally in one layer. The different users should of course have different access privileges. Efficient use of robots requires tight connections between the layers. The underlying implementation therefore requires special software solutions, which have to be designed considering typical hardware and software constraints.

The principles of ORC can be summarized as follows:

- It should be possible to deal with application examples of the type listed in Chapter 2 efficiently. In particular, it should be possible to implement application specific motion control strategies, as pointed out in the examples.
- The executive software layer defines the system interface for the ordinary user. The set of primitives that are available, e.g. `move`-procedures, are separated from the way the primitives can be accessed, e.g. by means of a robot programming language.
- The requirements on the programming language and user interface for the on-line and off-line programming cases are different and sometimes also contradictory. This is managed by using separate layers for these types of user programming. Robot programs, including the world model, typically need different representations, but they should still be possible to transfer between the on-line and off-line programming environments.
- The motion control level is divided into three different layers to support controller implementation, hardware distribution, and incorporation of external joints.

The problems tackled are not well suited for a complete theoretical analysis, i.e. experimental verification is needed. A commercial robot control systems has therefore been reconfigured for control experiments. The experimental platform includes robots, signal processors and micro processors in a VME system for real-time control, and Unix based workstations as host computers. The software tools developed for object oriented real-time programming of signal processors, as well as the tools for analysis and debugging of embedded control systems by use of host computer software, seems to be useful also outside robotics research. Turning back to ORC, the following points describe some aspects of the implementation.

- The requirement to support different programming levels implies that methods need to be passed as parameters between different software layers. Such methods can be algorithmic specifications of control parameters, application specific control strategies, or treatment of external sensor signals. Method passing can be implemented efficiently even in the multiprocessor case by the introduced actions, which are relocatable executable pieces of code generated by a special cross-compilation procedure.

- Transfer of robot programs between integrated, but different, environments for on-line and off-line programming typically requires transformation of programs expressed in different languages. The transformation also involves the world model data objects. The attributes of such objects were therefore classified into model attributes, spatial attributes, and soft attributes.
- Some algorithms requiring extensive computations, e.g. for optimization of motions, have previously only been possible to apply in an off-line style to precompute data for motions known in advance. In many practical cases, however, motions depend on internal and external signals, and they are sometimes known only a short time in advance or not known in advance at all. The proposed classification of motion commands with respect to sensor dependency, and the possibility to have motion commands interpreted befored they are performed, makes a motion command pipeline possible. This feature, normally not exposed to the ordinary user, makes more powerful algorithms applicable even with limited computing power.
- It is known that use of the true control system software and a dynamic model of the robot can be utilized to support simulation for off-line programming. This is well encapsulated in the object oriented design of the system. A step forward is that the solution is integrated with the motion command pipeline, external sensors, and motions depending on sensor signals.
- Sensor software interfaces have been designed for several software layers in the ORC architecture. The sensor interface in the motion control system requires three different sensor behaviors for normal control, for pipelining of motion commands, and for simulation of robot motions by use of the embedded system. The object oriented design provides a simple way for the user to specify these behaviors and incorporate new types of sensors.

In conclusion, the proposed architecture provides flexibility and efficiency in a combination that improves the applicability of industrial robots. Flexible user programming is based on a combination of on-line and off-line programming, integrated via transformation of robot programs. Flexible adaptation to present and future application demands

is achieved by having the system open and programmable at different levels during configuration of the system. Efficiency is then provided at run-time by a tight coupling between user level commands and low level control.

12

References

[1] V. R. A. and T. N. MUDGE. "Robots are (nothing more than) abstract data types." *Robotics Research: The Next Five Years and Beyond*, August, August 1984.

[2] ABB ROBOTICS. *Programming Manual Robot Control System S3*, 1991. Article number 6397 013-129.

[3] S. AHMAD. "Constrained motion (force/position) control of flexible joint robots." In *Proceedings of the 30th Conference on Decision and Control*, 1991.

[4] S. A. ALBUS, H. G. MCCAIN, and R. LUMIA. "NASA/NBS standard reference model for telerobot control system architecture (NASREM)." Technical Report Technical Note 1235, U.S. Nation Bureau of Standards, 1987.

[5] L. ANDERSSON and A. BLOMDELL. "A real-time programming environment and a real-time kernel." In *National Swedish Symposium on Real-Time Systems*, 1991.

[6] AT&T BELL LABORATORIES. *The Owners's Guide To SURFboard, Yet Another VMEbus DSP Module*, 1988.

[7] THE AT&T DOCUMENTATION MANAGEMENT ORGANIZATION. *WE DSP32 and DSP32C C Language Compiler User Manual*, 1988.

[8] A. BENVENISTE and G. BERRY. "The synchronous approach to reactive and real-time systems." *Proceedings of the IEEE*, **79:9**, 1991.

[9] C. BLUME and W. JAKOB. *Programming Languages for Industrial Robots.* Springer-Verlag, 1986.

[10] J. BOBROW, S. DUBOWSKY, and J. GIBSON. "Time-optimal control of robotic manipulators along specified paths." *International Journal of Robotics Research*, **4:3**, pp. 3–17, 1985.

[11] M. BOTTAZZI and C. SALATI. "A hierarchical approach to systems with heterogeneous real-time requirements." *The Journal of Real-Time Systems*, **No 3**, 1991.

[12] R. BRAUN, L. NIELSEN, and K. NILSSON. "Reconfiguring an ASEA IRB-6 robot system for control experiments." Technical Report TFRT-7465, Department of Automatic Control, Lund Institute of Technology, 1990.

[13] R. W. BROCKET. "On the computer control of movement." In *IEEE International Conference on Robotics and Automation*, 1988.

[14] R. BROOKS. "A layered control system for a mobile robot." *IEEE Journal of Robotics and Automation*, **2:1**, pp. 14–23, 1986.

[15] R. CHATILA and S. Y. HARMON, Eds. *Workshop on architectures for intelligent control systems.* IEEE International Conference on Robotics and Automation, 1992.

[16] I. COX and N. GEHANI. "Exception handling in robotics." *IEEE Computer*, March, March 1989.

[17] J. J. CRAIG. *Introduction to robotics: mechanics and control.* Addison-Wesley, second edition, 1989.

[18] O. DAHL. *Path Constrained Robot Control.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1992. ISRN LUTFD2/TFRT--1038--SE.

[19] O. DAHL and L. NIELSEN. "Torque limited path following by online trajectory time scaling." *IEEE Transactions on Robotics and Automation*, **6:5**, pp. 554–561, 1990.

[20] V. J. ENG. "Modal segments for a peg insertion task." Technical report, Harvard Robotics Laboratory, 1988.

[21] J. C. FIALA, R. LUMIA, and J. S. ALBUS. "Servo level algorithms for the NASREM telerobot control system architecture." In *SPIE Space Station Automation III*, volume 851, 1987.

[22] GRÖNDAHL ET.AL. *IVF-resultat 90609.* Mekanförbundet, Stockholm, 1990. Picture from Electrolux Industrial Systems.

[23] J. HALLAM. "Autonomous robots: from dream to reality." In *Proceedings from 'Robotikdagar'*, Linköping, Sweden, May 1991.

[24] V. HAYWARD and R. P. PAUL. "Robot manipulator control under unix, rccl: A robot control "C" library." *The International Journal of Robotics Research*, **5:4**, 1986.

[25] J. M. HOLLERBACH. "A survey of kinematic calibration." *Robotics Review*, pp. 207–242, 1989.

[26] K. J. ÅSTRÖM AND B. WITTENMARK. *Adaptive Control.* Addison Wesley, 1989.

[27] D. KAPILOW. *Pi User's Manual.* AT&T Bell Laboratories, August 1990.

[28] W. KOHN and T. SKILLMAN. "Hierarchical control systems for autonomous space robots." In *Proc AIAA Conf on Guidance, Navigation and Control*, pp. 382–390, 1988.

[29] V. KUMAR and K. WALDRON. "Adaptive gait control for a walking robot." *Journal of Robotic Systems*, **6:1**, pp. 46–76, 1989.

[30] E. A. LEE. "Programmable DSP architectures: Part II." *IEEE ASSP Magazine*, January, January 1989.

[31] R. LUHRS and A. NOWICKI. "Real-time dynamic planning for autonomous vehicles." In *Proc DARPA Knowledge Based Planning Workshop, Austin TX*, pp. 19–1ff, 1987.

[32] C. MALCOLM and T. SMITHERS. "Programming assembly robots in terms of task achieving behavioural modules: First experimental result." Technical report, Department of Artificial Intelligence, University of Edinburg, 1988. DAI Research Paper no. 410.

[33] U. MATTSSON. "Digital reglering med signalprocessorer och C++," (Digital control using signal processors and C++). Technical Report TFRT-5434, Department of Automatic Control, Lund Institute of Technology, 1991. Swedish.

[34] A. MEYSTEL. "Planning in a hierarchical nested autonomous control system." In *Vol 727, 'Mobile Robots'*, pp. 42–76. SPIE, 1986.

[35] D. J. MILLER and R. C. LENNOX. "An object oriented environment for robot system architectures." In *IEEE International Conference on Robotics and Automation*, 1990.

[36] M. S. MUJTABA and W. D. FISHER. "HAL, a work station based intelligent robot control environment." *unknown journal*, pp. 18–63 - 18–79, 1986.

[37] K. NILSSON. "A Matlab interface to real-time control systems." Technical report, Department of Automatic Control, Lund Institute of Technology, To appear.

[38] K. NILSSON and L. NIELSEN. "An architecture for application oriented robot programming." In *IEEE International Conference on Robotics and Automation*, 1992.

[39] M. SCHOPPERS. "Issues for real time knowledge based control systems design.". Unpublished, private correspondence, 1991.

[40] B. SHIMANO, C. GESCHKE, R. GOLDMAN, C. SPALDING, and D. SCARBOROUGH. "AIM: A task level control system for assembly." *ROBOTS 11*, pp. 45–62, 1987.

[41] M. G. SMITH. "An environment for more easily programming a robot." In *IEEE International Conference on Robotics and Automation*, pp. 10–16, 1992.

[42] B. STROUSTRUP. *The C++ programming language.* Addison Welsey, second edition, 1991.

[43] W. K. T. SKILLMAN and R. GRAHAM. "Hierarchical control of a mobile robot with a blackboard based system." In *Proc SPIE Conf on Mobile Robots III*, Nov 1988.

[44] VDI-VERLAG, DÜSSELDORF. *IRDATA, VDI-Richtlinie 2863, Blatt 1.*, 1986.

A

Appendix : Motion Sensor Interface

The description of the sensor interface software for the motion control system in Chapter 5 refers to the pieces of code in this appendix. The interface part, including the implementation of some member functions, for the base class for external sensors looks as follows (`template<T> class C...` declares the type `T` to be a generic data type in `class C`):

```
class MotionIO;  // HW interface specification.
class IO_spec;   // HW connection for the sensor.
class SimFunction;

template<class SensorValue> class Sensor {
  private:
    friend class SensorPair;
    Sensor();
    ~Sensor();

    enum behavior {real, dummy, simu};
    behavior tag;
```

```
    virtual SensorValue *Sample() =0;
    virtual SensorValue *DummySample() {
      SensorValue *ptr = new SensorValue;
      *ptr = DummyVal;
      return ptr;
    };

    SimFunction *simulated_sensor;
    SensorValue *GetSimuVal() {
      if (!simulated_sensor)
        return GetDummyVal();
      else
        return (SensorValue*)
                SimFunction::eval(*simulated_sensor);
    };
  public:
    // ******** Generic method for sampling: ********
    SensorValue *GetVal() {
      switch (tag) {
      case real:
        return Sample();
      case dummy:
        return DummySample();
      case simu:
        return Simulate();
      }
    };

    enum mode {off, single, trace, record};
    mode SetMode(const mode new_mode);

    void SetSimuFunc(SimFunction* simulate_sample) {
      delete simulated_sensor;
      simulated_sensor = new SimFunction(simulate_sample);
    };
    virtual void SetDummyVal(SensorValue* newDummy);
```

```
    virtual void Reset();
  protected:
    mode current_mode;
    SensorValue DummyVal;
    SensorValue CurrentVal;
};
```

The following class is used to ensure that both a real and a dummy sensor interface are instantiated. The pointers to the real respectively to the dummy sensors can then be put in structures, one for all real sensors and one for all dummy sensors, defining the entire sensor environment for each case. The control algorithms and precomputations can then use either of these without knowing which. This means that algorithms only have to consider the real case. The interface part for the SensorPair class is:

```
template<class SensorType> class SensorPair {
  private:
    friend class MotionSensors;
    SensorType* sensor;
    SensorType* dummy_sensor;

    SensorPair(IO_spec* HW_connection);
    SensorPair(SimFunction* simulate_sample);

    // Trick to force the compiler to check that
    // the actual SensorType really is a Sensor
    // (next line will show up in error message):
    void SensorType_not_derived_from_Sensor() {
      Sensor *p = sensor; p = dummy_sensor;
    };
};
```

Objects of type SensorPair behaves differently depending on if the sensor is simulated or real. The differences are handled by having two overloaded constructors as shown in the interface part. The implementation part for the constructors is simply:

```
template<class SensorType>
SensorPair::SensorPair(IO_spec HW_connection) {
  sensor = new SensorType(HW_connection);
  dummy_sensor = new SensorType();
  dummy_sensor->tag = Sensor::dummy;
};

template<class SensorType>
SensorPair::SensorPair(SimFunction* simulate_sample) {
  sensor = new SensorType(simulate_sample);
  sensor->tag = Sensor::simu;
  dummy_sensor = new SensorType();
  dummy_sensor->tag = Sensor::dummy;
};
```

The interface part of the sensor manager class is:

```
class MotionSensors
{ private:
    typedef Sensor* sensor_array[max_num_sensors];
    sensor_array* real_sensors;
    sensor_array* dummy_sensors;
  public:
    MotionSensors();
    MotionSensors(SimulationParameters SimPars);
    ~MotionSensors();

    // More code here...

};
```

The `MotionSensors` class uses a list of all types of sensors defined in the system. Each sensor is given a unique enumeration number and a string name that can be used to find the unique number. Sensors are then instantiated at run-time by calling the MakeSensor procedure last in the following sensor definition class:

```
class SensorDefs {
  public:
```

```
    enum available {OnOffSwitch,        // Digital one bit.
                    Tactile1,           // Tactile type 1.
                    Tactile2,           // Tactile type 2.
                    Scanner,            // Laser scanner.
                    // ...
                    Force,              // Force sensor.

                    // Add new sensors before this line.
                    END_MARK            // Must be last.
                   };

    static string name[END_MARK];
    name[OnOffSwitch] = "On/off one bit";
    // ...
    name[Force]       = "Force, standard"

    // Add new sensor names before this line.

    static SensorPair* MakeSensor(available sensor) {
      switch (sensor) {
        case OnOffSwitch:
          // ...
      };
    };
};
```